Detection and Mitigation of Security Threats in Cloud Computing

Tianwei Zhang

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Electrical Engineering
Adviser: Professor Ruby B. Lee

September 2017

Abstract

Infrastructure-as-a-Service (IaaS) clouds provide computation and storage services to enterprises and individuals with increased elasticity and low cost. Cloud customers rent resources in the form of virtual machines (VMs). However, these VMs may face various security threats.

This dissertation proposes a new architectural framework, *CloudMonatt*, to detect and mitigate potential security threats targeting customers' VMs in cloud computing. *CloudMonatt* monitors the security health of VMs and attests to customers if they are getting their desired security. It takes actions to mitigate the potential threats that can compromise the security properties requested. We design cloud management and security services, and define new hardware-software modules in cloud servers to provide the underlying measurements. We define secure communications protocols to guarantee that the monitoring service takes place in an unforgeable way.

To demonstrate how *CloudMonatt* can enhance the VMs' security, we consider a variety of threats and their defenses that can be integrated in *CloudMonatt*. We first consider threats on resource availability. We design a set of memory Denial-of-Service (DoS) attacks: an attacker VM can abuse the shared memory resources to significantly degrade a victim VM's performance. Then we statistically monitor VMs' resource consumption behaviors to detect these attacks, and use resource throttling to mitigate the availability threats.

Next, we consider subtle attacks on confidentiality, specifically cache side-channel attacks. An attacker VM can exploit a shared CPU cache to steal information from the victim VM. We collect VMs' micro-architectural behaviors and use a combination of signature and anomaly detection techniques to identify the existence of various side-channel attacks. We use targeted VM migration to eliminate these confidentiality threats.

Then, we consider attacks on system integrity within a VM. We show how to protect a VM's system integrity from malware, using Virtual Machine Introspection (VMI) to passively collect information for malware detection and also actively change the VM's execution paths to defeat the potential malware.

In summary, *CloudMonatt* is a general-purpose architecture for providing VM security monitoring and protection to cloud customers. We hope *CloudMonatt* can be a foundation for future work on protecting VMs' security health in cloud computing.

Acknowledgements

I would like to express my gratitude to my colleagues, friends and family, who gave me great support to complete this dissertation.

First and foremost, I would like to thank my adviser, Professor Ruby B. Lee. This dissertation would not be possible without her continuous support throughout my Ph.D. I am deeply impressed by her enthusiasm for research, attention to details and diligent work ethic. Her encouragement helped me overcome the difficulties in research. Her guidance helped me improve the skills of critical thinking, writing and presenting, which will bring significant benefits to my future career. I am extremely lucky to have Professor Lee as my Ph.D advisor.

There are several other people who gave me technical support in my research. I would like to express my sincere gratitude to Professor Jennifer Rexford who served as my dissertation readers. Their valuable feedback helps improve this dissertation. I would also like to thank Professor Niraj K. Jha and Professor Sharad Malik for serving as examiners for my Final Public Oral presentation. I am very grateful to Professor Yinqian Zhang from the Ohio State University, who has collaborated with me on two research projects. He provided me with new and priceless perspectives in research. He also served as one dissertation reader and offered useful suggestions to improve this dissertation.

It has been a great pleasure to work with some talented and enthusiastic people at the Princeton Architecture Laboratory for Multimedia and Security (PALMS). Jakub Szefer helped me start my first research project and write my first paper. He is a patient and responsible mentor. Pramod Jamkhedkar guided me in the research project of secure cloud computing. We worked together to set up the physical servers and cloud software framework, which is the foundation of this dissertation. Fangfei Liu and I shared most of our Ph.D years. I really appreciate discussions with her, and the research experience and codes she shared with me. I would also like to thank all

current PALMS members, Wei-Han Lee, Zecheng He and Guangyuan Hu, for the time they spent listening to my work and providing me with insightful comments.

I would like to thank the administrative assistants, Lori Bailey, Stacey Weber, Colleen Conrad, Roelie Abdi and Heather Evans for making everything easier. I would like to acknowledge the financial support for my research from National Science Foundation (NSF CNS-1218817) for the project "Cloud Security on Demand", and also from NSF 1526493 STARSS and SRC T3S.

I am really fortunate to have some best friends at Princeton University. I would like to thank Yaosheng Fu, Haotian Pang, Sen Tao and Yun Wang in the EE department for all their tremendous support and encouragement. I really enjoyed having dinner with them in the weekends. I would like to thank Li Chen, Qixing Ji, Borui Liu and Xin Teng. We had unforgettable memories together — hiking, road trips, playing cards, BBQ, just to name a few. I want to give my most sincere gratitude to my girlfriend, Xiaoyu Tang, for her endless love and support. I thank her for accompanying me through the tough Ph.D journey. I thank her for sharing the joy when I made achievements and comforting me when I was sad.

Last but most important, I want to thank my family. I would like to thank my parents, Xinling Zhang and Yuan Liu, and my sister, Lu Zhang, for their unconditional support both emotionally and financially over the years. They make enormous sacrifices to help me make the achievements today. I cannot imagine a life without their love and blessings.

To my parents.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Cloud computing offers large enterprises, small businesses and individuals new options for IT deployment. In this computing model, third party cloud providers maintain large-scale centralized computing resources and environments, and lease them out to customers. Customers can outsource their computation and data storage tasks to the cloud systems with great elasticity, low cost and high energy efficiency. Cloud computing introduces new characteristics (e.g., resource pooling and broad network access) to reduce operational costs for cloud providers, and offer high-quality services to customers. However, these characteristics can also introduce new security threats to customers' computations and data. This dissertation designs new architectures and methods to detect and mitigate security vulnerabilities in cloud computing. In this chapter, we provide background information on cloud computing, and the attack vectors in the cloud system.

1.1 Cloud Computing Definition

The National Institute of Standards and Technology (NIST) gives a definition of cloud computing as follows [156]:

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

1.1.1 Essential Characteristics

Based on this definition, cloud computing has several key features compared to traditional computing models:

On-demand resources. Cloud providers enable customers to specify the computing resources they need on demand, i.e., when they need it. Cloud providers typically adopt a "pay-as-you-go" pricing model, which charges customers based on the duration and amount of computing resources they use. This can greatly save customers upfront infrastructure costs, as they do not need to purchase and maintain physical servers.

Broad network access. Cloud computing is a type of Internet-based computing service that can be accessed via networks. Customers deliver their computation tasks and data to the remote datacenters, and manage them remotely. This provides flexibility to customers as they can access and share cloud services with others regardless of their locations or the devices they use to access cloud services.

Resource pooling. Cloud providers manage and categorize computing resources for different customers. The key technique for resource management is virtualization. Cloud providers adopt virtualization software and hardware to isolate and allocate physical resources to different customers efficiently. This helps achieve the feature of on-demand resources for customers, as well as maximize resource utilization and reduce costs for cloud providers.

Rapid elasticity. Cloud providers offer scalable services to customers. At runtime, customers can scale up their computing resources as their computing needs increase, or scale down when their computing needs decrease.

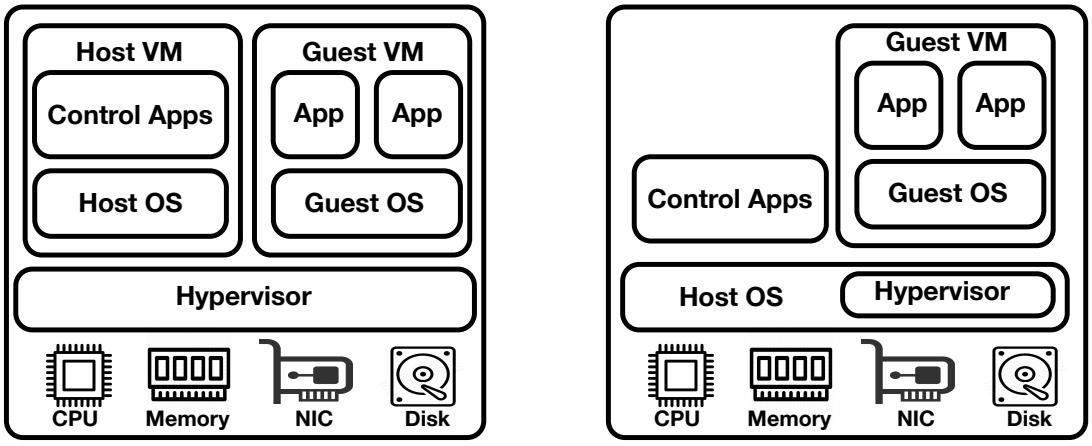
Measured service. Cloud providers utilize different metrics to monitor and measure the provision of services. These metrics can price the cloud services in the "pay-as-you-go" model. They can also achieve resource optimization and predictive planning: cloud providers can conduct automatic on-demand scaling and failure recovery based on these metrics.

1.1.2 System Support

The features of cloud computing are supported by the virtualization technique. Virtualization enables different operating systems to run concurrently on the same physical server. Each Operating System (OS) enjoys the same abstraction of having the entire machine, while they physically share the same server. Each OS runs within a Virtual Machine (VM). Virtualization is realized by both software and hardware.

Software support. In the software layer, virtualization is achieved by a privileged software called the hypervisor or the Virtual Machine Monitor (VMM). This software has several functions. First, it virtualizes the physical resources (CPU cores, memory, I/O devices, etc.) so that multiple VMs can run concurrently on one physical server. Second, it provides isolation between different VMs so each VM has its own CPU context and memory space. Third, it can manage VMs' activities, e.g., VM launch/termination, suspension/resumption, migration, etc.

There are two types of virtualization approaches [231]: (1) full virtualization is a technique that supports unmodified guest operating systems. The guest OSes can issue the same privileged instructions and sensitive calls as those running on real hardware. These instructions will be translated to executable instructions by the hypervisor (Binary Translation virtualization), or directly handled by the hardware (Hardware Assisted virtualization). (2) Para virtualization is another technique in which the guest operating systems need to be modified to issue special hypercalls to communicate directly with the hypervisor. This is called OS Assisted virtualization.



(a) Type-1 bare-metal virtualization platform (b) Type-2 hosted virtualization platform

Figure 1.1: Two types of virtualization platforms

There are also two types of hypervisors: (1) a Type-1 native hypervisor runs directly on the host's hardware to control a privileged host VM and other guest VMs. (2) A Type-2 hosted hypervisor runs inside an operating systems (called host operating system). It manages the guest VMs from the host operating system. Figure 1.1 shows the abstract structures of these two types of hypervisors.

Hardware support. In addition to software, new hardware is also designed to support virtualization and enable the rapid growth of cloud computing. First, with the development of multi-core processors, more and more processing units can be integrated on the same server. This increases the cloud server's capability to host more VMs concurrently, improving resource utilization and power efficiency.

Second, processor vendors add hardware virtualization extensions into their processors (e.g., Intel VT-x [9], AMD-V [5]). These extensions introduce new hardware components, instructions and execution modes to handle virtualization functions. For instance, Intel VT-x has several new features [9]. (1) VT-x introduces two operation modes: VMX root operation, which is a fully privileged mode and intended for the hypervisor; VMX non-root operation, which is not fully privileged and intended for the guest VMs. (2) VT-x introduces a Virtual Machine Control Structure (VMCS) for

each VM to manage and store its non-root operations and VMX transitions. (3) Intel VT-x uses the Extended Page-Table (EPT) hardware to support the virtualization of physical memory. (4) VT-x includes new instructions to handle VMCS operations and memory management. AMD-V has similar functions [5]. These hardware extensions can significantly accelerate the virtualization speed and improve performance compared to purely software solutions.

1.2 Cloud Computing Models

1.2.1 Deployment Models

Cloud computing offers different deployment models to deliver cloud services. These deployment models represent different cloud environments and usage scenarios.

Private cloud. In a private cloud model, the cloud infrastructure is provisioned for a single organization. This organization usually has dynamic or unpredictable computing needs, or requires direct control over the computation environment. Generally the private cloud system is installed behind firewalls under the control of its organization, so it only permits access by authorized customers from this organization. Note that a private cloud can be configured internally by the organization itself, or externally by a third-party cloud provider.

Community cloud. A community cloud is mutually shared among several organizations from a specific community with common concerns (security, compliance, jurisdiction, etc.). The cost is shared by these organizations so that this model can save more cost than private clouds. Similar to private clouds, a community cloud can be run and managed internally by the community, or externally by a third-party provider.

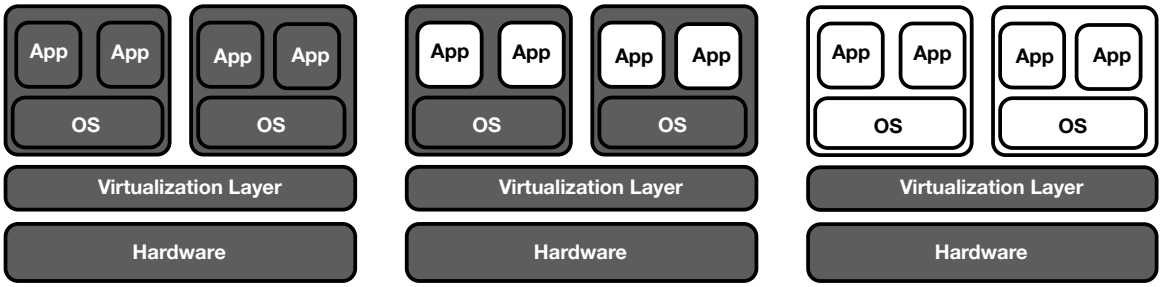
Public cloud. A public cloud delivers its service for public use. The whole system is usually shared by various organizations. Any customer with a credit card can pay to rent cloud services under the "pay-as-you-go" pricing model. The customers do not have direct control over the cloud system. A public cloud is usually run by a commercial cloud provider, e.g., Amazon Web Services, Microsoft Azure, Google Compute Engine, Rackspace, etc.

Hybrid cloud. A hybrid cloud combines two or more clouds, offering the benefits of multiple deployment models. It allows a customer to extend the capacity or capability of a cloud service by aggregation, integration or customization with another cloud service. For instance, in a hybrid cloud composed of a private and a public cloud, the organization can store protected data in its private cloud, and leverage computing resources from the public cloud to run applications that rely on the data. This keeps data exposure to a bare minimum because it is not storing sensitive data for long in the public cloud component.

1.2.2 Service Models

In addition to deployment models, cloud computing also has different service models. NIST defines three standard service models as Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS). Figure 1.2 shows the abstract architectures of these three service models. Inside the figures, gray blocks are managed by the cloud provider and white blocks are managed by the customers. We describe these services separately.

Software-as-a-Service. In the SaaS model, the cloud provider offers applications software to customers as on-demand services. Specifically, the cloud provider manages the infrastructure and platforms, installs applications software on the cloud servers, and grants customers network accesses to the applications software. In this way, cloud customers can use the applications directly via web browsers or program interfaces,



(a) SaaS (b) PaaS (c) IaaS

Figure 1.2: Three cloud service models. Grey blocks are controlled by the cloud provider and white blocks are controlled by the customers

without the need to install or configure applications on their own computers. Popular cloud applications include email, storage, social networks, etc.

With the help of virtualization, the cloud provider can run multiple copies of one application in different virtual machines to achieve scalability. Besides, one cloud server can host multiple different cloud applications to maximize resource utilization.

Platform-as-a-Service. The PaaS vendor provides programming environments to customers for running their applications. In this model, the cloud provider configures and delivers computing platforms and environments, e.g., OSes, programming libraries and databases, to customers, as a service. Then cloud customers can directly develop and run their own applications inside this platform, without the complexity of building and maintaining the underlying infrastructure such as servers and OSes. The cloud provider uses the virtualization technique to provide context isolation and allocate resources based on the application's demands.

Infrastructure-as-a-Service. The cloud provider leases an entire virtual machine (OSes, CPUs, memory, storage and networking) to customers, so they do not need to purchase physical servers or network devices. When launching VMs, customers can specify the desired computing resources. These options include the number of CPU cores, memory size, disk size, operating systems, etc. Then the cloud provider selects physical host servers and boots the VMs with the specified configurations on these

host servers. After the VMs are booted up, customers can access them remotely to deploy and run arbitrary software inside their VMs.

The cloud customers are able to request the cloud provider to suspend, resume or terminate their VMs at anytime during the VMs' lifecycles. Besides, the cloud provider may migrate customers' VMs to different servers for energy optimization or fault tolerance. Nowadays the cloud provider usually uses the live migration technique in which the VMs can continue execution during the migration process. This live migration can achieve near zero downtime, incurring negligible performance overhead to the migrated VMs, and customers are not aware that their VMs are being migrated.

1.3 Attack Vectors in Cloud Computing

Cloud computing introduces new features to ease IT deployment for enterprises and individuals. However, these new features also amplify existing vulnerabilities and create new vulnerabilities. The Cloud Security Alliance has identified several common threats that can compromise customers' computations and data in clouds [102]. We classify these threats into different categories based on the attack vectors. Figure 1.3 shows the abstract architecture of a cloud system with the potential attack vectors.

Service interface. To use a cloud service, a customer needs to register an account on the cloud provider's website. Then he can login to his account to manage the cloud service. However, some cloud systems lack strong user identification and authentication. This gives attackers opportunities to hijack customers' accounts, and thus access or compromise the critical areas of cloud services. Also, customers interact with the cloud services with some User Interfaces (UIs) or Application Programming Interfaces (APIs). If these interfaces are not designed correctly, an attacker can easily exploit the vulnerabilities to steal credentials or compromise the cloud services.

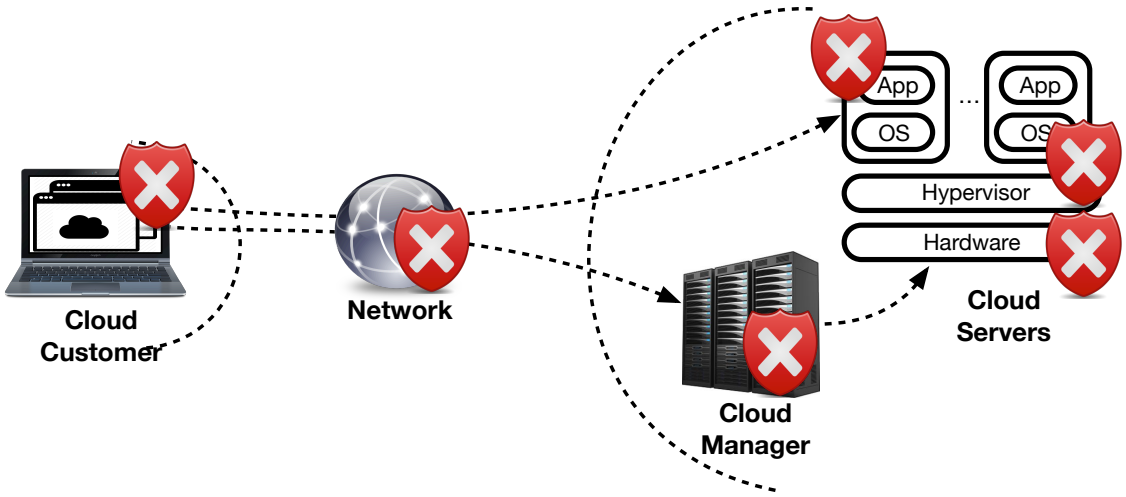


Figure 1.3: Attack vectors in cloud computing

Networks. Cloud computing requires customers and end-users to access cloud services via networks. If the networks are not well protected, attackers can steal sensitive data (e.g., personal health information, financial information, intellectual property) during transmission. Besides, attackers can conduct Denial-of-Service (DoS) attacks to prevent customers or end-users from accessing the data or applications in the clouds.

Cloud managers. The cloud managers have administrative privileges to manage all the cloud infrastructure and services. Untrusted or compromised cloud managers can introduce great security risks to the customers in two ways. First, accidents happening on a cloud system can cause permanent loss or leak of customers' data. Such accidents include accidental data deletion by cloud managers, or a physical catastrophe such as a fire or an earthquake. Second, a malicious insider can easily access sensitive information stored in the clouds, or compromise the cloud's key management.

Virtualized system. System vulnerabilities in the cloud servers or virtual machines can be exploited by attackers to intrude into the servers or VMs to steal data, take control of the entire server system or disrupt service operations. First, cloud servers introduce a new virtualization layer, which enlarges the attack surface. The hypervisor

has a large codebase size and inevitably contains security bugs that enable attackers to compromise the servers [173]. Second, vulnerabilities within the guest operating systems inside virtual machines also put the security of customers' data and services at significant risk.

Shared infrastructure. The cloud provider allocates different customers' data, applications or virtual machines on the same server. The sharing of the underlying components and resources can lead to new cross-domain vulnerabilities, for a multi-tenant infrastructure (IaaS), re-deployable platforms (PaaS) or multi-customer applications (SaaS). A malicious domain can exploit the shared components to steal sensitive information from another co-located domain. It can also abuse the shared resources to conduct host-based Denial-of-Service attacks to compromise other domains' resource availability.

Cloud services. Cloud computing provides elastic and low-cost cloud services. However, malicious customers can abuse these cloud services to conduct attacks. These customers can first acquire a large amount of cloud resources via free cloud service trials or credit card fraud. Then they can abuse the cloud resources to attack other customers, organizations or the cloud provider. Typical attacks include Distributed DoS attacks, large-scale email spam and phishing attacks, brute-force password guessing attacks, port scanning attacks, etc.

In Section 2.1, we will describe detailed example attacks for each attack vector.

1.4 Dissertation Summary

Given the severity of cloud threats and the large attack surface, this dissertation aims to design a secure cloud system that can detect and mitigate potential security threats against customers' applications and data, thus providing a secure cloud environment for customers.

Scope. We focus on *public clouds* with the *Infrastructure-as-a-Service* (IaaS) model. Public clouds are open to the public and allow different organizations and individuals to share the same system. So they have more complicated environments and more security issues than private clouds. We focus on the IaaS service model as other types of service models can be built on top of IaaS. So the security protection architecture and methods presented in this dissertation can also be applied to other types of cloud deployment and service models.

We consider the security threats caused by the shared infrastructure, and the virtualized system. These are unique to cloud computing and need specific cloud-based solutions. Some other types of threats (e.g., service interface, networks) are common in traditional computing models and have been well studied. So they are not in the scope of this dissertation. We aim to build secure cloud systems for customers, so we trust the cloud managers and customers, and do not consider the threats caused by malicious service providers or customers.

Summary. We design an end-to-end IaaS cloud architecture to protect customers' virtual machines based on their demands. In current public clouds, customers have different performance needs and they can choose computing resources and configurations to match their needs. They may also have different security needs for their computations and data in the clouds. However, current Service Level Agreements (SLAs) in public clouds do not provide security specifications and the corresponding services to satisfy customers' security needs. Our architecture provides on-demand security services for the cloud customers. It enables customers to request different security properties for their VMs. Then it provides persistent security protection for each VM throughout its life-time in the clouds.

In Chapter 3 we introduce the *CloudMonatt* architecture. *CloudMonatt* provides a novel security service to cloud customers: it allows customers to specify their desired security properties for virtual machines, and then monitors the security health of the

VMs during their lifetime in the cloud. Once *CloudMonatt* detects that the VMs are facing potential threats that violate the customers' security properties, it automatically conducts remediation actions to eliminate the threats and keep the VMs healthy. We show the basic and necessary settings and requirements to realize the monitoring functionalities in the IaaS cloud context. We also show how to perform translations between high-level security properties specified by the customers, and the low-level platform measurements. We use several case studies implemented in *CloudMonatt* to exemplify the usage of this monitoring service.

In the next three chapters, we detail the security mechanisms that can be adopted by cloud providers to defeat different threats. These can all be integrated into the *CloudMonat* framework. Chapter 4 considers *availability* threats and Chapter 5 considers *confidentiality* threats. Both of these two types of threats come from the shared infrastructure. We design novel methods to detect and mitigate such vulnerabilities, using statistical tests and existing hardware features. Chapter 6 considers system *integrity* threats from the guest OS in a leased virtual machine. We adopt the Virtual Machine Introspection (VMI) technique to defend against these vulnerabilities.

More specifically, Chapter 4 introduces availability threats caused by shared cloud servers, and provides a defense solution. In particular, we discuss memory Denial-of-Service attacks. We find that a hostile VM can intentionally generate memory resource contention and significantly degrade the performance of the victim VM co-located on the same server. We design a set of attack techniques that target different layers of hardware memory resources to enhance the attack effects. We evaluate these attack techniques in the lab as well as in public cloud settings. We then design and implement a novel and effective approach to detect and mitigate all known types of such availability attacks with small overhead. Our detection strategy provides a generalized method for detecting deviations from the normal behavior of the protected

VM, by statistically comparing reference and monitored probability distributions of important runtime measurements. Once the malicious VM is detected, we use a resource throttling scheme to reduce the attacker's execution speed to minimize the attack effects.

Chapter 5 introduces a new method to defeat one type of confidentiality threat: cache side-channel attacks on multi-tenant cloud servers. In this type of attacks, a hostile VM can generate contention on the CPU cache and then steal a victim VM's critical information through observing the victim's footprint on the shared cache. We observe that the attacker VM always has anomalous cache behaviors when the information is leaking from the victim to the attacker. To detect the existence of such information leakage, we first use signature-based detection to identify the critical moment that the victim is accessing sensitive information, which is also the moment that the attacker can steal sensitive information from the victim. Then we use anomaly-based detection to check if the attacker VM has anomalous cache behaviors at this critical moment. By doing so, we are able to detect the cache side-channel attacks with high fidelity. Once an attacker is detected, we eliminate this confidentiality threat by migrating the attacker VM to a different processor package or cloud server.

Chapter 6 considers the system integrity threats *within* the guest VMs. Customers' VMs may face different integrity vulnerabilities at runtime, e.g., kernel-level rootkits, user-level malware, or network-level attacks. We summarize the features of these vulnerabilities and the corresponding defense solutions. Then we exploit the Virtual Machine Introspection method to design security tools to protect the VMs from these vulnerabilities. These security tools are located in the hypervisor layer. They monitor the VMs' memory, disk and networks for malicious activities and actively change the VMs' executions or recover the compromised data to defeat the potential attacks. This enables the cloud provider to protect the VMs' system integrity for customers.

Organization. The dissertation is organized as follows: Chapter 1 provides the background information about cloud computing and its security issues. It also summarizes the contribution of this dissertation. Chapter 2 discusses past work with regard to security threats and security protection mechanisms in cloud computing. Chapter 3 introduces *CloudMonatt*, our cloud security health monitoring framework, architecture and protocol. We also present the evaluation and security verification of this architecture. Chapter 4 considers the availability property. We present memory DoS attacks and the corresponding defense against these attacks. Chapter 5 considers confidentiality attacks. We present a novel defense against cache side-channel attacks in the cloud. Chapter 6 considers the integrity of VMs from a compromised guest OS. We present methods to monitor and protect virtual machines' system integrity. Chapter 7 concludes this dissertation and discusses future work.

Chapter 2

Past Work

Chapter 1 briefly discusses the cloud computing background and potential security threats existing in cloud systems. In this chapter, we review the past work related to cloud computing security. We systematically describe the existing cloud-based attacks and vulnerabilities discovered by researchers, as well as the new approaches and architectures designed in prior work to defeat these attacks and enhance cloud system security (Section 2.1). We describe the security threats we consider in this dissertation and compare our methods proposed with the past work. We also review the secure cloud platforms from commodity products and past research literature, and compare them with the work we propose in this dissertation (Section 2.2). Specific past work relevant to each chapter will be discussed in that chapter.

2.1 Security Threats and Protections in Cloud Computing

Traditional cyber security threats, e.g., network vulnerabilities, operating system attacks, are also common in cloud computing. Besides, the unique features of cloud computing introduce new risks to public cloud providers and customers. First, cus-

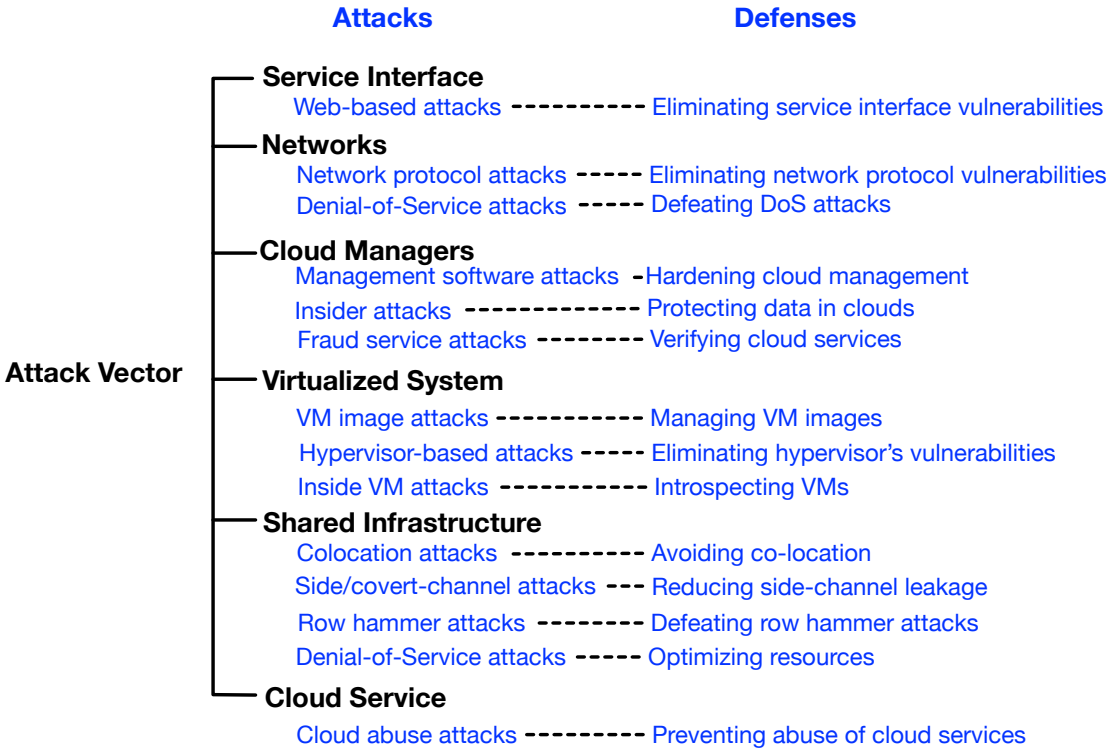


Figure 2.1: Cloud-based attacks and defenses.

tomers need to entrust the cloud providers to deploy computation and storage tasks for them. So the security states and resource management of the cloud services are not transparent to or controllable by customers. Customers heavily rely on the cloud providers to protect their data and computation. Second, cloud systems usually adopt the *multi-tenancy* feature, where cloud services belonging to different customers are allocated on the same platform. This feature can efficiently improve the whole system's resource utilization and reduce the operational costs. However, this can bring new vulnerabilities due to the infrastructure sharing. Third, the cloud providers usually use the virtualization technique to manage resources. This extra software layer can make the systems more complicated and add new attack vectors.

In this section, we review the existing cloud vulnerabilities from past work. We also talk about the defense solutions proposed in prior work to eliminate these cloud-based vulnerabilities. Basically we categorize these vulnerabilities and defenses based on

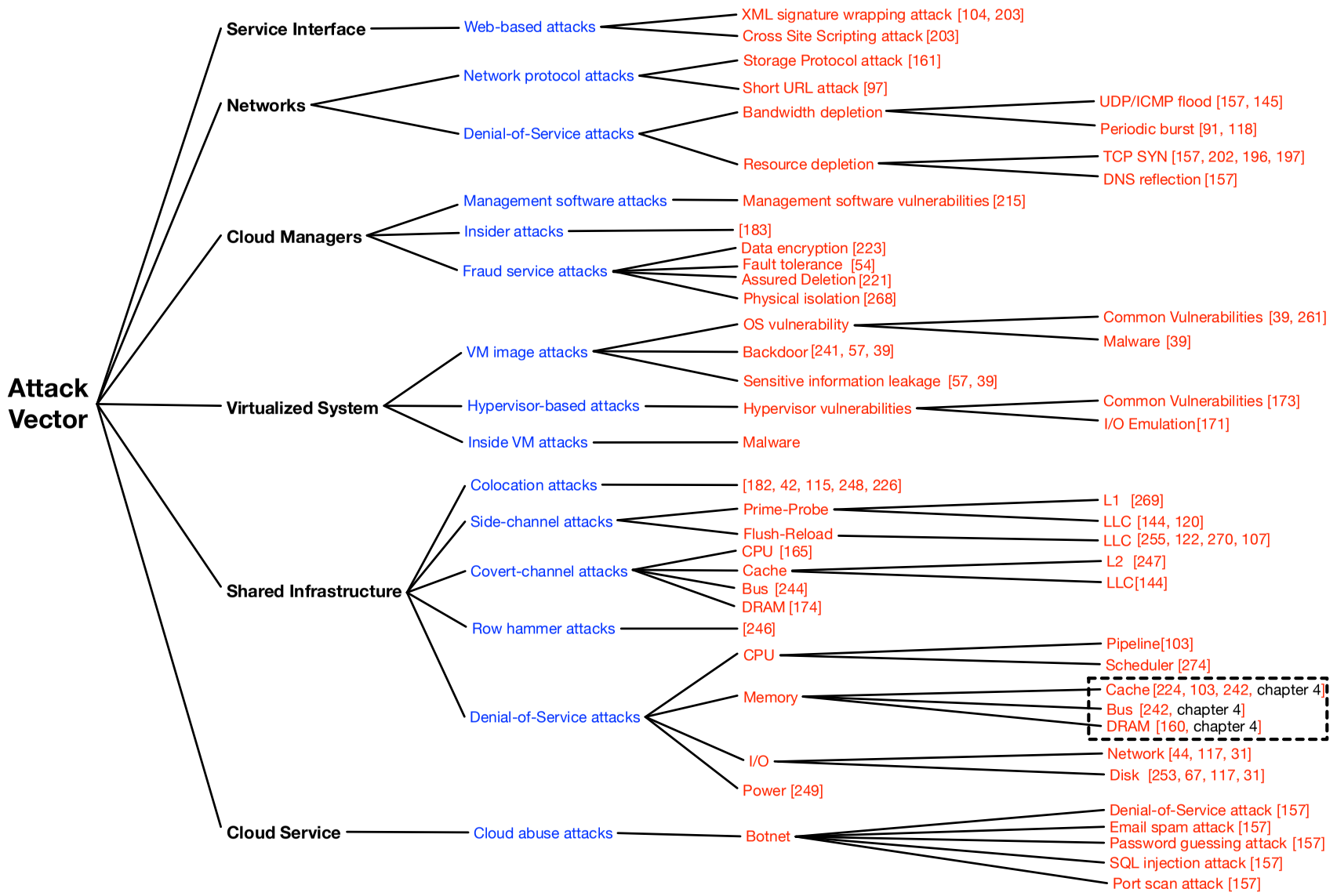


Figure 2.2: A taxonomy of cloud-based attacks discussed in this chapter. Dashed boxes show the new work we did in this dissertation.

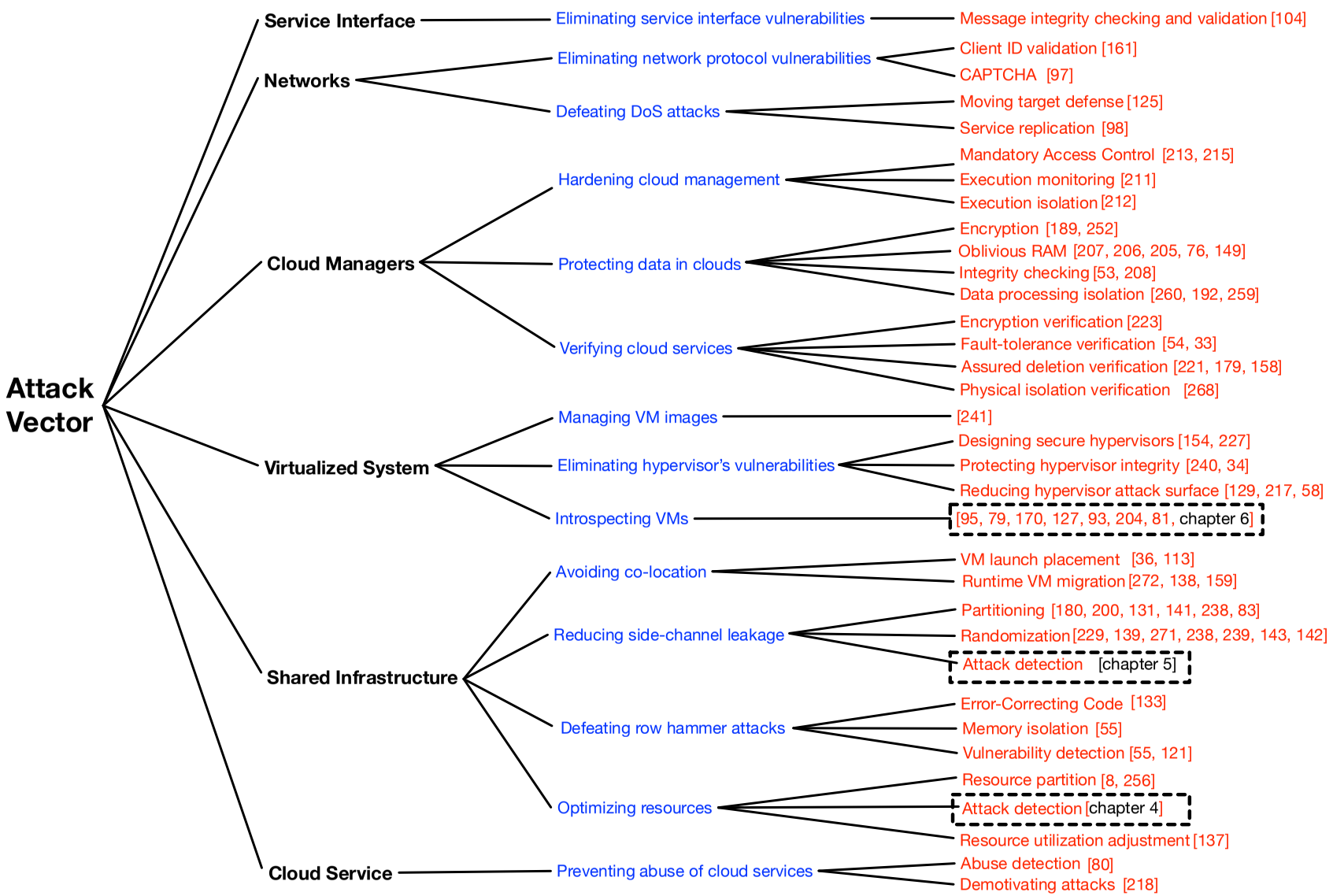


Figure 2.3: A taxonomy of cloud-based defenses discussed in this chapter. Dashed boxes show the new work we did in this dissertation.

the attack vectors, as described in Section 1.3. Figure 2.1 shows the taxonomy of the attacks and defenses we are going to discuss in this chapter. Figures 2.2 and 2.3 show the detailed attacks and defense approaches.

2.1.1 Service Interface

Customers access and manage cloud services via the cloud providers' websites or APIs. If these cloud service interfaces are not designed securely, attackers can exploit their bugs and vulnerabilities to manipulate customers' cloud usage, and hijack customers' accounts.

2.1.1.1 Service Interface Attacks

Web-based attacks. XML signature wrapping attacks were demonstrated to compromise the control interfaces of popular cloud services (e.g., Amazon EC2 [104, 203], Amazon S3 [203], Eucalyptus cloud software [203]). Basically customers and cloud providers' websites use Simple Object Access Protocol (SOAP) to exchange messages. Customers generate SOAP messages including their desired actions as well as the signatures of each action in the XML format, and send the messages to the cloud websites. The cloud websites execute the operations if these signatures are valid. However, an attacker can intercept a SOAP message, move the customer's actions to a "Wrapper" structure, and add malicious actions. As the customer's actions have not been modified but just moved, their signatures are still valid. However, the website will also conduct the attacker's malicious actions in addition to the original customers' ones. With this technique, attackers can inject arbitrary cloud control operations for execution, as if they come from the customers.

Somorovsky et al. [203] discovered Cross Site Scripting (XSS) attacks in the control interface of Amazon Web Services. Basically they observed that the Amazon online shop has the same credentials and login session as the Amazon Web Services. Then

an attacker can inject arbitrary HTML codes to the Amazon online shop website. The injected codes allow attackers to extract and steal customers' cookies of the online shop, which are also the cookies of the Amazon Web Services. Using these cookies, the attacker can break into the customers' accounts of Amazon Web Services and steal sensitive data.

2.1.1.2 Service Interface Defenses

Eliminating service interface vulnerabilities. Vulnerabilities from cloud service interfaces can be exploited by attackers to hijack cloud accounts. So it is necessary to reduce such threats in the web interfaces and APIs. Prior work to defeat web-based vulnerabilities can also be used to enhance the security of the cloud interface. For instance, to mitigate the XML signature wrapping attacks against cloud interfaces, Gruschka and Iacono [104] proposed a method to validate SOAP messages in two steps. First they validated the XML schema by checking the integrity of both the SOAP headers and bodies. Second, they proposed security policies related to wrapping attacks and verified if these policies were satisfied for the cloud web interfaces. These checking procedures can prevent attackers from modifying SOAP messages. Input validation can also help protect the service websites from XSS attacks.

2.1.2 Networks

As customers need to access their data and computations in the clouds via networks, the networking system can also bring different types of security threats to the cloud services.

2.1.2.1 Network Attacks

Network protocol attacks. Network protocols provide data encryption, integrity protection, and identity authentication for the cloud services. However, if the commu-

nication protocols are not designed correctly, adversaries standing between customers and the cloud systems can impersonate customers and gain unauthorized accesses to their data stored in the clouds. Mulazzani et al. [161] discovered three such attacks against the SaaS storage application, Dropbox. (1) Direct download attack: in Dropbox, customers fetched file blocks by submitting their host IDs and the hash values of these blocks. However, Dropbox did not check the host IDs' permissions to these blocks. So an attacker could exploit this vulnerability to download any data, as long as he had an arbitrary ID (not necessarily the ID of the file owner), and the hash value of the requested data block. (2) Stolen host ID attack: Dropbox used a unique host ID to link a customer's devices to his Dropbox account. If this host ID was stolen by the attacker, all this customer's files could be linked to the attacker's account, and downloaded by the attacker. (3) Hash value manipulation attack: Dropbox used the de-duplication technique to merge multiple data blocks with the same hash value from different files into one. An attacker could deliberately counterfeit a fake hash value and upload it to the Dropbox. If a block with the same hash value from another customer existed, then Dropbox created a link from this data block to the attacker. Then the attacker could gain unauthorized access to this data block. These vulnerabilities were fixed by Dropbox.

Georgiev and Shmatikov [97] demonstrated a set of attacks to steal critical files from Microsoft OneDrive and geo-location information from Google online map. SaaS applications usually used short URLs to replace long URLs to denote file locations in networks. An adversary could simply scan all the possible short URLs to identify the valid ones that direct to file locations. Accesses to these short URLs would be redirected to the long URLs which could reveal customers' critical information and files, e.g., user locations, photos, etc.

Denial-of-Service attacks. In the traditional computing model, one popular attack is a (Distributed) Denial-of-Service (DoS) attack, where the attacker tries to induce a

huge amount of network traffic from many different sources to the victim machine, saturating its network bandwidth and depleting its network resources. Such a threat is more prominent in cloud computing as the availability of the cloud services heavily depends on the network availability,

DoS attacks can be classified into two categories. The first one is bandwidth depletion, where the attacker tries to use up the network bandwidth and delay the victim's traffic. Two common techniques of bandwidth depletion DoS attacks are UDP flood and ICMP flood [157]. Liu [145] proposed a DoS attack in which an attacker could figure out the network topology and the bottleneck links in a cloud networking system. Then the malicious VMs can generate a large amount of UDP traffic to saturate these bottleneck links. Feng et al. [91] proposed Shrew attacks, which invoked periodic bursts of network traffic. Such burst traffic could saturate the network bandwidth in the cloud system and affect the victim's resource availability, without being detected by conventional detection approaches. Idziorek et al. [118] designed Fraudulent Resource Consumption attacks, which sent burst network traffic to the victim in a subtle way, making the target victim consume more network bandwidth and pay more money for their VMs.

The second approach is resource depletion, where the attacker tries to deplete the network resources on the victim host servers. Two common techniques of such attacks are TCP SYN flood and DNS reflection [157]. Somani et al. [202] proposed Economic Denial of Sustainability attacks, to cause economic loss of the target victim under the network resource depletion DDoS attacks. Shea and Liu [196, 197] evaluated the resistence of different benchmarks on different virtualization platforms against TCP SYN attacks.

2.1.2.2 Network Defenses

Eliminating network protocol vulnerabilities. Network protocols for cloud applications must be carefully designed to prevent attackers from compromising the communications between customers and their services. Mulazzani et al. [161] suggested to include validation of host IDs in the network protocols for secure authentication. Georgiev and Shmatikov [97] suggested to introduce CAPTCHAs or other methods in the protocols to separate human users from malicious automated scanners.

Defeating DoS attacks. Common solutions of defeating DoS attacks include intrusion detection systems, firewalls, etc. Besides, researchers have proposed new methods to effectively mitigate DoS vulnerabilities using the elastic computing resources from the cloud systems. Jia et al. [125] used cloud services to build a moving target defense mechanism to mitigate DDoS attacks. Upon detecting DDoS attacks against target instances in a cloud system, they quickly instantiate new instances at different network locations and replicate the attacked instances to the new ones while shutting down the attacked ones. This can effectively restrict attackers' target into a small set of replica instances and separate benign clients away from them. Gilad et al. [98] proposed Content Delivery Network (CDN) on Demand to defeat DDoS attacks against web applications at low cost. The idea is to replicate protected web pages to different web servers located in different cloud systems, and assign the nearest web server to each client. Each cloud system has a manager to monitor the replicated web servers. If one manager identifies that some web servers can not respond normally, it will activate the CDN-on-Demand mechanism and deploy new servers in the cloud system to protect the availability of web services.

2.1.3 Cloud Managers

Customers outsource computations and data storage to the public clouds. They entrust the cloud providers to protect their applications and data. An untrusted cloud manager can bring significant security risks to customers. It can steal or tamper with customers' sensitive data, or compromise service availability by disabling servers and networks.

2.1.3.1 Cloud Manager Attacks

Management software attacks. There exists vulnerabilities in the management software that could be exploited by the attackers to compromise the cloud manager. Sze et al. [215] discovered some vulnerabilities inside the OpenStack cloud software that enabled an attacker who took control of one computing server to extend his controls over the entire cloud system, affecting other VMs on other computing nodes or even bringing down the whole cloud. They demonstrated three attacks: (1) The attacker could extract the credentials and tokens of each customer who had VMs on this computing server. Then the attacker could manage the customer's cloud services and resources arbitrarily. (2) The attacker could generate fake messages and send them to any other computing servers to control the VMs on these servers. They could also change the root-password of the VMs. (3) The attacker could falsify VM status and resource utilization of any VMs on any compute nodes to the cloud manager. This fraudulence could change the cloud manager's VM/resource management and scheduling decisions, causing havoc to the whole cloud system.

Insider attacks. Rocha and Correia [183] demonstrated practical insider attacks in IaaS. If a malicious insider had root privileges on a cloud server, it could easily steal customers' confidential data inside the VMs. They presented four attacks: (1) the insider could create the memory snapshot of the victim VM, and retrieve cleartext

passwords from the snapshot. (2) Similarly, the insider could steal private keys from the VM snapshot. (3) In addition to the memory, the insider could also clone the VM's disk and extract confidential data from the disk image. (4) If the cloud server is protected, the insider could migrate the VM to a vulnerable server and conduct the above insider attacks.

Fraud service attacks. The cloud providers offer high-performance and security services to customers at higher prices. These include data encryption [223], fault tolerance prevention [54], assured deletion [221] and physical resource isolation [268]. However, deployment of such services is not transparent to customers. Even if the cloud providers commit to providing these services in the service level agreement (SLA), dishonest providers may cheat the customers who purchase these services, and do not actually apply the services. They are motivated to do so because this can save operational cost and it is hard for customers to find if their VMs have the desired protection.

2.1.3.2 Cloud Manager Defenses

Hardening cloud management. How to enhance the security of management activities is an important task for the cloud providers. Generally there are three directions. The first one is access control. Sun et al. [213] outlined a strawman design of a security cloud operating system, which uses Mandatory Access Control (MAC) to control cloud management services and customers' instances, and prevent attacks from propagating across cloud services. Similarly, Sze et al. [215] applied Mandatory Access Control and capabilities to confine interactions among different modules in the cloud management software. This can prevent malicious cloud servers from compromising the controller server and other trusted cloud servers.

The second one is execution monitoring. Sun et al. [211] designed CloudArmor to detect and prevent malicious cloud management activities conducted by adversaries.

CloudArmor monitors the syscall traces of each cloud service command, and establishes training models for these commands. When an adversary compromises one command and injects malicious behaviors, the syscall trace will deviate from the correct one, and CloudArmor will detect and reject the compromised command.

The third direction is secure execution isolation. Sun et al. [212] designed Pileus to protect and isolate critical management operations. Pileus splits each cloud operation into different event handlers, and allocates them to trustworthy cloud nodes that satisfy the cloud security policies. This can effectively minimize the risk of attack from other users.

Protecting data in clouds. Encryption can be used to protect data confidentiality in the cloud storage. Santos et al. [189] designed Excalibur, which uses Attribute-based Encryption to protect data in the clouds. Excalibur allows customers to encrypt data with customer-defined policies. Then a centralized monitor is introduced to check each server's configurations. Only the servers that satisfy the policies can decrypt and access customers' data. Yang et al. [252] used a similar idea to achieve fine-grained access control in cloud storage. Customers divide their data into several blocks and encrypt each block with a different key. These keys are encrypted by the Attribute-based Encryption, associated with certain access structures defined by the data owner. Only the users whose attributes satisfy the access structure in the ciphertext can decrypt the keys and obtain different granularities of information.

Encryption alone is not sufficient to guarantee confidentiality as data access patterns can also leak sensitive information. Researchers proposed to use Oblivious RAM (ORAM) to eliminate this vulnerability. In ORAM, the sequence of physical addresses accessed is independent of the actual requested data, so the cloud provider and customers can access the encrypted data while completely hiding data access patterns. Past work proposed different methods to optimize its performance and enhance its security. To reduce the bandwidth overhead, distributed ORAMs are

introduced, which partition an ORAM into different parts and distribute these parts to different locations so access requests can be distributed and served in parallel [207, 206, 205]. To reduce the access response time, novel schedulers are designed, which schedule the tasks of serving request and data processing in different priorities [76]. To enable data sharing, access control is introduced to ORAM systems [149], in which different users with different permissions can access the data based on the access control rules defined by the data owner.

Data integrity in the cloud storage is also important. Bowers et al. [53] designed HAIL to enable file integrity checking. The cloud customer distributes a file with multiple replicas across a number of servers. To check the integrity of this file, he can randomly choose a data block and fetch it from all the servers. Inconsistency in a small portion of blocks indicates that the integrity of these blocks are compromised and the cloud provider can recover these blocks based on other intact ones. Stefanov et al. [208] designed the Iris system, which exploits the Message Authentication Code (MAC) and Merkle-tree to provide scalable authentication of file integrity in the clouds.

Customers are concerned about the security of their sensitive data outsourced to untrusted public clouds. Past work have proposed methods to create isolated environments for data processing and storage in untrusted cloud environments. Zhang et al. [260] designed Sedic to preserve data confidentiality using Hybrid clouds. Sedic splits the computations and keeps tasks associated with sensitive data in the organization's private clouds while moving the rest to the public cloud. Schuster et al. [192] designed VC3 to achieve the same goal. VC3 exploits Intel SGX technique to create trusted isolated memory regions for customers' data and code and prevent them from being stolen or compromised by untrusted clouds. Zhai et al. [259] designed CQSTR, which creates cloud containers to manage software execution and control data flow. CQSTR enables customers to specify security policies and then use the cloud containers to prevent data leakage and misuse against untrusted cloud applications.

Verifying cloud services. Past work designed methods for customers to verify whether their cloud services are under certain protection as the cloud provider claimed. The first example is data confidentiality, where customers want to verify if their data are encrypted by the cloud provider and stored as ciphertext in the clouds. Dijk et al. [223] designed Hourglass to achieve this goal. The key element in Hourglass is the hourglass function. This hourglass function can convert input data to output data (i.e., hourglass format) and convert output data back to input data. However, it takes a long time to transform input to the hourglass format and a short time to transform the hourglass format back to the input. In Hourglass, the cloud provider is required to encrypt customers' data, convert the ciphertext to the hourglass format using the hourglass function, and store the hourglass format of ciphertext in cloud. When a customer wants his data, the cloud provider needs to convert the hourglass format of ciphertext back to the ciphertext, and then decrypt the ciphertext to get the plaintext for the customer. To verify data encryption, the customer can randomly select a data block and ask the cloud provider to show the hourglass format of encrypted ciphertext of this block. A honest cloud provider can directly fetch the hourglass format in its storage. A misbehaving cloud provider who only stores the plaintext of customers' data needs to take a long time to encrypt the data block and calculate the hourglass format of the ciphertext on-the-fly, which will be observed by the customer.

The second example is data fault-tolerance, where customers want to verify if the cloud provider stores multiple replicas of their files for fault-tolerance. Bowers et al. [54] designed Remote Assessment of Fault Tolerance (RAFT) to achieve this goal. In RAFT, the customer can challenge the cloud provider to retrieve a set of random blocks from one of his files. If a honest cloud provider stores multiple replicas of this file, it can fetch different blocks from different replicas. This task can be done concurrently in a very short time. If a misbehaving cloud provider only stores one copy of this file, then it has to fetch the blocks one by one from this single copy,

leading to a much longer response time. By measuring the response time, the customer can verify if his file is stored with multiple replicas. Armknecht et al. [33] proposed Mirror, a method to achieve the same goal. Mirror shifts the burden of constructing replicas from customers to the cloud providers and therefore saves customers' network bandwidth and cost.

The third example is assured deletion, where customers want to verify if the data they deleted previously have been permanently removed from the cloud storage. Tang et al. [221] proposed FADE, a system to achieve assured deletion using policy-based cryptographic operations. Customers encrypt their data using a *data key*, which is further encrypted with a *control key* associated with certain policies (e.g., access control, expiration time). When customers try to delete the data in the clouds, the *control key* will also be permanently destroyed so the plaintext of the data will never be leaked. Rahumed et al. [179] improved this system by adding version control functionality. Each file is split into different blocks, and each block is encrypted with a different key. Then different versions of one file can share the same common blocks. When one version is removed, its unique blocks will also be assured deleted, while the common blocks will still be kept for use. Mo et al. [158] further improved the assured deletion protocol to reduce the number of protected keys. They designed a Recursively Encrypted Red-black Key tree to store all the keys of each file. Then customers only need to store the root key to derive each leaf key for file encryption and decryption. They can add or delete keys with flexibility and elasticity.

The fourth example is dedicated server placement, where customers want to verify whether there are other VMs co-locating on the same cloud servers with their VMs. Zhang et al. [268] designed HomeAlone, which exploited the side-channel technique on the L2 cache to detect co-resident VMs. The customers can use their VMs to prime and probe some reserved space in the L2 cache to detect if there are activities from other VMs.

2.1.4 Virtualized System

The virtualized operating systems and hypervisors on cloud servers can introduce security threats to customers' data and computation.

2.1.4.1 Virtualized System Attacks

VM image attacks. Public clouds allow third parties (publishers) to create VM OS images, and upload them to the cloud app store for customers to use. When a customer chooses an image from the app store, the cloud provider initializes a VM from this image. The sharing of OS images can be exploited by malicious parties to steal private and sensitive information.

First, a VM image in the app store can contain malware and OS vulnerabilities. Balduzzi et al. [39] discovered that 98% of Windows images and 58% of Linux images in the Amazon EC2 cloud contain out-of-date software with critical vulnerabilities. Besides they also identified two Windows images infected by malware. Zhang et al. [261] identified several prevalent Common Vulnerabilities and Exposures (CVE) in public images in Amazon EC2 cloud.

Second, malicious publishers can deliberately install backdoors in the VM images, so they can login into customers' VMs originating from these images. There are three ways to achieve this: the first one is to leave the publishers' public keys in the images ([241, 57, 39]); the second one is to configure the SSH applications to enable password-based authentication and leave the publishers' passwords in the images ([39]). The third one is to extract the SSH host key pairs from their own VMs, as VMs originating from the same image sometimes generate the same SSH host key pairs [57].

Third, publishers may carelessly leave sensitive information in the VM images. This brings significant privacy threats to the publishers in different ways [57, 39]: (1)

Some images contain their publishers' cloud service API keys, which allow a malicious customer to hijack the publishers' accounts and create his own VMs at the expense of the publishers. (2) Some images have the publishers' SSH keys and credentials. A malicious customer can use these leaked keys to access the publishers' VMs. (3) Some images contain the shell histories, which have commands that may disclose passwords or credentials. (4) Browser's histories can be found in some published images. (5) Publishers' names, affiliations and photos can be found in some images. (6) Even if the publishers delete such information from the image filesystem, the customers are still able to recover these files using some tools.

Hypervisors-based attacks. A hypervisor is responsible for managing VMs and virtualizing hardware resources. Although hypervisors are typically much smaller than commodity OSes, common hypervisors still have fairly large code sizes. So it is impossible to eliminate security bugs or vulnerabilities inside the hypervisors. A malicious customer can exploit these vulnerabilities to take control of the hypervisors and conduct privilege escalation or Denial-of-Service attacks. Past work have studied the existing vulnerabilities in the hypervisors.

Perer-Botero et al. [173] characterized vulnerabilities of the Xen and KVM hypervisors from the Common Vulnerabilities and Exposures (CVE) database, and categorized them into eleven types: (1) virtual CPUs state saving/storing; (2) vCPUs scheduling on symmetric multiprocessor systems; (3) address translation in software MMU handled by the hypervisors; (4) incorrect handling of interrupt or timer mechanisms; (5) I/O and networking emulation; (6) I/O emulation in paravirtualized mode; (7) VM exits to the hypervisor from the VM; (8) Hypercalls; (9) VM management; (10) remote management software; (11) Hypervisor add-ons and extensions.

Pek et al. [171] introduced interrupt attacks to crash a virtualized system. This type of attacks exploits the vulnerabilities in the Direct Device Assignment technique, which assigns one physical I/O device exclusively to one VM and grants this VM full

control and direct access to the device. A malicious VM can re-configure the devices to generate arbitrary I/O interrupts, i.e., Message Signaled Interrupts (MSIs). The I/O Advanced Programmable Interrupt Controller (APIC) translates these MSIs to Non-Maskable Interrupts (NMI). Some NMIs can cause PCI System Errors and make the whole system halt.

Inside VM attacks. Since a virtual machine runs a full OS, all the malware that work in non-virtualized OSes can be invoked directly inside a VM without any modifications. This makes a VM as vulnerable as traditional systems.

2.1.4.2 Virtualized System Defenses

Managing VM images . It is important to protect the VM images in the cloud app store and eliminate potential vulnerabilities to their publishers and the retrievers. Wei et al. [241] designed Mirage, an image management system to address the security concerns of VM images using several approaches: (1) Access control: Mirage uses two types of access permissions, checkout and checkin, to carefully define who can revise or retrieve the images. (2) Image filters: Mirage uses repository-specific filters and user-specific filters to remove sensitive information from the publishers' original images. (3) Provenance tracking: Mirage tracks the derivation history and associated operations of the images for accountability. (4) Image maintenance: Mirage provides a set of maintenance services to the dormant images, e.g., malware detectors, license compliance managers and security patchers.

Eliminating Hypervisor's vulnerabilities . Past work have designed various mechanisms to enhance the security of hypervisors. The first direction is to design new secure hypervisors. McCune et al. [154] introduced TrustVisor, a tiny hypervisor to provide integrity for applications' critical data and codes. TrustVisor introduces the secure guest mode for the execution of applications, and uses x86 hardware virtualization support to enforce memory isolation between the hypervisor, the host OS and

applications. TrustVisor also creates a software micro-TPM instance for each application to perform integrity attestation. Vasudevan et al. [227] designed, implemented and verified an open-source eXtensible and Modular Hypervisor Framework (XMHF). XMHF consists of a XMHF core and small supporting libraries. A hypervisor application can extend the XMHF core and the basic hypervisor functionalities provided by this core to implement desired security functionalities.

The second direction is to protect the integrity of hypervisors. Wang et al. [240] designed HyperSafe, a lightweight approach to providing runtime integrity for Type-I hypervisors. HyperSafe uses the Write Protect (WP) bit to protect the hypervisor pages from being compromised by malicious programs. HyperSafe also sets up a target table containing all legitimate destinations for indirect control flow instructions to enforce the hypervisor program's control flow at runtime. Azab et al. [34] designed HyperSentry to measure the runtime integrity of a hypervisor. A remote client who wants to verify the hypervisor can use the Intelligent Platform Management Interface (IPMI) to trigger the server into the System Management Mode (SMM) and measure the hypervisor's code, data and CPU state. TPM-based protocols are exploited for secure communication.

The third direction is to reduce the hypervisors' privileges and functionalities. NoHype [129, 217] is designed to eliminate the hypervisor during the VM's runtime, thus reducing the attack surface from the hypervisor. Nohype achieves this by pre-allocating processor cores and memory for each VM during VM launch, assigning virtualized I/O devices directly to VMs to avoid needing the hypervisor to do I/O emulation, and modifying the guest OS to cache host system configuration for later use. Butt et al. [58] proposed self-service cloud computing to restrict the host VM's privileges. It splits the administrative privileges between a system-wide VM and per-client administrative VMs. The per-client administrative VMs are able to perform some privileged system tasks for their own VMs, while the system-wide VM cannot

inspect the code, data or computation of client VMs. Then security and privacy are preserved even if the host VM is compromised.

Introspecting VMs. A traditional method to defeat inside VM vulnerabilities is to place security tools inside the target VM. However, these security tools are located in the vulnerable system and hence are highly susceptible to the attacks. So researchers proposed to place the security tools in the hypervisor layer, making them introspect into the VM and monitor the activities inside the VMs. This technique is called Virtual Machine Introspection (VMI) [95, 79, 170].

One challenge for VM introspection is to narrow down the semantic gap between the views inside and outside the VMs. The first solution is to reconstruct the internal views of the guest VM in the hypervisor or secure monitoring VMs. Jiang et al. [127] cast the guest VM's semantic views of the OS into the hypervisor for monitoring in a non-intrusive manner. Fu et al. [93] designed VM-Space Traveler (VMST), which redirects the data related to the introspected VM's OS state to the monitoring VM for VM introspection.

The second solution is to reconstruct the suspect programs of the guest VM in the hypervisor or secure monitoring VMs. Srinivasan et al. [204] designed the process out-grafting technique, which relocates the suspect process to run side-by-side with the security tools. Dolan-Gavitt et al. [81] designed Virtuoso to automatically convert in-guest VM programs into out-of-guest VM programs that reproduce the same behaviors.

2.1.5 Shared Infrastructure

In a public cloud system, VMs or applications belonging to different customers can be placed on the same server, thus sharing the infrastructure. Although software isolation techniques carefully isolate memory spaces and CPU contexts between virtual machines and processes, the underlying hardware resources are still shared by different

customers' VMs and applications running on the same physical machine. This multi-tenant feature creates new vulnerabilities in cloud computing compared to private datacenters or personal computers. A malicious customer can exploit the shared infrastructure to attack the co-located VMs or applications. He first needs to conduct co-location attacks to achieve server co-location. Then he can conduct side/covert-channel attacks to steal the victim's data, row hammer attacks to tamper with the victim's data, or DoS attacks to compromise the victim's resource availability.

2.1.5.1 Shared Infrastructure Attacks

Co-location attacks. In an IaaS system, a VM co-location attack refers to the threat that the victim VM's location (i.e., its host server) can be identified by the attacker, and then the attacker can launch his VM on the same host server as the victim VM. Co-location attacks are the prerequisite for other shared infrastructure attacks.

Ristenpart et al. [182] first proposed network-based approaches to conduct co-location attacks in the Amazon EC2 cloud. An attacker can launch many VM instances in the same availability region as the victim VM, and use several ways to check if his VMs co-locate with the victim VM: (1) the attacker can use TCP SYN traceroute to identify the network traffic's first hop (which is the Dom0 in the host Xen server) of his VM and the victim VM. A matching Dom0 IP address between his VM and the victim VM indicates co-location. (2) The attacker can measure the network packet round-trip time between his VM and the victim VM. A smaller value indicates the two VMs share the same machine. (3) The attacker can check the internal IP addresses of his VM and the victim VM. Numerically close internal IP addresses indicate the two VMs are probably on the same server.

After Amazon EC2 eliminated these placement vulnerabilities, some other work proposed new methods to achieve co-location. Bates et al. [42] exploited the network

flow watermarking covert channel: the attacker injects network activities to each of his malicious VMs while measuring the victim VM's network performance. A network delay from the victim VM indicates that its performance is affected by the malicious VM, and thus co-location is confirmed. Herzberg et al. [115] proposed a co-location method using two steps. First the attacker needs to identify the victim VM's internal IP address. The attacker sets up a client machine to connect to the web service (e.g., downloading a file or a web page) of the victim via its public IP address. The attacker also uses a prober VM to send a large number of network packets to each possible internal address in the address block range. If the attacker's client machine observes that the victim's performance is affected by the prober VM, then the victim VM's internal address is the one that the prober VM is flooding. Second the attacker performs the Time To Live (TTL) scan using the internal addresses to measure the number of hops between his VM and the victim VM. A zero TTL indicates potential co-residence.

Xu et al. [248] used the DNS lookup mechanism to find the victim VM's internal IP address. Then they verified the co-location using two steps. The first step is pre-filtering unlikely pairs of co-located VMs by checking the /24 prefix in the internal IP addresses: if two VMs do not share the /24 prefix of internal IP addresses, they are not likely to be co-located. The second step is to use the bus locking covert channel to justify co-location: they construct a bus locking covert channel between each pair of VMs. If two VMs can communicate with each other via this covert channel, then they are located on the same physical machine. Varadarajan et al. [226] also exploited the bus locking covert channels to evaluate the financial costs of co-location in different public clouds.

Side/Covert-channel attacks. When the attacker and victim domains are on the same server, the attacker can exploit the shared CPU cache to extract crypto keys from the victim using cache side-channel attacks. The adversary has two basic techniques

to capture information on the cache. The first one is PRIME-PROBE. The adversary repeatedly accesses some cache lines and measures the time: a large time means a cache miss, indicating this cache set has been accessed by the victim. By checking the cache state altered by the victim, the adversary can obtain the cache accesses of the victim's program, and recover confidential data. Zhang et al. [269] first exploited the PRIME-PROBE technique to perform side-channel attacks on L1 cache in the Xen platform. Then PRIME-PROBE attacks on the last level cache were proposed to break different ciphers in different virtualized platforms [144, 120]. The second method is FLUSH-RELOAD. This requires the attacker and victim VMs to share some memory pages that contain critical instructions, which is enabled by the hypervisor's memory deduplication technique. The attacker VM flushes some cache lines in the cache using the *clflush* instruction then lets the victim execute, and then reloads these lines and measures the access time. A short access time at the reload indicates a cache hit so this line has been accessed by the victim VM. The attacker can obtain the victim VM's critical instruction traces and deduce the confidential data. Yarom et al. [255] demonstrated a FLUSH-RELOAD attack against the RSA cipher in different virtualized platforms. Irazoqui et al. [122] showed a similar attack against the AES cipher. Zhang et al. [270] showed the FLUSH-RELOAD technique could be used to attack different security-critical applications in the PaaS platform. Gruss et al. [107] designed cache template attacks using the FLUSH-RELOAD technique, which could automatically profile and extract information from arbitrary programs.

Another confidentiality attack is the covert-channel attack, where a malicious trojan inside the victim VM attempts to transmit secrets to another spy VM secretly. The trojan can generate certain characteristics using the shared hardware as the covert channels. Okamura and Oyama [165] proposed a cross-VM covert channel using CPU loads: the trojan can execute programs to transmit bit "1" and stay idle to transmit bit "0". The spy VM can measure its own CPU usage to infer the trojan's

CPU activities and thus decode the information. A more popular covert channel is established on the shared caches [247, 144]: the trojan can access some cache sets to transmit bit "1" and do nothing to transmit bit "0". The spy VM can check the states of these cache sets to infer the covert channel information. Wu et al. [244] proposed to use bus activities as the medium to transmit information: the trojan can lock the memory bus for bit "1" and release the bus for bit "0". Then the spy VM can extract the information by checking the bus locking states. Pessl et al. [174] proposed to use DRAM row buffer contention to establish covert channels. The trojan can occupy the bank row buffer for bit "1" and empty the bank row buffer for bit "0". By accessing the row buffer and testing if the access is a row hit or row miss, the spy VM is able to decode the sensitive information.

Row hammer attacks. Modern DRAM chips have large capacity and high density of memory cells. So a memory cell can suffer from disturbance errors due to electrical interference from the neighboring cells. Specifically, when an adversary rapidly and repeatedly accesses the DRAM with some specific patterns, certain data bits, in the memory region where the adversary has no access permission, can be flipped due to the electrical interactions. Such DRAM hardware vulnerability is called a row hammer attack. Xiao et al. [246] exploited this vulnerability to attack a para-virtualized platform from a guest VM. In their attacks, an adversary VM keeps accessing some selected data in the DRAM to flip critical bits in this VM's page table entry. By doing so the page table entry is changed to point to a forged page table without being noticed and checked by the hypervisor. The forged page table translates this VM's virtual page to a physical page that does not belong to this VM. So the attacker VM can steal or tamper with critical data from co-located VMs.

Denial-of-Service attacks. Hardware sharing can also be exploited to conduct host-based Denial-of-Service attacks. The adversary VM can generate contention on

different types of shared resources to degrade the victim VM's performance, or to increase its own performance.

The first resource in consideration is the CPU. Grunwald and Ghiasi [103] proposed to flush the shared processor pipeline to affect the victim's performance. They achieved this by executing denormalized floating point values to generate an underflow so the pipeline has to be flushed to handle the exceptional condition. Zhou et al. [274] designed a CPU resource attack where an attacker VM can exploit the boost mechanism in the Xen credit scheduler to increase its scheduling priority and obtain more CPU resource than paid for.

The second case is the memory system. Varadarajan et al. [224] proposed the resource-freeing attack, where an attacker can intentionally increase the victim VM's usage of one type of resource (e.g., network I/O) to force it to release other types of resources (e.g., CPU caches), so that a co-located VM controlled by the attacker may use more of the latter resources. Grunwald and Ghiasi [103] studied the effect of trace cache evictions on the victim's execution with Hyper-Threading enabled in an Intel Pentium 4 Xeon processor and showed that a malicious thread can slow down the victim's performance by a factor of 10-20. Woo and Lee [242] explored frequently flushing shared L2 caches on multi-core platforms to slow down a victim program. They studied saturation and locking of buses that connect L1/L2 caches and the main memory [242]. Moscibroda and Mutlu [160] studied contention attacks on the schedulers of memory controllers.

The third case is DoS on the I/O resources and networking. I/O resource contention can also bring performance degradation to the victim. For network resources, Bedi et al. [44] proposed a network-initiated DoS attack where an attacker VM causes contention in the Network Interface Controller to degrade the victim's performance. For disk resources, Yang et al. [253] proposed a method to reverse-engineer the I/O scheduling in the virtualization platform, which assists the attacker to design specific

Denial-of-Service attacks on the disk I/O resources. Chiang et al. [67] designed a more efficient adaptive attack, which identifies the I/O usage pattern of the victim, and synchronizes the attack phase with the victim. Huang and Lee [117] proposed cascading performance attacks, in which an attacker VM exhausts the I/O processing capabilities of the Xen Dom0, thus degrading the victim VM's performance. Similarly, Alarifi and Wolthusen [31] exploited VM migration to deplete Dom0's capability of I/O processing.

The last case in consideration is DoS for power. An attacker can deplete the host server's power consumption to make the victim's cloud services, or even the whole server break down. Xu et al. [249] designed a power attack to destroy a datacenter using the power over-subscription technique. The idea is to launch instances on the host server and run power-hungry programs to make the server reach power peak capacity. So the overall power consumption will exceed the quota, making the power unit fail and the server shut down.

2.1.5.2 Shared Infrastructure Defenses

Avoiding co-location. One method to eliminate vulnerabilities caused by shared infrastructure is to avoid or reduce the possibility of co-location between attacker and victim domains. This goal can be achieved via static VM launch or dynamic VM migration. During VM launch, some cloud providers offer dedicated VM options, in which the customers' VMs can use a dedicated server exclusively without sharing with other VMs. New VM placement policies were designed [36, 113] to reduce the probability that the attacker and victim VMs are sharing the same cloud server. At VM runtime, some work [272, 138, 159] proposed using Moving Target Defense to frequently migrate the VMs to add difficulty of VM co-location for attackers.

Reducing side-channel leakage. There are several directions for the cloud providers to defeat cache side-channel attacks in the clouds. The first one is to

prevent cache sharing by dividing the cache into different zones for different VMs or applications. This can be achieved by software or hardware methods. For software, some work [180, 200] exploited the page coloring technique to partition the cache. Kim et al. [131] designed STEALTHMEM to partition the LLC. It provides each VM a number of stealth pages which can not be replaced in the cache. For a software method using recent hardware performance feature, Liu et al. [141] exploited the Intel Cache Allocation Technology to protect the victim VM's critical data from being replaced by the attacker VM in the LLC. New hardware caches were designed [238, 83] to partition the caches by ways or sets to defeat side-channel attacks due to cache evictions.

The second idea is to use randomization in the system design so the attackers do not get any useful information. For software, system clock measurements were fuzzed to disrupt the attackers' observations [229, 139]. Zhang et al. [271] designed Duppel, which periodically performs cache cleansing during VM's executions and to add noise into the attacker's observations. For hardware, new caches were designed to randomize memory-to-cache mappings [238, 239, 143] and cache fetching [142].

Defeating row hammer attacks. Cross-VM row hammer attacks can be defeated via hardware or software solutions. For hardware, the Error-Correcting Code (ECC) memory can correct one single-bit error and detect 2-bit errors. This makes the row hammer attacks much harder [133]. For software, Brasser et al. [55] proposed two solutions: the first one is to extend the system bootloader to identify vulnerable memory pages. Row hammer exploitation tools [132, 105] are executed offline to discover the memory pages that could be tampered with by the row hammer attacks. Then the bootloader marks these vulnerable memory pages unavailable at boot-time so these pages will not be used at runtime. The second one is to extend the OS kernel to enforce a strong isolation of the physical memory of different system entities, e.g., user and kernel spaces. It ensures that memory between different entities is physically

separated by at least one row, so one entity cannot affect the memory of another entity. Irazoqui et al .[121] designed MASCAT, a static code analysis tool to scan application binaries and detect potential microarchitectural attacks, such as row hammer attacks. This tool leverages the signature-based detection technique to search the binary files for implicit characteristics that microarchitectural attacks usually exhibit in their design. In row hammer attacks, the attacker needs to continuously bypass the cache and access a fixed DRAM location. This is used as the signature of row hammer attacks.

Optimizing resources. To alleviate DoS attacks caused by resource contention, the cloud provider can optimize the resource usage between different domains and reduce performance interference. For memory contention, one method is to partition the hardware memory resources between different domains (e.g., Intel Cache Allocation Technology [8]). For I/O contention, the cloud server can monitor and manage the bandwidth of I/O traffic to prevent resource depletion. It can also use Direct Device Assignment [256] to physically eliminate I/O interference between each VM. These solutions are widely adopted by public cloud providers. For the power resource, Li et al. [137] proposed PAD to defeat datacenter power attacks under the over-subscription setting. PAD creates a virtual battery pool to enable load sharing and adjust power utilization of each rack. It can detect and shave power spikes to avoid power consumption failure.

2.1.6 Cloud Services

Malicious parties can abuse the cloud services to conduct large-scale attacks.

2.1.6.1 Cloud Service Attacks

Cloud abuse attacks . In an IaaS cloud system, the attackers can deploy a large number of VMs and use these VMs as botnets for cybercrime. Miao et al. [157]

identified several popular attacks in the clouds through analyzing the network flow logs. These attacks include Denial-of-Service attacks, email spam attacks which send email spam to multiple SMTP servers, brute-force password attacks which try to connect to the victim VM via SSH and guess the password, SQL injection attacks which send different SQL queries to exploit software vulnerabilities, and port scan attacks which scan for the active ports in the victim VM and figure out the applications or services it is running.

2.1.6.2 Cloud Service Defenses

Preventing abuse of cloud services. There are different ways to prevent cybercrime from happening in the cloud systems. The first way is to detect the misuse of the cloud services. Doelitzscher et al. [80] designed an anomaly detection system to analyze the cloud usage behaviors of cloud customers and identify malicious VMs in the IaaS clouds. It builds behavior models considering user-specific as well as cloud-wide behaviors, and uses the machine learning technique to discover anomalous VMs. The second way is to demotivate attackers by increasing the cost of cloud abuse attacks. Szefer and Lee [218] designed a method, which requires the customers to pay a small deposit using bitcoin before using the cloud services. If they use the service legally, the cloud provider will pay back the deposit to the customers. If malicious behaviors are detected, the cloud provider will keep the deposit as reimbursements. So the attackers lose the motivation due to the increased attack cost.

2.1.7 What is Covered in This Dissertation

A cloud system may face threats from different attack vectors, as introduced above in this section. In this dissertation, we attempt to study and address some security threats caused by the *shared infrastructure* and *virtualized system*. These attacks and solutions are unique to cloud computing. For the *shared infrastructure* threats, we consider the

Denial-of-Service attacks (availability) and side-channel attacks (confidentiality). The root cause of these two types of attacks is hardware sharing due to the multi-tenancy feature. For the *virtualized system*, we study the solutions of defeating inside VM attacks (integrity) using the virtualization technology.

We do not cover the following threats. Threats from the *service interface* (e.g., web attacks) and *networks* (e.g., DoS attacks) are not new to cloud computing: they were discovered long before the cloud era, and their mechanisms and solutions have been well studied. So we do not consider these two attack vectors in this dissertation. We trust the cloud providers in our threat model, so threats from malicious *cloud managers* are not covered in this dissertation. Also the threats of abusing *cloud service* for cybercrime are not in the scope of this dissertation.

We trust the cloud providers and assume that they can operate the cloud services correctly and deploy new security services for customers. So threats from malicious *cloud managers* are not covered in our threat model. This dissertation attempts to protect customers' VMs, not to defeat malicious customers. So the threats of *cloud service* abuse are not in the scope of this dissertation.

In Chapters 4, 5 and 6, we discuss attacks and defenses of availability, confidentiality and integrity properties.

Availability. In Chapter 4, we introduce novel DoS attacks caused by shared memory system. On modern cloud servers, physical CPUs are usually not shared by different VMs so CPU contention is not common. I/O contention is widely studied. However, hardware memory system is widely shared by VMs and the severity of memory contention is not well understood. Due to advances in computer hardware design, caches and DRAMs are larger and their management policies more sophisticated, and prior memory DoS attacks introduced in Section 2.1.5.1 may not work in modern cloud settings. We propose new memory DoS attack techniques on modern cloud servers. We evaluate the attacks in the lab and public cloud settings.

Past work offer solutions (Section 2.1.5.2) to achieve performance isolation and fairness. However, these solutions are designed for benign applications. They may fail to mitigate resource contention caused by intentional memory abuses. Besides, they only focus on one specific resource and cannot solve all the resource contention attacks. In Chapter 4, we present a novel and generalizable method to detect and mitigate all known DOS attacks on the memory resources.

Confidentiality. Side-channel defenses in prior work (Section 2.1.5.2) require significant modifications to the hardware, hypervisors or guest OSes, making them less practical for deployment in current cloud datacenters. Besides, some of the solutions are only effective for one specific type of attacks. In Chapter 5, we propose a new method to detect the existence of side-channel threats, and then mitigate them. Our method is based on the anomaly detection technique, and able to detect different types of side-channel attacks with high fidelity. Our solution is designed as a lightweight extension to the hypervisor, which does not require new hardware support or hypervisor/OS modifications.

Integrity. VM introspection methods introduced in Section 2.1.4.2 can be used for attack detection, malware analysis and forensics in cloud systems where the guest VMs' owners have full control of the privileged hypervisor. In public clouds, the cloud provider has the opportunity to provide this service to customers. We show how this can be done in Chapter 6. We integrate the VM introspection in a public cloud framework and show how the cloud provider can use this technique to enhance different aspects of security of customers' VMs on demand, without allowing cloud customers to control the hypervisor of the cloud server.

Vendors	Cloud Server Integrity Checking	Virtual Machine Integrity Checking	Inside VM Monitoring	Resource Monitoring	Multi-tenant Attack Detection
Academia	CloudVerifier [191] Excalibur [189]	CloudVerifier [191]			
OpenStack	Trusted Computing Pools [21]			Monasca [20]	
Amazon Web Services			Inspector [3]	CloudWatch [2]	
Microsoft Azure			Antimalware [15]	Application Insights[16]	
Google Compute Engine				Stackdriver [23]	
Our Work	Chapter 3	Chapter 3	Chapter 6	Chapter 3	Chapters 4, 5

Table 2.1: Comparisons between different cloud security platforms

2.2 Cloud Security Platforms

From the past work, we can see that security threats come from various entities and activities in the clouds, and the corresponding solutions are designed specifically for these threats. It is desirable for the cloud providers to use a unified cloud security platform to monitor and protect customers' VMs in comprehensive ways. In this section, we review some of these platforms from both the research literature and from commodity products, and make comparisons with our work in this dissertation. Table 2.1 shows such comparisons between different platforms from academia, open-source communities, and public cloud vendors. We classify their functions into different categories.

Cloud server integrity checking . This function is to verify the server configuration and platform integrity before launching VMs. Some research papers proposed methods to conduct such checking. Schiffman et al. [191] designed CloudVerifier, which checks the integrity of the software stack on the cloud server before allocating VMs on the server. Santos et al. [189] designed Excalibur, which checks each host server's configurations and allocates customers' VMs on the servers whose configurations match their specified policies. OpenStack designed the Trusted Computing Tool service [21], which exploits the Intel TXT technology [10] for VM allocation. It measures the integrity of the BIOS, the hypervisor and the host OS on each cloud server, and

guarantees that customers' VMs are placed on the cloud servers with verified and secure software stacks. Since our architecture is developed from OpenStack, it is also capable of checking server integrity.

Virtual machine integrity checking . This function is to check the OS image before launching the VM from it. CloudVerifier [191] proposed to measure the integrity of the VM image and launch the VM only when the image is verified. Our *CloudMonatt* architecture is also able to achieve the same function (Chapter 3).

Inside VM monitoring. This service is to detect vulnerabilities inside VMs at runtime. Amazon Web Services designed Inspector [3]. Amazon Inspector requires customers to install a security agent inside their VMs. Then the security agent monitors the activities inside the VM, including the network, file system, and process activities. Relevant data are collected, analyzed and compared to a set of security rules to check if there are any suspicious activities inside the VMs. Amazon Inspector is able to identify Common Vulnerabilities and Exposures (CVE), system mis-configurations, authentication vulnerabilities, insecure network protocols, etc. Microsoft Azure designed Antimalware [15]. It also requires customers to install security tools inside their VMs. Then these tools are able to identify and remove viruses, spyware and other malicious software, with configurable alerts when known malicious or unwanted software attempts to install itself or run inside the VMs. Our architecture is also able to achieve similar functions. Different from the above methods, our architecture exploits the VM introspection method to monitor activities in an unmodified VM and does not require customers to install or configure anything inside their VMs. (Chapter 6).

Resource monitoring . A lot of platforms provide customers with dynamic resource usage statistics and allow them to define their own metrics related to these resource statistics, and set alarms for resource consumptions. OpenStack designed Monasca [20], a Monitor-as-a-Service solution to provide VMs' resource consumption status

to customers. Monasca is able to monitor the VMs' CPU, memory and disk usage. Amazon Web Services designed CloudWatch [2], enabling customers to track VMs' resource usage. It supports monitoring of CPU usage, disk read/write throughput, inbound and outbound network traffic, etc. Microsoft Azure [16] also enables customers to monitor CPU percentage, inbound and outbound network traffic, disk read/write throughput. Google Compute Engine designed Stackdriver [23] to monitor the same resource consumption. Such resource consumption information allows customers to know how well their VMs or applications are performing, and helps them determine performance bottlenecks and fine-tune the performance. Similar to the above systems, our platform can use Hardware Performance Counters to measure VMs' resource consumption and provide such information to customers. Moreover, we are able to detect potential security vulnerabilities for customers through collecting and analyzing these performance values (Chapter 3). This is novel compared to both the existing security platforms and to the performance monitoring platforms.

Multi-tenant attack detection. One big feature of our architecture is that we can detect security vulnerabilities caused by the multi-tenancy feature. These include host-based availability vulnerabilities (Chapter 4), and side-channel information leakage (Chapter 5). These security services are missing in other platforms.

Our architectural framework is flexible in that it can support other security protection solutions as well. Future work can explore the integration of the vulnerability mitigation methods from Section 2.1, into our *CloudMonatt* platform.

2.3 Chapter Summary

In this chapter, we reviewed the related work on potential threats as well as new defenses in cloud computing, summarized in Figures 2.2 and 2.3. We described these attacks and defense solutions based on the attacker vectors, introduced in Section 1.3.

We also reviewed past work about cloud security platforms summarized in Table 2.1. In this dissertation, we design a novel architectural framework, *CloudMonatt* (Chapter 3), that enables the cloud providers to deploy different security protection mechanisms, and cloud customers to select different security properties for their virtual machines based on their demands. We will show how the cloud providers can provide some aspects of key security properties like availability (Chapter 4), confidentiality (Chapter 5) and integrity (Chapter 6) for customers, using this framework. We first begin with the presentation of this *CloudMonatt* architecture in the next chapter, followed by different security detection and mitigation cases in later chapters.

Chapter 3

VM Security Health Monitoring and Attestation

Cloud customers need guarantees regarding the security of their virtual machines (VMs), operating within an Infrastructure as a Service (IaaS) cloud system. However, monitoring a VM's security health is challenging: the security status of a VM is determined by different factors while customers have limited privileges to collect the corresponding measurements. Such a gap between the customers' security requirements and the measurements collected from the host server makes the monitoring task more difficult.

In this chapter, we present *CloudMonatt*, an architecture to monitor and attest virtual machine security health, with the ability to attest this to the customer in an unforgeable manner (most parts of this chapter have been published in [262, 263]). *CloudMonatt* monitors the security health of VMs over the VMs' lifecycle. It provides automatic remediation responses to failing security health indicated by negative attestation results. We provide concrete examples to show how to bridge the semantic gap between what the customer wants to know versus what can be measured in the

cloud. We show a concrete implementation of property-based attestation and a full prototype based on the OpenStack open source cloud software.

3.1 Background

In an IaaS cloud, a customer requests to launch a virtual machine in the cloud system. The cloud provider places the VM in a virtualized cloud server, and allocates a specified amount of physical resources (CPU, memory, disk, etc.) to this VM. During the VM's lifetime, the customer would like to know if his VM has good security health.

The security health of a VM indicates the likelihood that the VM satisfies the security properties the customer desires for his leased VM. For example, if the customer stores sensitive data in the cloud server's storage, a healthy VM enforces *confidentiality* protection of the data from other VMs, or from physical attackers. For another customer with time-critical service needs, a healthy VM means that resources that have been contracted for in the Service Level Agreement (SLA) are always *available* to the VM.

The security health of a VM depends on a variety of factors in the complicated cloud environment. First, a VM can get infected with malware or OS rootkits at runtime. Such *inside-VM* vulnerabilities can take complete control of the VM and significantly compromise its security state. Second, cloud management software usually have large code base sizes. This inevitably introduces bugs and gives adversaries opportunities to conduct privilege escalation attacks and gain root privilege [173]. Then the adversaries have full control of the whole server, as well as the capability of compromising any VM's security health on this server. Third, cloud systems usually adopt the "multi-tenancy" feature, where different customers share the same cloud server, as co-tenants or co-resident VMs. These VMs may belong to competitors, spies, or malicious attackers. Past work have shown that the "bad neighbor" VMs

are able to steal critical information through side-channel attacks [182, 269], thus compromising the VM's *confidentiality health*, or steal computing resources through Resource-Freeing attacks [224] or Memory DoS attacks [265], thus compromising the victim VM's *availability health*. We call the threats from the host OS and co-located VMs *outside-VM* vulnerabilities, which are hard for customers to defeat. Hence, a VM's security health depends on not only the activities inside the VM, but also the VM's interactions with its environment.

Monitoring the VMs' security health poses a series of challenges in a cloud system. First, the customer's limited privileges prevent him from collecting comprehensive security measurements to monitor his VM's health securely. He only has access to the VM, but not to the host server. For *inside-VM* vulnerabilities, once the VM's OS is compromised by the attacker, the customer may not get correct measurements. For *outside-VM* vulnerabilities, the customer cannot collect information about the co-resident VMs, hypervisor, etc. Second, the customer's desired security requirements are expressed in terms of a VM, but the security measurements usually involve the physical server, the hypervisor and other entities related to this VM. This creates a semantic gap between what the customers want to monitor and the type of measurements that can be collected. Third, the VMs go through different lifecycle stages and may migrate to different host servers. A seamless monitoring mechanism throughout the VMs' lifetime is therefore highly desirable. Fourth, there are numerous entities between the customers and the point of VM operations. It is important to collect, filter and process the monitoring information securely to attest, i.e., pass on to the customer in an unforgeable way, only the requested information.

In this chapter, we design a flexible architecture called *CloudMonatt*, to monitor and attest the security health of customers' VMs within a cloud system. *CloudMonatt* is built upon the *property-based* attestation model (defined in detail in Section 3.1.2.1), and provides several novel features. First, it provides a framework for monitoring

different aspects of security health. Second, it shows how to interpret and map actual measurements collected to security properties that can be understood by the customer. These bridge the semantic gap between requested VM properties and the platform measurements for security health. Third, attestations can be done at runtime and for VM migrations, not just at boot up and VM launch time. Fourth, *CloudMonatt* provides remediation response strategies based on the monitored results. To demonstrate the practicality of *CloudMonatt*, we implement a full prototype on the OpenStack open source cloud platform.

Given that *CloudMonatt* is designed to monitor and report virtual machines' security health, it is important and necessary to systematically check that it works correctly as expected, with no vulnerabilities that could be exploited by attackers to subvert its security scheme. As such, we conduct a systematical security verification of *CloudMonatt*, proving that its VM health monitoring service is trustworthy and adversaries have no chance to falsify the attestation report for the customer or the cloud provider. Verifying a distributed cloud system like *CloudMonatt* is challenging as it involves a variety of cloud servers. This requires us to consider the external communication protocols between servers, as well as the internal activities inside each server. To solve this, we break down the whole verification task into two parts, the external network verification and the internal server verification. We model each component as state machines, propose the security invariants for checking, and use a protocol checking tool, *ProVerif* [51] to model the system protocols/operations and verify the invariants. The verification results not only raise our confidence in our *CloudMonatt* design, but also provide suggestions for further strengthening its reliability and trustworthiness.

3.1.1 Security on Demand Framework

CloudMonatt is an outcome of the Security on Demand (SoD) cloud framework project [124], which provides on-demand, customized VM security for the cloud customers. Just as customers request computing resources in the Service Level Agreement (SLA), they can also request different types and levels of security. Basically this framework achieves three primary design goals: (1) enable customers to request customized security for their VMs; (2) enable different secure server architectures to service these customized security requests; and (3) provide persistent security for each VM throughout its life-time in the cloud. Below we detail these three goals.

Customized security requests. In the SoD framework, customers are allowed to choose among a range of security options that are most suitable for their VMs, where each option is based on a specific underlying threat model. These security options include two parts. The first one is security levels, which define different entities that customers trust within the cloud. For example, customers can select to protect their VMs from untrusted co-located VMs, hypervisor, hardware, cloud manager, guest OS or applications. The second part is security properties. Customers can specify the security properties they desire for their VMs, such as different aspects of confidentiality, integrity, availability, etc.

Secure server architectures. The SoD framework leverages existing secure server architectures to service these highly customized VM security requests. Numerous secure hardware-software architectures have been proposed [217, 240, 61, 155, 186], which can provide security protections under different threat models. The cloud provider can install different types of secure servers in a datacenter, each capable of providing different types of security. The cloud provider collects, monitors and maintains these servers' current security capabilities. Given a customer's request for certain types of security protection in his SLA, the cloud provider determines the most

appropriate type of secure server for the VM. Once the set of servers is determined, the cloud provider deploys the VM on one of the selected servers.

Persistent security protection. The SoD framework provides lifetime VM security using live VM migration. The security enforcement process for a given VM can be disturbed by two events: (1) change in the security level requested by the cloud customer; (2) an attack on the server on which the VM is running. When either of these events occur, the VM is remapped to a new server capable of satisfying the new security requirements, and it is live-migrated to the newly selected server. This process is carried out throughout the lifetime of the VM to ensure persistent security. An additional trigger and strategy for migration is based on the "Moving Target Defense", where VMs are moved to other servers every now and then so that attackers cannot try to co-locate their malicious VMs on the same physical server.

Given a SoD environment, where a cloud customer can request and pay for extra security services, how does the cloud customer check that he is getting these services? This dissertation proposes *CloudMonatt*, an architecture, to answer the question in terms of the security health of a VM the customer can request at any time. Similar to SoD, *CloudMonatt* also allows customers to specify the security properties they desire for their VMs. Then the cloud provider launches the VMs on the cloud servers that are capable of monitoring the specified properties. Different from SoD, *CloudMonatt* keeps attesting the VMs' security health thoughout their life-time. If the security status of the monitored VM violates the customer's specification, *CloudMonatt* provides more remediation solutions, in addition to live migration in the SoD framework (Section 3.2.5).

3.1.2 Related Work

3.1.2.1 Remote Attestation

Remote attestation is defined as *"the procedure of making claims about the security conditions of a targeted system based on the evidence supplied by that system"* [72]. It often involves three entities: an *attester* is the targeted system which provides the evidence; a *verifier* is an entity which requests an attestation report for a given attester; an *appraiser* is an entity which makes decisions by evaluating the security conditions based on the attester's evidence. We review two types of attestations.

Binary attestation. Proposed by the Trusted Computing Group (TCG) [99], binary attestation was a breakthrough development which helped enable attestation of platform integrity of a remote server. The attester calculates binary hash values of the platform configurations, and sends them to the verifier. The verifier, who typically also plays the role of the appraiser, compares these values with reference or "good" configurations, and determines whether the state of the attester is acceptable.

Many systems enabled with remote attestations have been designed, based on the Trusted Computing Group's binary attestation [99, 100]. Sailer et al. [186] proposed the Integrity Measurement Architecture (IMA), to measure the integrity of executables from BIOS to application level. Jaeger et al. [123] extended IMA to Policy Reduced Integrity Measurement Architecture (PRIMA), which can measure the Mandatory Access Control (MAC) Policy defined for controlling information flows across user processes. Shi et al. [199] proposed the architecture of Binding Instructions aNd Data (BIND) to realize fine-grained attestation on the integrity of code in distributed systems. Seshadri et al. [195] proposed the Pioneer system, which can attest the integrity of code executions on legacy computing systems. Garfinkel et al. [94] designed Terra, an architecture with a trusted virtual machine monitor (TVMM) to provide a secure computing environment by isolating critical application code in different VMs.

Binary attestation has certain shortcomings [184, 163]. First, binary measurements sent to the verifier provide configuration and implementation details of the attester, which is a privacy issue and may lead to fingerprinting attacks. Second, the verifier (who is also the appraiser) must be aware of the correct configurations of the target platform. Third, the target platforms may get updated leading to a change in configurations, and thus requiring the verifier to be notified about it each time.

Property-based attestation. To address the above shortcomings, property-based attestation was proposed [184], which attests security properties, functions and behaviors of systems. In property-based attestation, the verifier and the appraiser are separate entities. The appraiser is a trusted third party, who is trusted by the attester and the verifier. The appraiser has full knowledge of the attester. Its job is to transform the attester's measurements into properties and vice versa, and determine if the attester satisfies a set of given properties. A common solution for realizing an appraiser's interpretation mechanism is delegation-based attestation [184, 63]. In this approach, the appraiser can issue a *property certificate*, proving that a given configuration fulfills a specific property demanded by the verifier. Other approaches like proxy-based [177] are also proposed and implemented.

A variety of methods deriving from *property-based attestation* are explored, to attest different security properties. Haldar et al. [111] proposed semantic attestation, which monitors programs' high-level dynamic behaviors and properties. Alam et al. [30] proposed model-based behavioral attestation to attest the behaviors of security policies associated with the platforms. Sirer et al. [201] proposed logical attestation, which translates the programs' high-level attributable properties to logical expressions for verification.

Compared with binary attestation, the advantages of property-based attestation are as follows: properties do not reveal the configuration and implementation details, and thus do not violate the privacy of the attester; properties do not change as often

as the target platform's configurations; properties are easier to understand and express. However, the specification and interpretation of properties to be attested remain as challenging, open problems [163]. They make it very difficult for computer architects to convert the concept of *property-based attestation* into real architectures.

3.1.2.2 Attestation in Cloud Computing

Attestation of VM security health in the cloud environment is more complex. In IaaS, the *verifier* is the customer who launches a VM in the cloud and the *attester* is the VM. The health of the target VM depends on not only the applications and OS within the VM, but also its interactions with the host environment. In addition, since a VM experiences different activities during its lifecycle, it is important to consider attestation throughout the VM's life.

Direct attestation. Direct attestation allows the customers to talk to the VMs directly. One example is the virtual Trusted Platform Module (vTPM) [45, 88, 185, 190, 230]. Since a physical TPM cannot be directly used by the VMs within virtualized environments, vTPMs are designed to provide the same usage model and services to the VMs. Then attestation can be carried out directly between the customers and virtual machines by the vTPM instances. These instances can be realized by implementing TPM emulators in the hypervisor or host OS, by modifying the hardware TPM to enable TPM virtualization [45], or by combining both software and hardware TPMs [88]. To overcome binary attestation's shortcomings, Sadeghi et al. [185] proposed virtual property-based attestation, in which the vTPM instances are assigned the tasks of security property management and interpretations.

The virtual TPM solution raises some problems for VM monitoring: it cannot attest the security conditions of the VM's environments. Furthermore, the monitoring tool resides in the guest OS, so it needs modification of the guest OS, and commodity OSes are also highly susceptible to attacks.

Centralized attestation. To overcome the above problems, the concept of *centralized attestation* is introduced in the cloud system to manage the attestation procedure. Schiffman et al. [191] implemented a centralized "cloud verifier" that can provide the integrity attestations for customers' VM applications. Customers issue the authorization for the VM to access applications only when the integrity attestation passes. Santos et al. [189] designed a centralized monitor to check the platform's configurations and map them to security attributes. This enables customers' VMs to be allocated on the platforms with specified attributes. Then Attribute-Based Encryption is exploited to seal and unseal data between customers and cloud servers to ensure they are not compromised. However, the above work are still based on *binary attestation* for platform integrity and configuration checking, and do not consider other security properties like confidentiality or availability, nor the VMs' interactions (intended or unintended) with the outside-VM environment.

3.1.2.3 Attestation Protocols

Remote attestation needs the support of cryptographic protocols. For *binary attestation*, the most basic protocol is the standard signature scheme which was originally adopted by TCG (TPM specification v1.2) [99, 186]. Manufacturers burn a private key into the micro-processor chip, and then the TPM generates the attestation key-pairs, signs the public key, and sends it to the privacy certificate authority for a certificate. The TPM specification v1.2 also includes the Direct Anonymous Attestation functionality [56], which can preserve the anonymity of the attested platforms from the verifier using the group signature scheme. Then TCG released TPM specification v2.0 [101, 64], which included multiple cryptographic functionalities and flexibly selective cryptographic algorithms, e.g., anonymous signatures, pseudonym signatures, and conventional signatures. Stumpf et al. [210] enhanced the TCG-based attestation

protocol by integrating Diffie-Hellman key exchange protocol to defeat masquerading attacks.

For *property-based attestation*, Chen et al. [63] designed a provable and efficient protocol with a delegation solution. This protocol holds the security features of unforgeability (i.e., the signature can only be produced by the valid TPM) and unlinkability (i.e., the verifier cannot deduce the specific configuration of the platform). It also supports the revocation of invalid certifications. Chen et al. [65] proposed another property-based attestation protocol based on the ring signature scheme. This protocol can preserve the platform's privacy and avoid the involvement of a trusted third party to certify properties, which will be done by the attested platform. Different from past work, the protocol in *CloudMonatt* involves four entities in a cloud system. We design new protocols to provide unforgeability for attestation reports, and anonymity for attested platforms. Anonymity will be discussed in Section 3.2.4.

3.2 CloudMonatt Architecture

3.2.1 Design Goals of the Architecture

The design goals of the *CloudMonatt* architecture are:

1. To provide a flexible distributed cloud architecture that can detect and monitor the security health of the customers' VM in the cloud, e.g., by detecting its vulnerabilities, the vulnerabilities of the platform it is running on, or the vulnerabilities due to co-resident VMs;
2. To provide a secure protocol to request and receive security property monitoring measurements from the cloud's secure servers, and produce an unforgeable attestation report; and

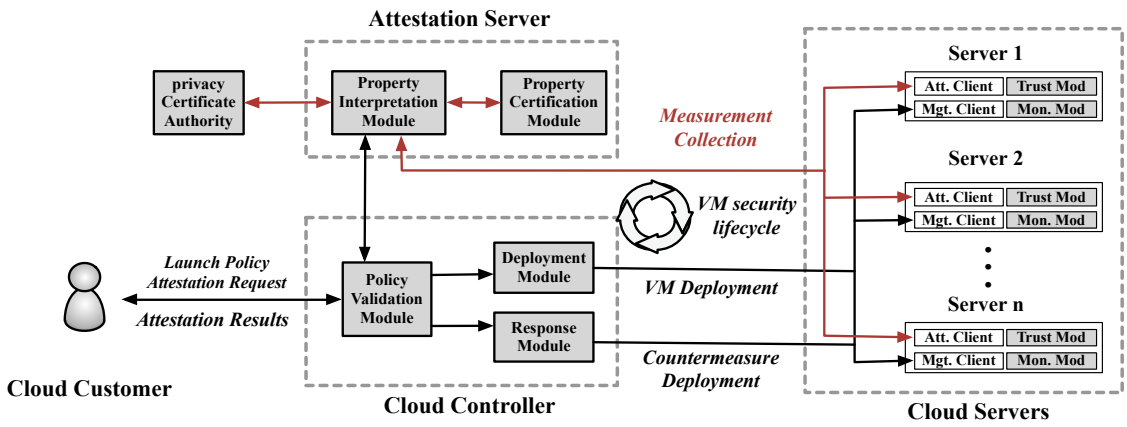


Figure 3.1: Architectural overview of *CloudMonatt*

3. To interpret security health measurements, determine if a requested security property is held for the VM, and enable different remediation responses when the VM's security health is appraised as inadequate.

In this section, we describe the main architecture for achieving goals (1) and (2), which are independent of the specific security properties a server can implement within the *CloudMonatt* architecture. Section 3.2.2 describes the main architectural components. Section 3.2.3 describes the threat model, referring to these components. Section 3.2.4 describes the monitoring and attestation protocols. Section 3.2.5 discusses the attestation and response actions during VM lifecycle. Goal (3) depends on the specific security property being monitored, and Section 3.3 gives several concrete examples of property interpretations.

3.2.2 Architecture Overview

Figure 3.1 shows an overview of the *CloudMonatt* architecture. This includes four entities: 1) Cloud Customer, 2) Cloud Controller, 3) Attestation Server and 4) Cloud Server.

Cloud Customer: The customer is the initiator and end-verifier in the system. He places a request for leasing VMs with specific resource requirements and security

Request API	Description
startup_attest_current(**Vid**, **P**, **N**)	Invoke an attestation of VM **Vid** for security property **P**, before launching the VM
runtime_attest_current(**Vid**, **P**, **N**)	Invoke an immediate attestation of VM **Vid** for security property **P**
runtime_attest_periodic(**Vid**, **P**, **freq**, **N**))	Invoke a periodic attestation of VM **Vid** for security property **P** at the frequency of **freq** or at random intervals
stop_attest_periodic(**Vid**, **P**, **N**))	Stop a periodic attestation of VM **Vid** for security property **P**

Table 3.1: Types of monitoring and attestation requests. (nonces N are added for freshness for each request)

requests to the Cloud Controller. He can issue any number of security attestation requests during his VM's lifetime. Table 3.1 shows the attestation and monitoring APIs provided to the customers. *CloudMonatt* allows customers to invoke the monitoring and attestation requests at any time during the VM's lifecycle. It also gives the customers two modes of operation: one-time attestation and periodic attestation.

- *One-time attestation:* the customer can request the attestation at any time. Then the Attestation Server performs the required attestation and sends back the results.
- *Periodic attestation:* the customer can ask for periodic attestations with specified constant or random frequency (this can prevent the attacker from reverse-engineering the attestation scheme and scheduling the attack phases to avoid detection). The cloud server supplies the measurements, and the Attestation Server accumulates and interprets the measurements periodically. The customer receives fresh results periodically and can stop the process at any time.

Cloud Controller: The Cloud Controller acts as the cloud manager, responsible for taking VM requests and servicing them for each customer. The `Policy Validation Module` in the Controller selects qualified servers for customers' requested VMs. These servers need to both satisfy the VMs' demanded physical resources, as well as support the requested security properties and their property monitoring services. The `Deployment Module` allocates each VM on the selected server.

During the VMs' lifecycle, the customers may request the Cloud Controller to monitor the security properties associated with their VMs. The Cloud Controller will

entrust the Attestation Server to collect the monitored security measurements from the correct VMs, and send a report back to it. It then sends the results back to the customers to keep them informed of their VMs' security health. When these results reveal potential vulnerabilities for the VMs, the `Response Module` in the Controller carries out appropriate remediation responses.

Attestation Server: The Attestation Server acts as the attestation requester and appraiser, and consists of two essential modules. 1) The `Property Interpretation Module` is responsible for validating measurements, interpreting properties and making attestation decisions. It needs a certificate from a `privacy Certificate Authority` (pCA) to authenticate cloud servers. The `privacy Certificate Authority` may be a separate trusted server already used by the cloud provider for standard certification of public-key certificates that bind a public key to a given machine. 2) The `Property Certification Module` is responsible for issuing an attestation certificate for the properties monitored. It stores mappings of security property **P** to measurements **M**. This gives a list of measurements **M** that can indicate the security health with respect to the specified property **P**. There can be different Attestation Servers for different clusters of cloud servers, enabling scalability of the *CloudMonatt* architecture.

We introduce the Attestation Server for security monitoring/attestation while the Cloud Controller is responsible for management. This job split achieves better scalability, since different attestation servers can be added to handle different clusters of cloud servers. It consolidates property interpretation in the attestation servers, rather than replicating this in each cloud server, or burdening the Cloud Controller. This also achieves better "separation of duties" security, since the Cloud Controller need only focus on cloud management while the Attestation Server focuses on security. It also improves performance by preventing a bottleneck at the Cloud Controller if it had to handle management as well as myriad attestation requests and security property interpretations.

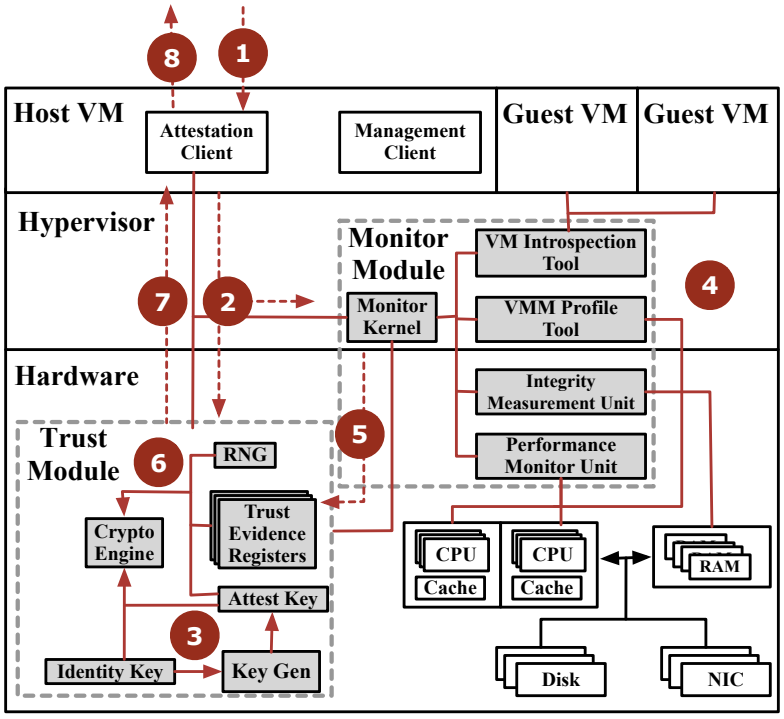


Figure 3.2: Server architectures enabling security monitoring include new trusted hardware and software features (shown in grey) in a Trust Module and a Monitor Module.

Cloud Server: The Cloud Server is the computer that runs the virtual machine in question. It is the attester in the system. It provides different measurements for different security properties. Figure 3.2 shows the structure of a cloud server with a Type-I hypervisor (e.g., Xen [41]). This has the hypervisor sitting on bare metal, and a privileged VM called the host VM (or Dom0) running over the hypervisor. Not all the cloud servers in the cloud provider's data center have to be trusted (almost all existing ones are not), only those servers on which security monitoring is necessary need to be secure. To support *CloudMonatt*'s goals, a cloud server must include a `Monitor Module` and a `Trust Module`.

The `Monitor Module` contains different types of monitors to provide comprehensive and rich security measurements. These monitors can be software modules or existing hardware mechanisms like performance counters or Intel TXT technology.

For example, the Performance Monitor Unit (present ubiquitously in Intel x86 [9] and ARM processors [6]) has numerous Hardware Performance Counters to collect runtime measurements of the VMs' activities. An Integrity Measurement Unit (which could use the Dynamic Root of Trust for Measurement in Intel TXT technology [10]) can be used to measure accumulated hashes of the system's code and static data configuration. In the hypervisor, a Virtual Machine Introspection tool (e.g., LibVMI [12]) can be used to collect the information inside the specified VM, and the VMM profile tool (e.g., xentrace [26]) can be used to collect dynamic information about each VM's activities.

We define a new hardware `Trust Module` in Figure 3.2. This `Trust Module` is responsible for server authentication using the `Identity Key`, crypto operations using the `Crypto Engine`, `Key Generation` and `Random Number Generation` (RNG) blocks, and secure measurement storage using the `Trust Evidence Registers`. By using new hardware registers to store the security health measurements (trust evidence), we do not need to include the main DRAM memory in our Trusted Computing Base, although trusted RAM can also be used instead of `Trust Evidence Registers` in the Trust Module.

Figure 3.2 also shows the functional steps taken by the `Monitor Module` and the `Trust Module`. The Cloud Server includes an `Attestation Client` in the host VM that ① takes requests from the Attestation Server to collect a set of measurements. It invokes the `Monitor Module` ② to collect the measurements and the `Trust Module` ③ to generate a new attestation key for this attestation session. This new attestation key is signed by the `Trust Module`'s private identity key. The required measurements of suspicious events or evidence of trustworthy operation are ④ collected from the `Monitor Module` and ⑤ stored into new `Trust Evidence Registers`. These `Trust Evidence Registers` are analogous to the performance counters used for evaluating the system's performance, except that they measure aspects of the system's security.

The `Trust Module` then ⑥ invokes its `Crypto Engine` to sign these measurements and ⑦ forwards the data to the `Attestation Client` which ⑧ sends it to the Attestation Server. The `Trust Module` contains a `Key Generator` and a `Random Number Generator` for generating keys and nonces.

3.2.3 Threat Model

The threat model is that of hostile VMs running in the cloud on the same cloud server, or hostile applications or services running inside a VM, that try to breach the confidentiality or integrity of a victim VM's data or code. They may also try to breach its availability, in spite of the cloud provider having allocated the VM its requested resources. The cloud provider is assumed to be trusted (with its reputation at stake), but may have vulnerabilities in the system. We assume that the Cloud Controller and the Attestation Server are trusted — they are correctly implemented, with secure bootup and are protected during runtime. However the Cloud Servers need not be trusted, except for the `Trust Module` and `Monitor Module` in each server. Note that the Cloud Controller and Attestation Server can be redundancy protected for reliability and security, and are only a small percent of all the servers in the cloud's data center. Also, not all the thousands of cloud servers need to be *CloudMonatt*-secure servers.

We focus on two types of adversaries: (1) An adversary, who tries to exploit vulnerabilities in the customers' VMs, either from inside the VM, or from another malicious VM co-resident on the same server. (2) An active adversary who has full control of the network between different servers, as in the standard Dolev-Yao threat model [82]. The adversary is able to eavesdrop as well as falsify the attestation messages, trying to make the customer receive a forged attestation report without detecting anything suspicious. With regard to this second adversary, *CloudMonatt* needs secure monitoring and attestation protocols which we define next.

3.2.4 Monitoring and Attestation Protocols

In a distributed architecture where communication is over untrusted networks, the protocols are an essential part of the security architecture: they establish trust between the customer and the cloud provider, and between different computers in the cloud system. In *CloudMonatt*, an attestation protocol must be unforgeable in spite of the network attacker and the other attackers in the untrusted servers. This requires secure communications among the four entities in Figure 3.1, and unforgeable signatures of the measurements and the attestation report from the place of collection (in the Cloud Server) through the Attestation Server, Cloud Controller and finally to the customer. We first describe the main attestation protocol. Details of the cryptographic keys involved, the secure communications and storage will be clarified later.

Figure 3.3 shows the attestation protocol in *CloudMonatt*.

1. The customer initially sends to the Cloud Controller the attestation requests including the VM identifier **Vid**, the desired security property **P** and a nonce $\mathbf{N}_1$. The nonce is an arbitrary number used only once in this session. It is used to prevent replay attacks over the channel between the customer and the Cloud Controller.
2. The Cloud Controller knows the current mapping of all VMs to their assigned cloud servers. It discovers the host server of VM **Vid**, **I**, and sends to the Attestation Server the request, which includes **Vid**, **I**, **P** and another nonce $\mathbf{N}_2$.
3. Given the property **P**, the Attestation Server identifies the required monitoring measurements **rM**. Then it sends **Vid**, **rM** and its nonce $\mathbf{N}_3$ to the cloud server **I** where the VM is running.
4. In the Cloud Server, the `Monitor Module` collects the required measurements **M** and stores them into the `Trust Evidence Registers`. Then the `Trust Module` calculates the quote $\mathbf{Q}_3$ as the hash value of (**Vid**, **rM**, **M** and nonce $\mathbf{N}_3$) (We

borrow the term "Quote" from TPM notation, to represent a cumulative hash measurement), and sends to the Attestation Server a signature of **Vid**, **rM**, **M**, $\mathbf{N}_3$ and $\mathbf{Q}_3$.

5. The Attestation Server verifies the signature and checks the integrity of the measurements by calculating the hash value and comparing it with the quote $\mathbf{Q}_3$. Then it interprets the measurements **M** and property **P** and generates the attestation report **R**. The Attestation Server calculates the quote $\mathbf{Q}_2$ as the hash value of (**Vid**, **I**, **P**, **R** and $\mathbf{N}_2$), and sends to the Cloud Controller a signature of **Vid**, **I**, **P**, **R**, $\mathbf{N}_2$ and $\mathbf{Q}_2$.
6. The Cloud Controller verifies the signature and checks the integrity of the report **R** via the hash value $\mathbf{Q}_2$. Then it generates the quote $\mathbf{Q}_1$ by hashing **Vid**, **P**, **R** and $\mathbf{N}_1$, signs these values and sends the signature to the customer.
7. The customer verifies the signature and hash value. If they are correct, the customer gets the correct report **R**.

Secure Storage and Communications. For secure storage, the `Trust Module` provides `Trust Evidence Registers` for saving attestation measurements, which are only accessible to the `Trust Module` and `Monitor Module`. Accesses to the databases in the Cloud Controller and the Attestation Server are also protected to ensure data confidentiality and integrity.

For secure communications over networks, the *CloudMonatt* architecture expects the customer, Cloud Controller, Attestation Server and secure Cloud Servers to implement the SSL protocol. Our contribution is defining the *contents* of the SSL messages, and the keys and signatures required for unforgeable attestation reports and Cloud Server anonymity.

Key Management. We now describe the keys used in Figure 3.3. The Cloud Controller, Attestation Server and each secure Cloud Server must have one long-term public-private key-pair that uniquely identifies it within the cloud system. This

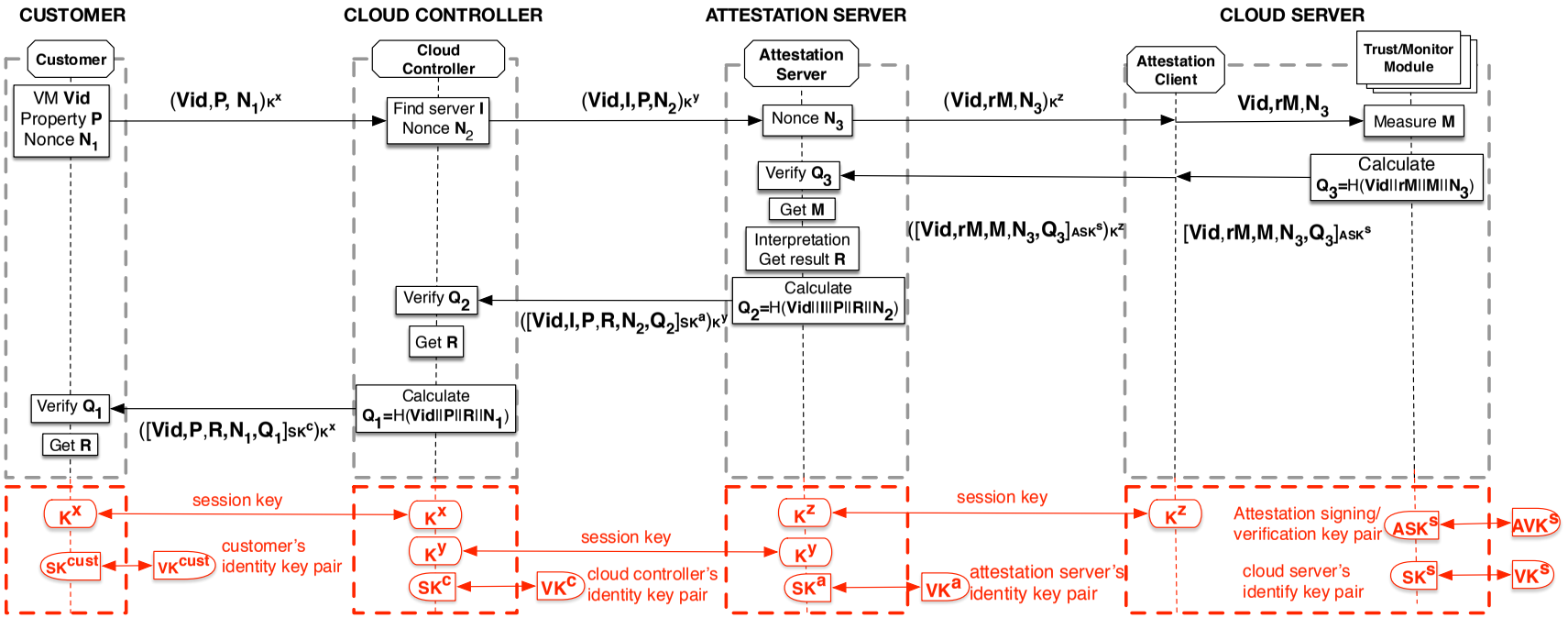


Figure 3.3: Attestation Protocol and Key Management in *CloudMonatt*. We use the notation $[M]_K$ for a private key operation with key K, $\{M\}_K$ for a public key operation with key K, and $(M)_K$ for a symmetric key operation with symmetric key K. N_i represents a Nonce between two communication parties.

is minimally what is required for SSL support, and is already present in all cloud servers. Hence, each secure cloud server owns a pair of public-private identity keys, $\{\mathbf{VK}^s, \mathbf{SK}^s\}$. The private key, $\mathbf{SK}^s$, can be burned into the `Trust Module` when manufactured, or more preferably, securely inserted into a non-volatile and tamper-proof register in the `Trust Module` when the server is first deployed in the cloud data center. This private identity key is never released outside of the `Trust Module`. The public key, $\mathbf{VK}^s$, can be used to authenticate the cloud server. A cloud server mainly uses this identity key-pair to generate a temporary key pair for each attestation request.

A new session-specific key-pair, $\{\mathbf{AVK}^s, \mathbf{ASK}^s\}$, is created by the `Trust Module` whenever an attestation report is needed, so as not to reveal the location of a VM. (An attacker may try to find the server which hosts the victim VM, then he can try to co-locate his VM on the same server. We do not want our attestation protocol to help an attacker do this [182]). The public attestation key $\mathbf{AVK}^s$ is signed by the Cloud Server's $\mathbf{SK}^s$ and sent to the pCA for certification. The pCA verifies the signature via $\mathbf{VK}^s$ and issues the certificate for $\mathbf{AVK}^s$ for that server. This certificate enables the Attestation Server to authenticate the Cloud Server "anonymously" for this attestation.

For secure communications between the servers, SSL first authenticates sender and receiver using their public-private key-pairs, then generates symmetric session keys for encrypting the messages passed between each pair of servers. Hence, Figure 3.3 shows the communications between the customer and the Controller protected with a symmetric session key $\mathbf{K}^x$, between the Controller and the Attestation Server with a symmetric session key $\mathbf{K}^y$, and between the Attestation Server and Cloud Server with symmetric session key $\mathbf{K}^z$.

3.2.5 VM Lifecycle and Attestation Responses

Attestations can be performed at all stages of a VM's lifecycle, during VM launch, during its runtime, before and after any VM migrations and on VM termination.

VM Startup and Responses. Startup attestation can ensure that the VM is correctly initialized and launched. This is an attestation of the integrity of the platform and the VM image. If the platform's integrity is compromised, *CloudMonatt* will select another qualified server for hosting this VM. If the VM image is compromised, then the VM launch request will be rejected. If both the VM and platform pass the integrity checks, the VM will be successfully launched on this server.

VM Runtime and Responses. *CloudMonatt* provides a flexible protocol for monitoring the VM's runtime activities, as described in Section 3.2.4 and Table 3.1. Customers can issue a one-time attestation request, or a periodic attestation request, during the VM's execution to monitor its health. *CloudMonatt* provides a set of responses to a VM that is compromised, or under attack. Currently we implement:

- *Termination*: the cloud controller can shut down the VM to protect it from attacks.
- *Suspension*: the controller can temporarily suspend the VM when it detects the platform's security health may be questionable. Meanwhile, it can initiate further checking and also continue to attest the platform. If the attestation results show the cloud server has returned to the desired security health, the controller can resume the VM from the saved state.
- *Migration*: when the security health of the current server is questionable or the server has been compromised, the controller tries to find another secure cloud server that can satisfy the VM's security property requirements. If a suitable server is found, the controller migrates the VM to that server. Otherwise, this VM is terminated for security reasons.

VM Migration and Responses. A VM may need to migrate to other servers due to resource optimization, or for security reasons. *CloudMonatt* finds a qualified server that supports this VM's resource demands and security and attestation needs. The VM may need to be shut down if no server is found.

In the next section, we elaborate on what security health monitoring means for different security properties like confidentiality and availability, in addition to integrity. In past work, integrity has been the primary, if not the only security property measured (and usually only on bootup). We give concrete examples to illustrate the definition and monitoring of a broader range of security properties, including example attacks, to illustrate potential security breaches in the cloud.

3.3 Case Studies

We define the *Security Health* of a virtual machine as an indication of the likelihood of its security being affected by the actions of hostile VMs co-resident on the same cloud server, or hostile applications, services or malware within the VM itself. Different indicators of different aspects of security health can be monitored. In our context, these different aspects of security are the security properties requested by the customer. These security properties can be monitored by the various monitors in the server's `Monitor Module` or collected by the `Trust Evidence Registers` in the server's `Trust Module`. The *CloudMonatt* architecture is flexible and allows the integration of an arbitrary number of security properties and monitoring mechanisms, including logging, auditing and provenance mechanisms.

To monitor and attest a security property, three requirements must be satisfied: (1) the Attestation Server can translate the security property, requested for attestation by the customer, to the measurements to request from the target cloud server; (2) the target cloud server implements a `Monitor Module` that can collect these measurements,

and a `Trust Module` with a `Crypto Engine` that can securely hash and sign the measurements and send them back to the Attestation Server. (3) the `Property Interpretation Module` in the Attestation Server is able to verify the measurements and auxiliary information, and interpret if the security property is satisfied.

Property Mapping and Interpretation. The Attestation Server has a mapping of security property **P** to measurements **M**. This gives a list of measurements **M** that can indicate the security health with respect to the specified property **P**. The Attestation Server can also behave as the property interpreter and decision maker: when it receives the actual measurements **M'** from the server and VM, it can judge if the customers' requested security properties are being enforced. (A simpler Attestation Server may just pass back the measurements **M'** without performing any interpretation or initiating any remediation responses.)

There are many possible security properties that a customer may want. They may include specific properties related to the cornerstone security properties of confidentiality, integrity and availability. We illustrate below with a few examples to show that *CloudMonatt* is flexible enough to support a variety of detection mechanisms. More complicated cases can be found in Chapters 4, 5 and 6. Other new methods can easily be integrated into the *CloudMonatt* framework.

3.3.1 Startup and Runtime Integrity

A customer wants to check the integrity of both the host platform and the VM before launching his VM in the cloud. Besides, he may also want to know if his VM is infected with malware during runtime,

Example Attacks: Attackers (inside-VM or outside-VM) may try to launch a malicious hypervisor, host OS, or guest OS. These software entities could have been corrupted during storage or network transmission. Similarly, the VM image could have been compromised, with malware inserted.

The attacker can spread a virus into the customer's VM. Then the malware inside-VM can compromise the customer's critical programs. Once the malware gets root privilege in the OS, it can compromise the whole VM.

Monitoring Mechanism: For VM launch integrity checking, the monitoring mechanism involves accumulated cryptographic hashes of the software that is loaded onto the system, in the order that they are loaded. A standard TPM chip can be used, and integrated into the hardware platform. The measurement is typically done in two phases: First, the server's platform configuration (hypervisor, host OS, etc.) is measured (i.e., hashed) during server bootup. Second, the VM image is measured before VM launch.

The Attestation Server can have full knowledge of the attested software, and the correct pre-calculated hash values of its executable files. It can use these correct values to check the hash measurements sent back by the cloud server, and issue the integrity property attestation, if the hash values match. Alternatively, the Attestation Server can use a trusted Appraiser system (like an Integrity Measurement Architecture (IMA) [186]) to check if the measured hash values conform to the correct values for a pristine, malware-free system, before sending the Startup Integrity Property attestation back to the customer.

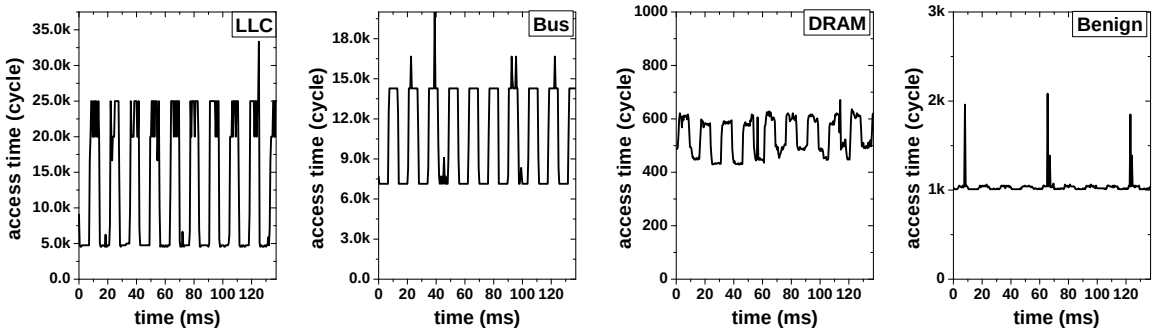
For VM runtime integrity checking, a common technique uses VM Introspection (VMI) [95, 81, 93] implemented as a hypervisor-level monitor. VMI allows the hypervisor to monitor the VM from outside the VM, and examine the states of the target VM. Different VMI tools have been designed to detect and analyze the malware inside the VMs, such as VMwatcher [127] and Ether [79]. These tools can be integrated into *CloudMonatt*. For example, when customers ask to check if there is malware running as a background service and hiding itself in the target VM, the Attestation Server can issue a request for getting the list of running tasks for that VM. The `VM Introspection Tool` located in the hypervisor's `Monitor Module` can probe into the

target VM's memory region to obtain the running tasks list [127]. This information will be written into the `Trust Evidence Registers` and transmitted back to the Attestation Server. The customer can compare this actual task list in the returned Attestation Report with the one he gets from querying the corrupted guest OS, to detect the malware running in his VM. More examples of virtual machine introspection can be found in Chapter 6.

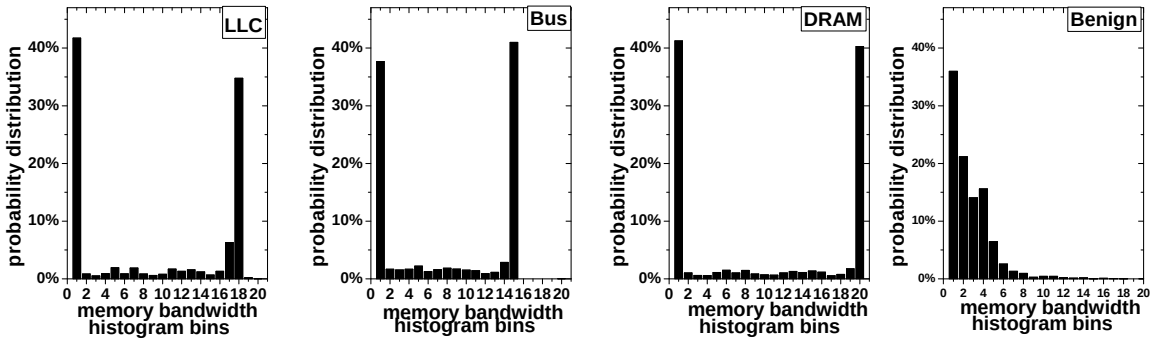
3.3.2 Runtime Confidentiality Breach through Covert Channels

For VMs with confidential code or data, cryptography is typically used to protect confidential data-at-rest and data-in-transit. However, during execution, the confidential data is decrypted and any secret key being used is also decrypted. During this time, although VMs are protected (isolated) from each other by the hypervisor, it may still be possible to leak the secret crypto key used via a cross-VM covert or side channel. In this section, we show how *CloudMonatt* detects the existence of micro-architectural covert channels. Chapter 5 will demonstrate a method of detecting and mitigating cache side channels.

Example Attacks: A covert channel exists when a colluding insider (e.g., a program inside the victim VM) can use a medium not normally used for communications to leak secret information to an unauthorized party in another VM. No security policies are overtly broken by overt communications, but are broken by covert communications. When VMs on the same server share physical resources, the contention for these shared resources can leak information, e.g., in the form of timing features. Such characteristics can be different cache operations (hit or miss) [182, 247], memory bus activities (locked or unlocked bus) [244], or DRAM controller states (bandwidth saturated or not). Figure 3.4a shows the covert channel information observed by the receiver VMs, using each of the Last Level Cache (LLC), bus and DRAM as the covert



(a) Covert-channel attacks.



(b) Detection of covert channels.

Figure 3.4: Frequency distribution detection of three covert channels verses a benign program.

channel communication medium. It also shows the receiver's observations when the sender VM runs a benign application (Apache).

Monitoring Mechanism: A key idea to detect these covert channels is that *programs involved in covert channel communications give unique patterns of the events happening on these hardware* [62]. If a customer requests covert channel protection and periodic attestation of this, *CloudMonatt* can use `Hardware Performance Counters` to monitor the attested VM's memory bandwidth every 0.1ms. After a certain monitoring period, *CloudMonatt* calculates the frequency distribution histogram for the memory bandwidth used. Specifically, it divides the entire range of observed bandwidth values into 20 bins with equal size, and then counts how many bandwidth values fall into each bin. Then *CloudMonatt* uses 20 `Trust Evidence Registers` to store the number of values in each bin to represent the memory bandwidth distribution.

These 20 values are sent as the security health measurements for detecting these LLC, bus or DRAM covert channels. We use 20 bins in our experiment, but a different number can be used to save space or increase accuracy.

Covert-Channel Property Interpretation: When the Attestation Server receives the 20 values, the `Property Interpretation Module` calculates the probability distribution (shown in Figure 3.4b) of the memory bandwidth. If a covert channel exists, the distribution graph gives two peaks: each peak representing the activity of transmitting a "0" or a "1", respectively. On the contrary, a benign application tends to give multiple smaller peaks. The Attestation Server can use machine learning techniques to conduct pattern recognition of covert channels. Once the Attestation Server identifies the existence of attacks, it can perform further actions (e.g., sending warnings to the customers and asking them to double check malicious processes) to reduce false positives. This detection method might also introduce some false negatives since it cannot cover all covert-channel attacks. More sophisticated detection methods can be integrated into *CloudMonatt* to detect other types of attacks.

3.3.3 Runtime CPU Availability

Availability of the resources and services agreed upon by the cloud customer and the cloud provider in the Service Level Agreement (SLA) is a very important security problem in cloud computing. Even if over-provisioning is practiced, the cloud provider is still responsible for providing a fair resource allocation for each VM based on its SLA. During runtime, the customer wants to know if his VM is given the requested resources as paid for. We now show an example of an availability attack, and how CPU resource availability can be monitored.

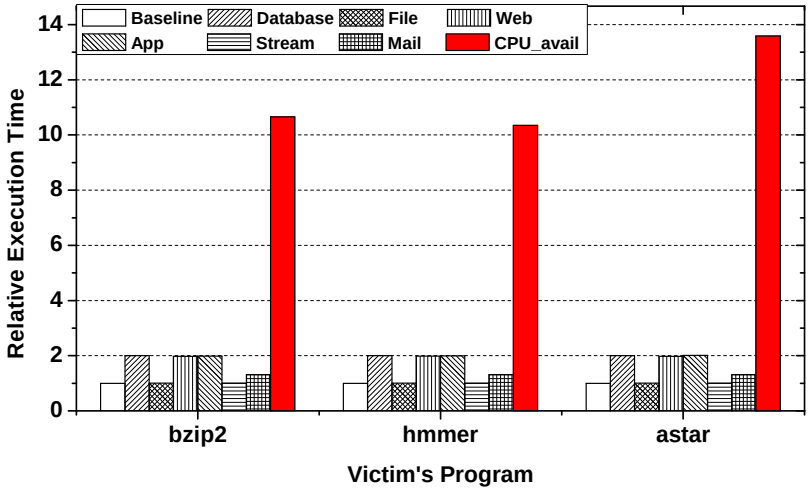
Example Attacks: An attacker may try to get more resources to severely reduce the availability of shared resources to a victim VM, thus degrading its performance. This may be to improve the attacker's own performance, or it may just be to attack

the victim and deny him his rightful use of cloud resources. We demonstrate a new CPU resource availability attack, and use it as an example of resource availability monitoring in *CloudMonatt*.

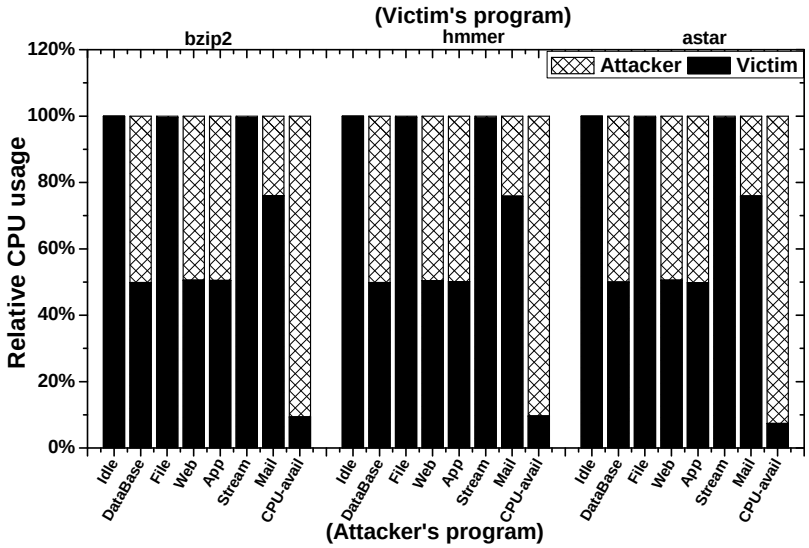
CPU resource availability attack: This attack targets the boost mechanism of Xen's credit scheduler algorithm. Specifically, each VM receives some credits periodically, and the running VM pays out credits. The Xen scheduler wakes up the VM with extra credits in Round-Robin order. However, when a VM is woken up by certain interrupts, it always gets higher priority to take over the CPU. So the attacker's strategy is to launch a VM with multiple vCPUs and use them to keep sending and receiving Inter Processor Interrupts (IPIs) to each other, so one of the attacker's vCPUs always has the highest priority. Since the attacker's VM always has higher priority than the victim VM, they consume a lot of CPU resources, thus starving the victim's CPU usage.

Figure 3.5a shows the results for the denial of CPU service attack. The attacker VM and victim VM are located on the same CPU using a Xen hypervisor. The victim VM runs three CPU-bound programs from the SPEC2006 benchmark suite. The attacker VM runs different services typically done in the cloud, as well as the CPU availability attack we designed. When the attacker is I/O-bound (File, Stream or Mail servers), the attacker does not consume much CPU and the victim VM has no performance degradation. When the attacker runs CPU-bound tasks (Database, Web or App servers), the victim's execution time is doubled since it can get a fair share of 50% of the CPU quota. However, when the attacker performs the CPU availability attack described above, the victim's performance is degraded by more than ten times.

Monitoring mechanism: The basic idea for availability monitoring is to measure the resource usage of the attested VM, e.g., CPU usage in this example. During the testing period for CPU availability, the `VMM Profile Tool` measures the attested VM's CPU time: it observes the transitions of each virtual CPU on each physical



(a) Performance under attacks.



(b) Measurements of vulnerability.

Figure 3.5: CPU availability attacks and detection

core, and keeps record of the virtual running time for the attested VM. After the testing period, the `VMM Profile Tool` stops the measurements and calculates the total virtual running time: $CPU\_measure$. This measurement is written into one `Trust Evidence Register`, signed and sent back to the Attestation Server.

Availability Property Interpretation: The Attestation Server retrieves the attested VM's virtual running time and calculates the relative CPU usage as the ratio of a VM's virtual running time to real time. If the relative CPU usage is very small,

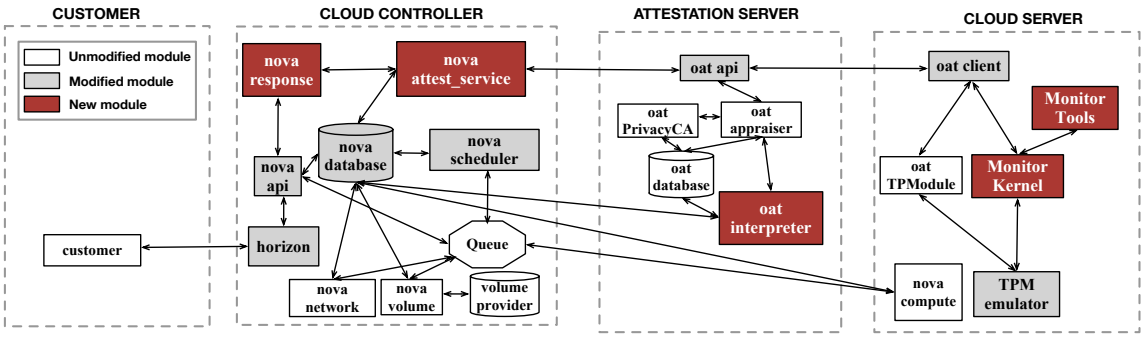


Figure 3.6: Implementation of attestation architecture.

then the Attestation Server interprets the VM's CPU availability as compromised (as shown in Figure 3.5b).

3.4 Evaluation

3.4.1 Prototype Implementation

We implemented our property-based cloud attestation on the OpenStack platform [19]. We integrated the OpenAttestation software (oat) [17] for host remote attestation protocols. We integrated the TPM-emulator [209] and leveraged it to emulate the functions of the `Trust Module` in the hardware. Our evaluation results in Section 3.4.2 show that the emulation of the `Trust Module` has little impact on the system performance. We make *CloudMonatt* open-source and available online [1]. Figure 3.6 displays our prototype implementation.

Cloud Controller. The Cloud Controller is implemented by the OpenStack. OpenStack is composed of different services. We modified two services. The first one is *horizon*, which is implemented as OpenStack's dashboard and provides a web-based user interface to customers. The second one is *nova*, which is used to manage computing services in cloud servers. We modify four modules in the Cloud Controller:

[1] https://github.com/eepalms/CloudMonatt.git

- *horizon*: We extend the VM launch interface with the monitoring and attestation options: when launching VMs, the customers can specify which properties they want to monitor for their VMs. When the cloud provider searches for a destination machine for initial VM allocation or migration, it must choose servers which support such properties. We also added new options (Table 3.1) for customers to monitor the VM's health. The customers provide the security properties they want to monitor for their VM, and they will receive the attestation results.
- *nova api*: we modify this module to pass new VM launch options, monitoring requests from *horizon* to *nova*, as well as attestation results from *nova* to *horizon*.
- *nova database*: We modify the controller's database to enable it to store the customers' specifications about the security properties required for their VMs, from *nova api*. We also add new tables in the database, which record each servers' monitoring and attestation capabilities: i.e., what properties they support for monitoring.
- *nova scheduler*: the *nova scheduler* is modified to implement the `Policy Validation Module` and `Deployment Module` of the Cloud Controller in Figure 3.1. It is responsible for choosing the host for the VM during initial allocation and migration. The default scheduler in OpenStack is to choose the server with the most remaining physical resources, to achieve workload balance. We add a new filter: *property_filter*, to select qualified cloud servers to host VMs based on their customers' security properties, monitoring and attestation requirements.

We add two new modules in the controller:

- *nova attest_service:* This essential module manages the attestation services. It connects *nova database* (for retrieving security properties), *oat api* (for issuing attestations and receiving results) and *nova response* (for triggering the responses).

- *nova response:* This implements the `Response Module` in Figure 3.1. It is responsible for providing some responses if the attestation fails, as discussed in Section 3.2.5.

Attestation Server. The attestation server and client are realized by OpenAttestation. The Attestation Server has four main modules: *oat database* stores information about the cloud servers and measurements; *oat appraiser* is responsible for triggering attestations and reporting the measurements; *oat PrivacyCA* provides public-key certificates for the cloud servers. We modify *oat api* and add a new module *oat interpreter*:

- *oat api:* We extend the APIs with more parameters, i.e., security properties and VM id.
- *oat interpreter:* This essential new module implements the Property Interpretation and Certification Modules of the Attestation Server. It can interpret the security health of the VM and make attestation decisions, based on the information of the cloud server from the *nova database* and the security measurements from the *oat database*.

Cloud Servers. In each cloud server, *nova compute* is the client side of OpenStack nova. We modify *oat client*, the client side of OpenAttestation, to receive attestation requests. We modify the *TPM emulator* to provide secure storage and crypto functions. We add two new modules:

- *Monitor Kernel* can start the security measurements and store the values into the *TPM emulator*, and
- *Monitor tools* can integrate different software VMI tools, VMM Profile tools or other logging or provenance tools, into the server to perform the monitoring and take measurements.

3.4.2 Performance Evaluation

Our testbed includes three Dell PowerEdge R210II servers, each with a quad-core 3.30 GHz Intel Xeon processor, 32GB RAM, and on-board dual Gigabit network adapter with 1 Gbps speed. We select one server as the cloud controller, equipped with Nova Controller and OpenAttestation Server. The other two servers are implemented as cloud server nodes.

We consider two performance issues: the overhead of VM launching due to new security requirements, and the overhead of attestation during runtime. We also evaluate different responses for attestation failure recovery. OpenStack Ceilometer [18] is exploited for timing measurements.

VM Launch: In the original OpenStack platform, VM launch involves the following four steps:

- *Scheduling*: allocate VMs to appropriate servers based on customers' requirements and servers' workloads.
- *Networking*: allocate the networks for VMs.
- *Block_device_mapping*: set up block devices for VMs.
- *Spawning*: start VMs on the selected servers.

Our OpenStack *CloudMonatt* architecture involves five steps for VM launching. At the *Scheduling* stage, the controller needs to check *oat database* to find qualified servers which have the security features that support the customer's desired security properties. Steps 2, 3, and 4 are the same as above. We add a fifth stage *Attestation* after the *Spawning* stage. This stage will check if the VM has been launched securely.

Figure 3.7 shows the time for each stage of VM launching. We test three VM images (cirros, fedora and ubuntu) with three VM flavors (small, medium and large). This figure shows that the overhead of the *Attestation* stage is about 20%, which

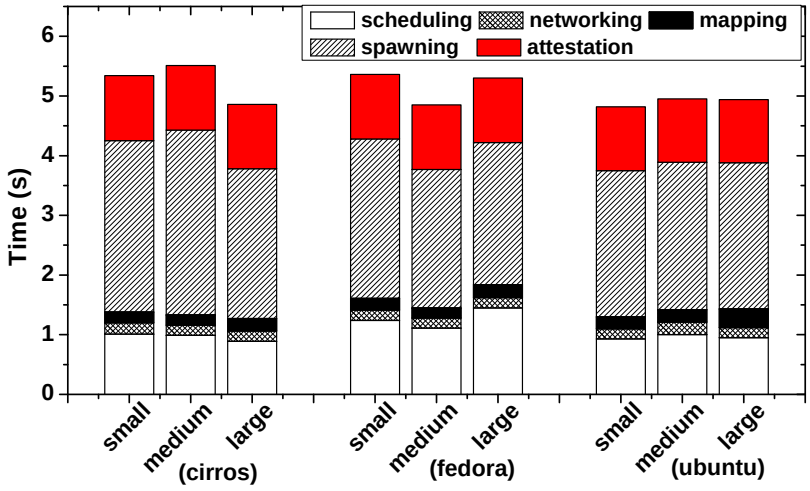


Figure 3.7: Performance for VM launching.

is acceptable for VM launching. The main overhead of an attestation is from the message transmitting in the network.

VM Runtime: During VM runtime, customers can monitor the VM at any time, or periodically at a given frequency. To test the performance effect of periodic runtime attestation, we ran different cloud benchmarks in one virtual machine, while the customer issues the periodic runtime attestation request at different frequencies. Figure 3.8 shows the effect of periodic runtime attestation at a frequency of 1 minute, 10 seconds and 5 seconds, on ubuntu-large VM.

This figure indicates that there is no performance degradation due to the execution of runtime attestation. This is for CPU-resource monitoring, where the measurements are taken during the VM switch – the VMM Profile Tool does not intercept the VM's execution. Whether runtime attestation causes performance degradation to the VM execution time depends on the measurement collection mechanism. However, if the periodic attestation frequency is low, then the performance effect is negligible.

Response: The effectiveness of attestation in preventing runtime security breaches depends on two factors: (1) how long it takes to detect potential exploitation of vulnerabilities. This is related to attestation time and mode; and (2) how long it

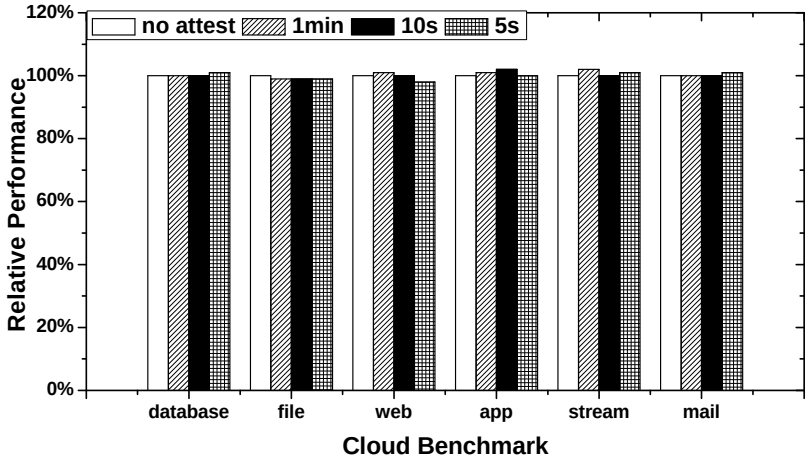


Figure 3.8: Performance effect of runtime attestation.

takes to perform the remediation responses. We evaluate the overhead of the defense strategies described in Section 3.2.5.

Figure 3.9 shows the attestation time and reaction time for each response strategy, providing insights into which strategy should be used. Two factors influence the choice of a response: (1) The reaction time of the response should ideally be less than the "damage time", where we define "damage time" as the time from the point at which the attack is detected to the point at which damage results from the attack. In this respect, Termination is the fastest while Migration is the slowest. (2) The response strategy should also be determined by the specific nature of the attacks and the customers' security needs and usage scenarios. For example, Termination sacrifices VM availability as the customer cannot use the VM any more; Suspension enables the customer to continue the VM only after the server recovers from security breaches; Migration enables the customer to use the VM immediately after the migration is done. So migration may be the best for service availability.

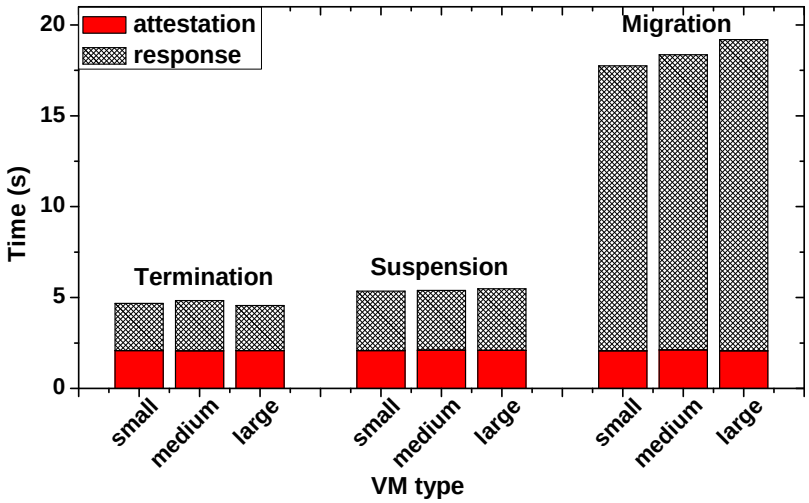


Figure 3.9: Attestation reaction times during VM runtime.

3.5 Appendix to Chapter 3: Security Verification

It is necessary to verify *CloudMonatt* to guarantee that there are no security vulnerabilities that could allow an attacker to subvert its protection. We conduct a security evaluation and verification of *CloudMonatt*. We aim to address two questions: (1) can *CloudMonatt* provide unforgeable VM health reports to customers as well as the cloud provider? (2) What are the minimal security requirements (i.e., minimal Trusted Computing Base) that can guarantee the security and correctness of *CloudMonatt*?

3.5.1 Verification Methodology

To verify a system's protocols and operations, we first specify the verification goals and invariants based on the system's functionality. Then we build models for the system, and identify the trusted and untrusted subjects in the system. We implement the models and verification invariants in model checker tools and run the tools to test if the invariants pass for every possible path through the system models from the initial state to the end state. If an invariant fails in some cases, we try to find the

vulnerabilities and construct the corresponding attacks. We describe these steps of verifying *CloudMonatt* in detail below.

Analyzing verification goals. *CloudMonatt* has two basic functionalities: (1) reporting VMs' potential security threats to the cloud provider so it can take the corresponding countermeasure to mitigate the threats; (2) notifying the customers of their VMs' security health. So *CloudMonatt* must ensure that the cloud provider and customers can receive the correct and unforgeable monitoring reports. These are the two verification goals of *CloudMonatt*.

Figure 3.10 shows the structure of verification goals and their dependent conditions. The two red blocks show the two main goals we want to verify: (1) the goal that the customers can receive the correct reports depends on three conditions: the Cloud Controller can receive the correct reports, process them correctly and transmit them securely to the customers. (2) The goal that the Cloud Controller can receive the correct reports also depends on three conditions: the Attestation Server can receive the correct measurements, process them (i.e., generate correct reports) correctly, and transmit the reports to the Cloud Controller securely. In addition to the above two main goals, the condition that the Attestation Server can receive the correct measurements depends on two conditions: the cloud server collects correct measurements, and such measurements can be transmitted to the Attestation Server securely. The trustworthiness of each server depends on two conditions: the critical software and hardware modules function correctly, and messages are exchanged securely between these modules.

In order to verify the main goals in a scalable way, we classify these goals and conditions into different types, and break the verification task into two steps, adapting and extending the methodology from [216]. The first step is *external verification*, which aims to verify the main verification goals (red blocks in Figure 3.10). In this step, we treat each server as a blackbox (dashed boxes in Figure 3.10). For each server we only consider the black block as a precondition and assume it is already satisfied,

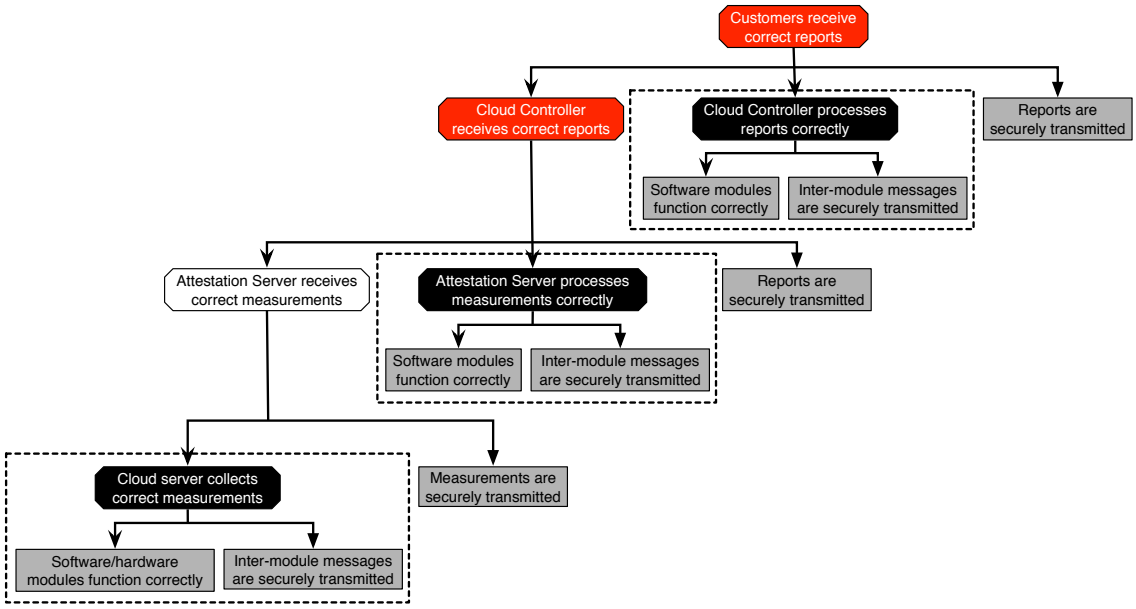


Figure 3.10: The structure of verification goals of *CloudMonatt*. Red blocks are the main goals of *external verification*. Black blocks are the preconditions of *external verification*, as well as the postconditions of *internal verification*. Grey blocks are the basic preconditions.

while ignoring other basic preconditions (grey blocks) inside the boxes. Under such preconditions and other basic preconditions outside of the blackboxes, we verify if the main goals are held. The second step is *internal verification* in which we consider the activities inside each server. In this step, the precondition we assume in the previous step becomes the postcondition that we want to verify. We want to check if such postcondition is held, i.e., the precondition we make in the previous step is correct, under the basic preconditions inside the dashed box.

Modeling systems. To verify the above goals of *CloudMonatt*, we need to translate the system protocols and the underlying architectures into representative yet tractable models. We adopt the symbolic modeling method [52], where the cryptographic primitives are represented by function symbols and perfect cryptography is assumed. Specifically, we first specify subjects involved in this verification procedure. A subject can be a customer or a server in the distributed system, or a hardware/software module inside a server. For *external verification*, since we treat each server as a blackbox, then

we model each server and the customer as a subject. For *internal verification*, we need to consider the internal activities inside the server, so we model each software and hardware module involved in the system operation as a subject. Each subject has a set of states with inputs and outputs based on the system operation. The transitions between different states are also defined by the architecture designs and protocols.

Among all the subjects, there is an initiator subject that starts the system protocol and a finisher subject that ends the protocol (The initiator and finisher could be the same subject). This initiator subject has a "Start" state while the finisher subject has a "Done" state. The verification procedure starts at the initiator's "Start" state. At each state in each subject, it takes actions corresponding to the transition rules. It will exhaustively explore all possible rules and states to find all the possible paths from the initiator's "Start" state to the finisher's "Done" state. Then we judge if the verification goals are satisfied in all of these paths. The system is verified to be secure if *there are paths from initiator's "Start" state to finisher's "Done" state, and all the verification goals are satisfied in any of these paths.*

Specifying security invariants. Invariants are conditions that need to hold true for there to be no violation of the verification goals or postconditions. The invariants can be specified from the goals or postconditions that we want to verify. For *CloudMonatt*, the goals of *external verification* are to ensure the customer and the Cloud Controller receive the correct reports. So the invariants are that *the reports received by the customer and the Cloud Controller are always the ones matching the security property and VM id they specify.* The postconditions for *internal verification* are to ensure that the servers process the data correctly. So the invariants are that *the output (e.g., measurements, report) sent from the server are always the ones correctly mapped to the input sent to the server.*

Identifying preconditions. Preconditions refer to the basic requirements that are needed to keep the security invariants true within the system protocols or operations.

Basically it specifies the necessary subjects (e.g., network links connecting different servers, software or hardware modules inside the server) that should be trusted. For *external verification*, the preconditions are the assumptions we make about each servers. For *internal verification*, the preconditions are the subjects that should be included in the Trusted Computing Base. The verification results can help us identify the minimal TCB for *CloudMonatt*, i.e., the necessary and critical software/hardware modules or servers that should be well protected to guarantee the correctness of *CloudMonatt*.

In the next two sections, we conduct the *external verification* and *internal verification* separately. We use *ProVerif* [51] to model the system and verify the security invariants. *ProVerif* is a software tool for checking security properties in cryptographic protocols. It supports a variety of cryptographic primitives, e.g., symmetric and asymmetric cryptography, digital signatures, hash functions, etc. If a security property is proven unsatisfied, *ProVerif* can reconstruct the attacker execution trace that falsifies the property. We show how to use *ProVerif* to check the system interactions, in addition to network protocols.

3.5.2 External Verification

Modeling. We model each server involved in this distributed system as an interacting state machine, as shown in Figure 3.11. Each subject is made up of some states. The customer is the initiator as well as the finisher subject. The whole process starts from the customer side, who sends to the Cloud Controller the attestation request including the VM identifier **Vid** and the desired security properties **P**. Then the Cloud Controller discovers the host cloud server, and forwards the request to the Attestation Server, with the server identifier **I**. The Attestation Server identifies the necessary monitoring measurements and sends the measurement request **rM** to the host cloud server. The cloud server collects the required measurement, calculates the hash value, **Q**, of the measurements requested and sends these values back to the Attestation

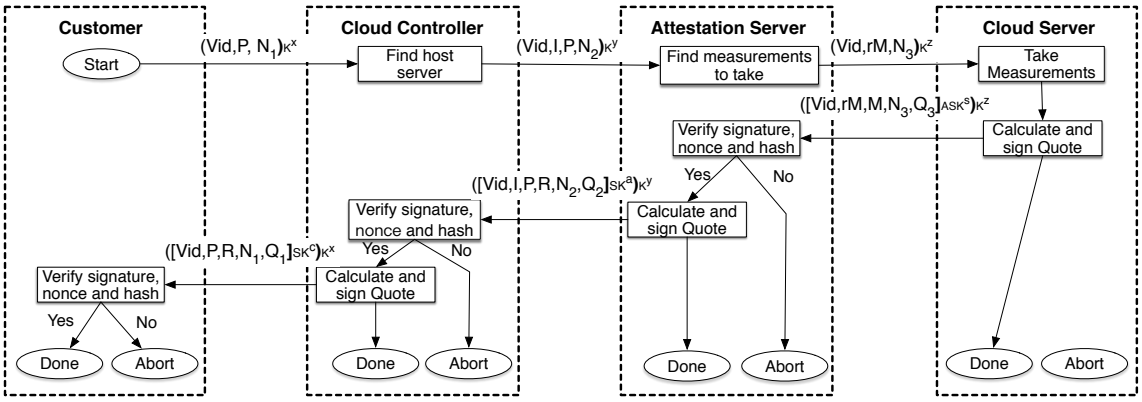


Figure 3.11: The external protocol in *CloudMonatt*. $\mathbf{K}^x$, $\mathbf{K}^y$ and $\mathbf{K}^z$ are symmetric keys between the customer and the Cloud Controller, between the Cloud Controller and the Attestation Server, and between the Attestation Server and the cloud server. $\mathbf{SK}^c$, $\mathbf{SK}^a$ and $\mathbf{ASK}^s$ are the private signing keys of the customer, the Cloud Controller, the Attestation Server and the cloud server. $\mathbf{N}_1$, $\mathbf{N}_2$ and $\mathbf{N}_3$ are the nonces used by the customer, the Cloud Controller and the Attestation Server.

Server, after which the cloud server reaches the "Done" state. The Attestation Server checks the signature, the hash value and the nonce: if this check fails, the Attestation Server goes to "Abort" state. Otherwise it interprets the measurements and the property, and generates the attestation report $\mathbf{R}$, as explained in Section 3.2.4. Then the Attestation Server signs the report, transmits it to the Cloud Controller, and goes to state "Done". After receiving the report, the Cloud Controller checks the signature, the hash value and the nonce. If anything is incorrect, the Cloud Controller goes to state "Abort". Otherwise it hashes and signs the report, and ends at state "Done" after sending the report to the customer. If the customer finds the encrypted signature of the report is correct, it goes to state "Done". Otherwise, it goes to state "Abort".

Security invariants. As we discussed in Section 3.5.1, the external verification is to check if the customer and cloud provider can receive the correct attestation reports. We identify several specific security invariants for this task in our modeled state machines:

① The Cloud Controller is able to reach state "Done". When it is at state "Done", the attestation report **R** it receives is indeed the one for VM **Vid** with property **P**, specified by the customer.

② The customer is able to reach state "Done". When he is at state "Done", the attestation report **R** he receives is indeed the one for VM **Vid** with property **P**, specified by the customer.

Invariant ① is to ensure the Cloud Controller gets the correct attestation reports. Invariant ② is to ensure the customer gets the correct attestation reports.

Preconditions. We make several preconditions about each server and check if the above security invariants can be satisfied under these preconditions. These preconditions indicate the subjects that should be included in the TCB. Verifying the sufficiency and necessity of these preconditions can help us find the minimal TCB for *CloudMonatt*.

(C1) The cloud server is trusted.

(C2) The Attestation Server is trusted.

(C3) The Cloud Controller is trusted

Here a "trusted" server means it will strictly follow the operations indicated in our protocol. For instance, a trusted cloud server will collect and sign correct measurements; a trusted Attestation Server will process the measurements and generate the reports correctly; a trusted Cloud Controller will process the VM health reports correctly. In addition, a trusted server will keep its secrets from attackers.

Implementation. We model the authentication and communication protocols of *external verification* in *ProVerif*. Specifically, we declare each subject (the customer, the Cloud Controller, the Attestation Server and the cloud server) as a process. Inside the process we model the operations of state machines shown in Figure 3.11. Each process keeps some secrets (e.g., cryptographic keys, attestation reports or

measurements). If the subject is trusted, then the attacker cannot get these secrets, and we use the keyword `private` to denote these variables. Otherwise the variables are declared as public and the attacker can obtain the values.

To model the network activities in this distributed system, we declare a `channel` between each pair of subjects, to represent the untrusted communication channel. These channels are under full control of the network-level adversaries, who can eavesdrop or modify any messages transmitted in the channels.

We also use the cryptographic primitives from *ProVerif* to model the public key infrastructure for digital certificate, authentication and key exchange. Then we model all the steps in Figure 3.11 for an unbounded number of attestation sessions, i.e., the customer keeps sending attestation requests to the cloud system and receiving the reports. *ProVerif* can check if the adversary can compromise the integrity of the report in any attestation session, and display the attack execution trace if a vulnerability is found.

We can use *ProVerif*'s reachability proof functionality to verify if the Cloud Controller and customer is able to reach state "Done". *ProVerif* allows us to define an event **E** inside a process at one state, which specifies some conditions. Then we can check if this event will happen when the protocol proceeds using the query statement: "`query event`(**E**)". *ProVerif* can enumerate all the possible execution traces and check if this event is reachable in some cases. If so, this query statement returns true as well as the trace that reaches the event. Otherwise the statement returns false. So we can use the statement "`query event`(**Done**)" to check if the customer and Cloud Controller can receive the attestation report.

ProVerif does not provide direct functionalities to prove integrity. However, we can also use its reachability proof functionality to verify the integrity property of a message. Specifically, to verify the integrity of the attestation report in invariant ①, we check if the report received by the Cloud Controller, **R'** is the correct one, **R**,

determined by the VM identifier **Vid** and the security property **P**, when the Cloud Controller reaches state "Done". Then we establish an event: "(**R'**≠**R**)" at state "Done" to denote the integrity breach. We use the statement "`query event`(**R'**≠**R**)" to verify the integrity. If this statement is false, it means the attacker has no means to change the message **R** without being observed by the Cloud Controller. Then the integrity of **R** holds. Similarly, to verify invariant ②, we check if the report **R'** received by the customer at state "Done", is the correct one **R**.

Results. *Proverif* shows that state "Done" is reachable for both customers and Cloud Controller. Then we verify if the security invariants ① and ② are satisfied under the preconditions (C1) – (C3). First, *ProVerif* confirms that preconditions (C1) – (C3) are sufficient to guarantee that the customer and the Cloud Controller can receive the correct attestation reports. Note that as we put trust on the Cloud Controller, the Attestation Server and the cloud server, we do not need to consider the server-level adversaries. Even though the network-level adversaries can take control of all the network channels between each server, they cannot compromise the integrity of the messages without being observed, since all the messages are hashed, signed and encrypted before being sent to the network.

Second, we check if preconditions (C1) – (C3) are necessary to keep the invariants correct. *ProVerif* shows that it is necessary to place the subjects of (C1), (C2) and (C3) in the TCB. Missing any precondition can lead to violations of some invariants: if the cloud server is not trusted, then the server-level adversary can counterfeit wrong measurements, causing the Attestation Server to make wrong attestation decisions, and pass them to the Cloud Controller and the customer. So invariants ① and ② are not satisfied. If the Attestation Server is not trusted, then it can generate wrong attestation reports for the customer and the Cloud Controller. So invariants ① and ② are not satisfied. If the Cloud Controller is untrusted, it can modify the attestation reports before sending to the customer. So invariant ② is not satisfied.

In the next section, we perform *internal verification* of the trusted servers, to show which component in each of the servers should be trusted, in order to satisfy the preconditions we assume in this section.

3.5.3 Internal Verification

From Section 3.5.2 we know that to satisfy the external verification goals, we need to assume the correctness of the preconditions in each server, i.e., trusting the data processing in the Cloud Controller, the Attestation Server and the cloud server. However, we do not need to place the entire server into the TCB. On the one hand, trusting each component in one server is not a necessary condition to satisfy the precondition we assume for this server. Including all the components of the server into the TCB would require stronger security protection for the entire server, which is expensive and difficult to achieve. It is also impossible to trust every component in the server, especially for the cloud server which hosts the guest VMs rented to the customers. *CloudMonatt* cannot ensure that the guest VMs are trusted. As such, we conduct the *internal verification* to identify which components inside the server need to be trusted, to satisfy the preconditions we make in the *external verification*.

3.5.3.1 Cloud Server

First, we verify the system operation and interactions on the cloud server.

Modeling. We abstract the key components inside a cloud server, and model them as state machines, as shown in Figure 3.12. Besides, we also include the Attestation Server to interact with the cloud server. The Attestation Server is the initiator and finisher subject in the internal protocol. The whole process starts when the Attestation Server sends the measurement request to the cloud server. The `Attestation Client` processes the request and passes it to the `Monitor Module`. The `Monitor Kernel` inside the `Monitor Module` figures out the corresponding monitor tool and invokes it to

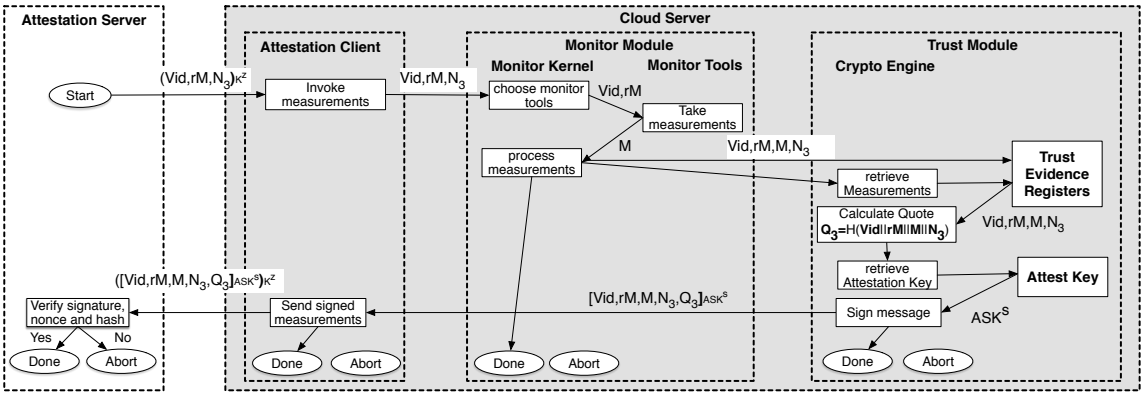


Figure 3.12: Internal protocol (interactions) in the cloud server

collect the correct measurements. Then it stores the measurements together with other related information in the `Trust Evidence Registers`. Then the `Crypto Engine` in the `Trust Module` retrieves the measurements, calculates the quote (see Section 3.2.4) and signs it using the `Attestation Key`. Then the signature is encrypted by the `Attestation Client` and sent to the Attestation Server. After this all the subjects inside the cloud server reach the state "Done". The Attestation Server will check the hash and signature. It goes to state "Done" if the check succeeds and state "Abort" if the check fails.

Security invariants. the invariant for *internal verification* is to check if the cloud server collects the correct measurements and sends these to the Attestation Server. So we translate this invariant to the statement as below:

① The Attestation Server is able to reach state "Done". When it is at state "Done", the measurements **M** it receives are indeed the one for VM **Vid** with request **rM**, which were sent to the cloud server.

Preconditions. We identify a set of possible preconditions to satisfy the above invariant. We classify these preconditions as different modules and inter-module communications. We check the necessity and sufficiency of these preconditions for guaranteeing the integrity of measurements taken from the cloud server.

1. `Attestation Client`:

(C1.1) this module is trusted.

2. `Monitor Module`:

(C2.1) the `Monitor Kernel` is trusted.

(C2.2) the `Monitor Tools` are trusted.

(C2.3) the channel between the `Monitor Kernel` and the `Monitor Tools` is trusted.

3. `Trust Module`:

(C3.1) the `Crypto Engine` is trusted.

(C3.2) the `Trust Evidence Registers` are trusted.

(C3.3) the `Attestation Key` is securely stored.

(C3.4) the channel between the `Attestation Key` and the `Crypto Engine` is trusted.

(C3.5) The channel between the `Crypto Engine` and the `Trust Evidence Registers` is trusted.

4. Inter-module communication:

(C4.1) the channel between the `Attestation Client` and the `Monitor Kernel` is trusted.

(C4.2) The channel between the `Attestation Client` and the `Crypto Engine` is trusted.

(C4.3) The channel between the `Monitor Kernel` and the `Trust Evidence Registers` is trusted.

Implementation. *ProVerif* does not provide functionalities for modeling and verification of architecture-level interactions. However, we can model the server system as a network system, and verify the server in a similar way as the network protocol verification. Specifically, we can model a software or hardware component as a process. Each component keeps some variables and operates as a state machine. If one component is in the TCB, then its variables will be declared as `private`. Otherwise

its variables are public to attackers. If the attacker has the privilege to control the communication between two components, then we declare a public `channel` for these two components. On the contrary, if two modules are linked by one channel that is trusted, then we combine the two processes into one process so that the two modules can exchange messages directly without being compromised by third party attackers.

We model all the steps in Figure 3.12 for an unbounded number of sessions, i.e., the Attestation Server keeps sending measurement requests to the cloud server and receiving the results. *ProVerif* enumerates all the possible states during the infinite sessions and checks if the property is maintained.

Results. *ProVerif* shows that state "Done" is reachable for the Attestation Server. Then We consider and verify the sufficiency and necessity of the above preconditions that satisfy the security invariants. We use the same reachability functionality of *ProVerif* to verify the integrity property under different preconditions.

For precondition (C1.1), *ProVerif* shows that the integrity property is still satisfied, and the adversary cannot tamper with the messages, even if he takes control of the `Attestation Client`. If the adversary changes **Vid** or **rM** before they are sent to the `Monitor Module`, the `Monitor Module` will collect the wrong measurements **M**. However, the `Trust Module` will also sign the modified **Vid** or **rM**. The Attestation Server will notice this modification and go to state "Abort". So (C1.1) is not a necessary condition and can be removed from the TCB.

For (C2.1), (C2.2) and (C2.3), *ProVerif* shows that without any of the three preconditions, the integrity checking of measurements will fail. *ProVerif* also shows attack execution traces if one precondition is missing. For instance, if the `Monitor Kernel` is untrusted, it can send a different **Vid** or **rM** to the `Monitor Tools` to collect wrong measurements **M**. If the `Monitor tools` are untrusted, even they receive the correct measurement request, they will give wrong measurements data. If the communication channel is open to the adversary, he can easily modify the measurement

requests or results without being noticed by the other trusted subjects. So (C2.1), (C2.2) and (C2.3) are necessary conditions to protect the measurements' integrity, and must be kept.

ProVerif shows preconditions (C3.1) – (C3.5) are also necessary to guarantee the integrity of measurements. It also shows the attack execution traces without these conditions. If the attacker compromises the `Crypto Engine`, the `Attestation Key` or their communication channel, it can generate a fake signature over any measurements using the signing key $\mathbf{ASK}^S$, while the Attestation Server will never detect this integrity breach. If the `Trust Evidence Registers` or their connection with the `Crypto Engine` are compromised, then a server-level adversary can easily tamper with the security measurements stored in the untrusted `Trust Evidence Registers` or transmitted in the untrusted channel, without being detected by the Attestation Server.

Precondition (C4.1) is not necessary, with the same reason as precondition (C1.1). Precondition (C4.2) is not necessary. The adversary cannot compromise the message integrity since the message in this channel is signed. Precondition (C4.3) is necessary. If this channel is not trusted, the adversary can modify the measurements, $\mathbf{M}$, then the `Trust Module` will store and sign the wrong measurement.

Based on the above results, the necessary conditions to guarantee the measurements' integrity are: (1) the `Monitor Module` is trusted (i.e., the `Monitor Kernel`, the `Monitor Tools`, and their communications); (2) the `Trust Module` is trusted (i.e., the `Crypto Engine`, the `Attestation Key`, the `Trust Evidence Registers`, and their communications); (3) the communication channel between the `Monitor Module` and the `Trust Module` is trusted. *ProVerif* shows that it is also sufficient for the cloud server to maintain the property of measurements' integrity if only these subjects are included in the TCB of a cloud server. In particular, *ProVerif* shows that the `Attestation Client` need not be trusted.

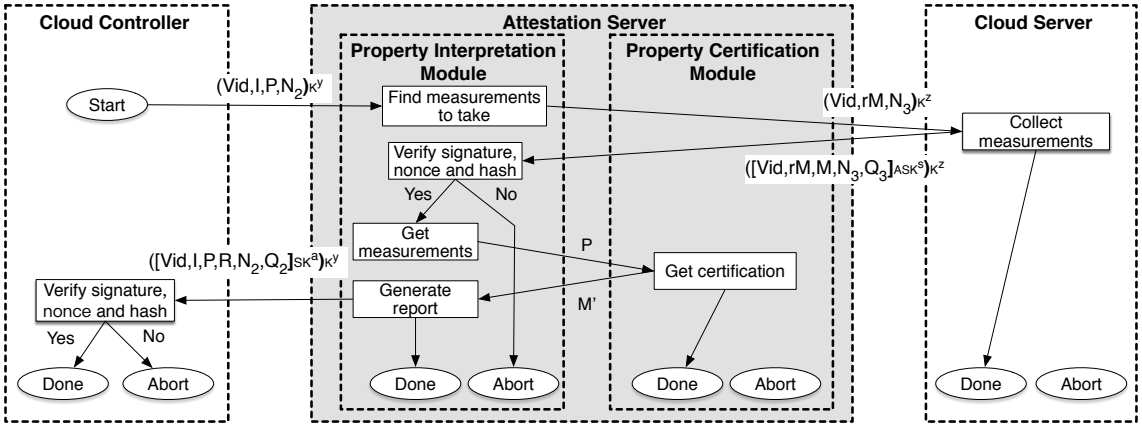


Figure 3.13: Internal protocol (interactions) in the Attestation Server

3.5.3.2 Attestation Server

We use the same method to verify the Attestation Server.

Modeling. Figure 3.13 shows the state machines of the Attestation Server. The Attestation Server has two modules: the `Property Interpretation Module` and the `Property Certification Module`. The `Property Interpretation Module` is used to invoke the attestation. The `Property Certification Module` is used to provide the measurements certification for one security property.

Security invariants. The invariant for verification of the Attestation Server is to check if the Attestation Server generates the correct attestation report and sends it to the Cloud Controller:

① The Cloud Controller is able to reach state "Done". When it is at state "Done", the report **R** it receives is indeed the one for VM **Vid** with property **P**, which were sent to the Attestation Server.

Preconditions. We identify a set of preconditions for the Attestation Server:

(C1) The `Property Interpretation Module` is trusted.

(C2) The `Property Certification Module` is trusted.

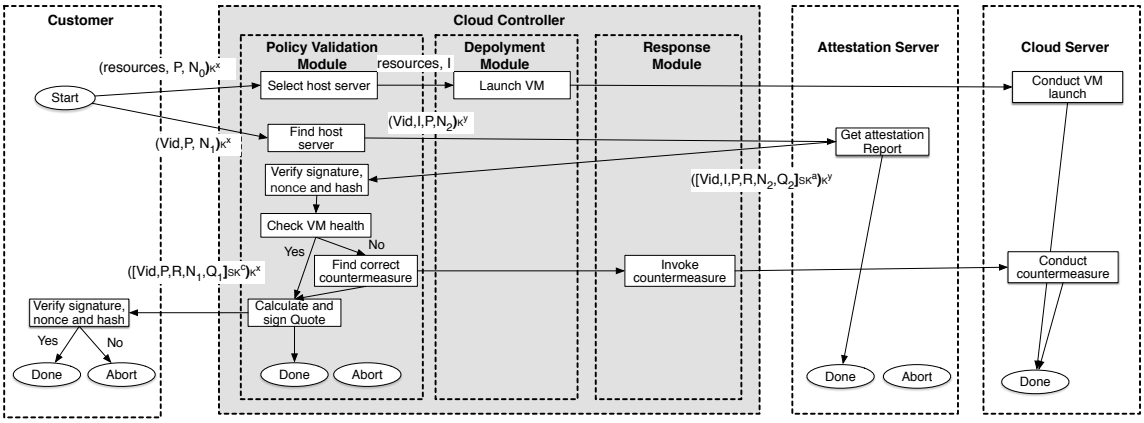


Figure 3.14: Internal protocol (interactions) in the Cloud Controller

(C3) The channel between the `Property Certification Module` and `Property Interpretation Module` is trusted.

Implementation. We use *ProVerif* to verify the Attestation Server in a similar way as the cloud server.

Results. We find that both of the two modules and their communications should be placed into the TCB of *CloudMonatt*. If the `Property Certification Module` is incorrect, it will give wrong property certifications. If the `Property Interpretation Module` is compromised, it can invoke the wrong attestation requests or send wrong attestation reports. Untrusted channels between these modules will bring the same attack effects.

3.5.3.3 Cloud Controller

Modeling. In the Cloud Controller, we consider the interactions between three modules: the `Policy Validation Module`, `Deployment Module` and `Response Module`. The state machines of these three modules are displayed in Figure 3.14. We consider two events: the customer launches the VM, and issues runtime attestation requests.

Security invariants. The invariant for verification of the Cloud Controller is to check if the Cloud Controller can send the the correct attestation report to the cloud customer.

① The customer is able to reach state "Done". When he is at state "Done", the report **R** he receives is indeed the one for VM **Vid** with property **P**, specified by the customer.

Preconditions. The preconditions for the Cloud Controller is related to the three modules involved:

(C1) The `Policy Validation Module` is trusted.

(C2) The `Deployment Module` is trusted.

(C3) The `Response Module` is trusted.

(C4) The channel between the `Policy Validation Module` and `Deployment Module` is trusted.

(C5) The channel between the `Policy Validation Module` and `Response Module` is trusted.

Implementation. The verification of the above models is similar to the cloud server and the Attestation Server.

Results. To guarantee the integrity of attestation reports, the `Policy Validation Module` must be trusted. If this module is compromised, then the whole monitoring service will be compromised. For the `Deployment Module` and the `Response Module` and their communication channels with the `Policy Validation Module`, they are not necessary to protect the integrity of attestation reports. However, they are used to control VMs. So they should also be trusted to guarantee that the cloud system functions correctly.

3.5.4 Verification Discussions

The external and internal verification results can help us verify and enhance the security of *CloudMonatt*. We identify the components that are required in *CloudMonatt*'s TCB. Then we can use existing software-hardware solutions to protect these components.

In a cloud server, verification results show that the `Monitor Module` and `Trust Module` of a cloud server should be included in the TCB. Normally, third party customers only get guest VM privilege (ring 0) while the `Monitor Module` and `Trust Module` have hypervisor privilege (ring -1). So a normal tenant has no capability to subvert the security functions provided by these two modules. To enhance the protection of *CloudMonatt* and defeat attacks (e.g., privilege escalation) caused by the vulnerabilities of the original virtualized system other than *CloudMonatt*, we can exploit some secure architectures. On the one hand, we can build *CloudMonatt* cloud servers upon security-aware systems which are designed and verified to eliminate potential vulnerabilities [134, 227, 108, 228]. On the other hand, we can leverage some architectures and methodologies to specifically protect the critical modules of *CloudMonatt*. For instance, we can use Bastion [61, 60] or Intel SGX [155, 32] to protect the `Monitor Module`. Bastion can protect the execution of trusted software modules within untrusted commodity software stacks. We can use a Bastion architecture as a cloud server and implement the `Monitor Kernel`, software `Monitor Tools`, and kernel modules/drivers of hardware `Monitor Tools` as the trusted software modules that need protection. Then Bastion places these modules into secure execution compartments isolated from the rest of the untrusted software stack, and enforces memory access control as well as storage protection. Bastion provides the secure inter-module control flow scheme to achieve secure communications between the `Monitor Kernel` and the `Monitor Tools`. Alternatively we can also use Intel SGX, and build isolated execution environments, i.e., enclaves, to protect the `Monitor`

`Module` from the untrusted software stacks. The `Trust Module` can also be protected by Bastion or SGX secure enclaves. Alternatively, since some hardware extensions are designed for the `Trust Module`, it can be placed into a separate chip. This achieves isolation between the `Trust Module` and the main CPU cores, and reduces the Trusted Computing Base.

In the Cloud Controller and the Attestation Server, all the critical modules should be well protected. To achieve this, we can use Bastion or Intel SGX to protect these software modules, as the cloud server. Besides, since these centralized servers are used to manage cloud services, we can provide additional protections to enhance their security. For example, we can disable scheduling guest VMs on these central servers. This eliminates the possibility that an attacker launches VMs on these servers, conducts privilege escalation attacks to get hypervisor-level privilege and compromises the software entities in the host OS or hypervisor. We can also establish firewalls on these servers to defeat network attacks. We can also provide redundancy in th Attestation Servers and Cloud Controllers to enhance the availability of these critical trusted servers. Since these central servers are just a small number of servers in the cloud system, it is feasible to have special security protections for them.

3.6 Chapter Summary

In this chapter, we introduce *CloudMonatt*, an architecture that enables secure monitoring and attestation of security features provided by a cloud server for the customer's VMs. First, we describe the design of *CloudMonatt* and show its key advances over prior work: (1) it is flexible and provides a rich set of security properties for VM attestation; (2) it bridges the semantic gap between the security properties a customer wants to request and the measurements collected from a cloud server; (3) it enables initialization as well as runtime attestation during the lifetime of the VM; (4) it defines

a novel periodic attestation capability during VM runtime; (5) it provides automated responses to bad attestation results to prevent potential, or further, security breaches; (6) it is protected by secure attestation protocols with a set of cryptographic keys that must be present or established; and (7) it is readily deployable: we leverage existing cloud mechanisms and well-honed security mechanisms where possible, identifying the minimal changes needed for a cloud system to implement our *CloudMonatt* architecture on the OpenStack cloud software.

Second, we conduct security verification of *CloudMonatt* to validate the security and correctness of this security-aware cloud architecture. To achieve scalable verification of this large distributed system, we split the verification task into an external part (considering the interactions between each server) and an internal part (considering the interactions and operations inside the server). We identify the security invariants and preconditions, model and verify each part using a cryptographic checking tool, *ProVerif*. This verification not only raises our confidence in the design, but also helps us understand which modules/servers are critical and guides us to further enhance the security of *CloudMonatt*. Future work could be designing new security mechanisms using secure architectures to realize the security preconditions we identified, and make *CloudMonatt* more secure.

We show simple cases that can be integrated in our *CloudMonatt*. In the next three chapters, we will show more complicated and interesting cloud-based detection and mitigation mechanisms to protect different security properties of customers' VMs. We will also describe how these mechanisms can be integrated into the *CloudMonatt* framework in each chapter (Sections 4.4.3, 5.4 and 6.5). In Section 7.1 and Figure 7.1, we will summarize these mechanisms and give a complete description of their integrations in the *CloudMonatt* framework.

Chapter 4

Detection and Mitigation of Availability Vulnerabilities

Memory DoS attacks are Denial of Service (or Degradation of Service) attacks caused by contention for hardware memory resources on a cloud server. Despite the strong memory isolation techniques for virtual machines (VMs) enforced by the software virtualization layer in cloud servers, the underlying hardware memory layers are still shared by the VMs and can be exploited by a clever attacker in a hostile VM co-located on the same server as the victim VM, denying the victim the working memory he needs.

In this chapter, we study the DoS attacks on the hardware memory resources (some parts of this chapter have been published in [265]). We first show quantitatively the *severity* of contention on different memory resources. We then show that a malicious cloud customer can mount low-cost attacks to cause severe performance degradation for a Hadoop distributed application, and 38× delay in response time for an E-commerce website in the Amazon EC2 cloud.

Then, we design an effective, new defense against these memory DoS attacks, using a statistical metric to detect their existence and *execution throttling* to mitigate

the attack damage. We achieve this by a novel re-purposing of existing *hardware performance counters* and *duty cycle modulation* for security, rather than for improving performance or power consumption. We implement a full prototype on the OpenStack cloud system. Our evaluations show that this defense system can effectively defeat memory DoS attacks with negligible performance overhead.

4.1 Background

To maximize resource utilization, cloud providers schedule virtual machines (VMs) leased by different tenants on the same physical machine, sharing the same hardware resources. While software isolation techniques, like VM virtualization, carefully isolate memory pages (virtual and physical), most of the underlying hardware memory hierarchy is still shared by all VMs running on the same physical machine in a multi-tenant cloud environment. Malicious VMs can exploit the multi-tenancy feature to intentionally cause severe contention on the shared memory resources to conduct Denial-of-Service (DoS) attacks against other VMs sharing the resources. Moreover, it has been shown that a malicious cloud customer can intentionally co-locate his VMs with victim VMs to run on the same physical machine [182, 226, 248]; this co-location attack can serve as a first step for performing memory DoS attacks against an arbitrary target.

The severity of memory resource contention has been seriously underestimated. While it is tempting to presume the level of interference caused by resource contention is modest, and in the worst case, the resulting performance degradation is isolated on one compute node, we show this is not the case. We present advanced attack techniques that, when exploited by malicious VMs, can induce much more intense memory contention than normal applications could do, and can degrade the performance of VMs on multiple nodes.

To demonstrate that our attacks work on real applications in real-world settings, we applied them to two case studies conducted in a commercial IaaS cloud, Amazon Elastic Compute Cloud (EC2). We show that even if the attacker only has *one* VM co-located with one of the many VMs of the target multi-node application, significant performance degradation can be caused to the entire application, rather than just to a single node. In our first case study, we show that when the adversary co-locates one VM with one node of a 20-node distributed Hadoop application, he may cause up to 3.7× slowdown of the entire distributed application. Our second case study shows that our attacks can slow down the response latency of an E-commerce application (consisting of load balancers, web servers, database servers and memory caching servers) by up to 38 times, and reduce the throughput of the servers down to 13%.

Despite the severity of the attacks, neither current cloud providers nor research literature offer any solutions to memory DoS attacks. Our communication with cloud providers suggests such issues are not currently addressed, in part because the attack techniques presented in this chapter are non-conventional, and existing solutions to network-based DDoS attacks do not help. Research studies have not explored defenses against adversarial memory contention either. As will be discussed in Section 4.1.3.2, existing solutions [77, 273, 251, 276, 29] only aim to enhance performance isolation between benign applications. Intentional memory abuses that are evident in memory DoS attacks are immune to these solutions.

Therefore, a large portion of this chapter is devoted to the design and implementation of a novel and effective approach to detect and mitigate all known types of memory DoS attacks with low overhead. Our detection strategy provides a generalized method for detecting deviations from the baseline behavior of the victim VM due to memory DoS attacks. We collect the baseline behaviors of the monitored VM at runtime, by creating a *pseudo isolated period*, without completely pausing co-tenant VMs. This provides periodic (re)establishment of baseline behaviors that adapt to

changes in program phases and workload characteristics. Once memory DoS attacks are detected, we show how malicious VMs can be identified and their attacks mitigated, using a novel form of selective execution throttling.

We implemented a prototype of our defense solution on the opensource OpenStack cloud software, and extensively evaluated its effectiveness and efficiency. Our evaluation shows that we can accurately detect memory DoS attacks and promptly and effectively mitigate the attacks. The performance overhead of persistent performance monitoring is lower than 5%, which is low enough to be used in production public clouds. Because our solution does not require modifications of CPU hardware, hypervisor or guest operating systems, it minimally impacts the existing cloud implementations. Therefore, we envision our solution can be rapidly deployed in public clouds as a new security service to customers who require higher security assurances (like in Security-on-Demand and *CloudMonatt* clouds [124, 262]).

4.1.1 Threat Model and Assumptions

We consider security threats from malicious tenants of public IaaS clouds. We assume the adversary has the ability to launch at least one VM on the cloud servers on which the victim VMs are running. Techniques required to do so have been studied [182, 226, 248], and are orthogonal to our work. The adversary can run any program inside his own VM. We do not assume that the adversary can send network packets to the victim directly, thus resource freeing attacks [224] or network-based DoS attacks [145] are not applicable. We do not consider attacks from the cloud providers, or any attacks requiring direct control of privileged software on the host server (e.g., hypervisor, host OS).

We assume the software and hardware isolation mechanisms function correctly as designed. A hypervisor virtualizes and manages the hardware resources (see Figure 4.1) so that each VM thinks it has the entire computer. A server can have multiple

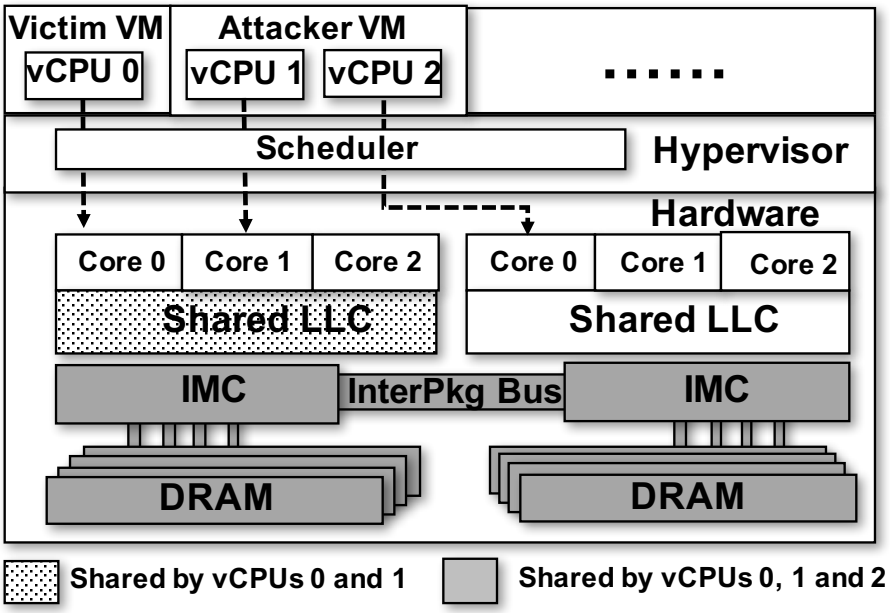


Figure 4.1: An attacker VM (with 2 vCPUs) and a victim VM share multiple layers of memory resources.

processor packages, where all processor cores in a package share a Last Level Cache (LLC), while L1 and L2 caches are private to a processor core and not shared by different cores. All processor packages share the Integrated Memory Controller (IMC), the inter-package bus and the main memory storage (DRAM chips). Each VM is designated a disjoint set of virtual CPUs (vCPU), which can be scheduled to operate on any physical cores based on the hypervisor's scheduling algorithms. A program running on a vCPU may use all the hardware resources available to the physical core it runs on. Hence, different VMs may simultaneously share the same hardware caches, buses, memory channels and DRAM bank buffers. We assume the cloud provider may schedule VMs from different customers on the same server (as co-tenants), but likely on different physical cores. As is the case today, software-based VM isolation by the hypervisor only isolates accesses to virtual and physical memory pages, but not to the underlying hardware memory resources shared by the physical cores.

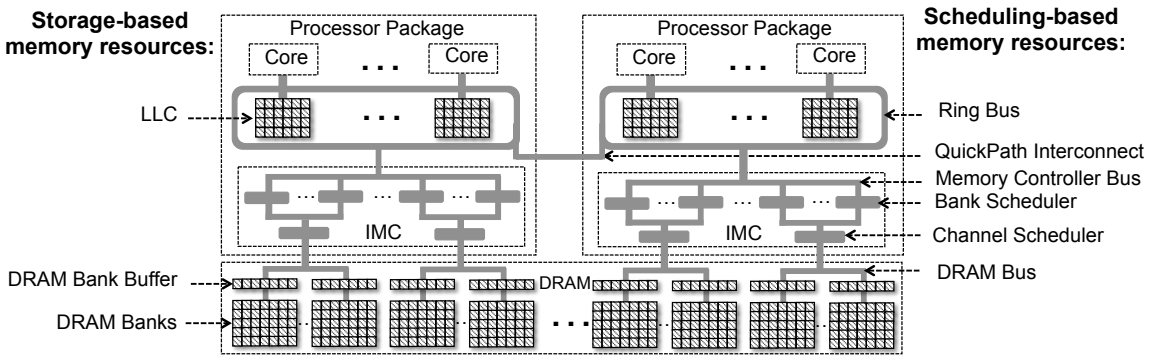


Figure 4.2: Shared storage-based and scheduling-based hardware memory resources in multi-core cloud servers.

4.1.2 Hardware Memory Resources

Figure 4.2 shows the hardware memory resources in modern computers. Using Intel processors as examples, modern X86-64 processors usually consist of multiple processor packages, each of which consists of several physical processor cores. Each physical core can execute one or two hardware threads in parallel with the support of Hyper-Threading Technology. A hierarchical memory subsystem, from the top to the bottom, is composed of different levels of *storage-based* components (e.g., caches, the DRAMs). These memory components are inter-connected by a variety of *scheduling-based* components (e.g., memory buses and controllers), with various schedulers arbitrating their communications. Memory resources shared by different cores are described below:

Last Level Caches (LLC). An LLC is shared by all cores in one package (older processors may have one package supported by multiple LLCs). Intel LLCs usually adopt an inclusive cache policy: every cache line maintained in the upper-level caches (i.e., core-private Level 1 and Level 2 caches in each core — not shown in Figure 4.2) also has a copy in the LLC. In other words, when a cache line in the LLC is evicted, so are the copies in the upper-level caches. A subsequent access to the memory block mapped to this cache line will result in an LLC miss, which will lead to the much

slower main memory access. On recent Intel processors (since Nehalem), LLCs are split into multiple slices, each of which is associated with one physical core, although every core may use the entire LLC. Intel employs static hash mapping algorithms to translate the physical address of a cache line to one of the LLC slices that contains the cache line. These mappings are unique for each processor model and are not released to the public. So it is harder for attackers to generate LLC contention by accessing a continuous memory buffer ([242]), since this buffer cannot exactly cover the entire target cache.

Memory Buses. Intel uses a ring bus topology to interconnect components in the processor package, e.g., processor cores, LLC slices, Integrated Memory Controllers (IMCs), QuickPath Interconnect (QPI) agents, etc. The high-speed QPI provides point-to-point interconnections between different processor packages, and between each processor package and I/O devices. The memory controller bus connects the LLC slices to the bank schedulers in the IMC, and the DRAM bus connects the IMC's schedulers to the DRAM banks. Current memory bus designs with high bandwidth make it very difficult for attackers to saturate the memory buses. Also, elimination of bus locking operations for normal atomic operations make bus locking attacks via normal atomic operations (e.g., [242]) less effective. However, some exotic atomic bus locking operations still exist.

DRAM banks. Each DRAM package consists of several banks, each of which can be thought of as a two dimensional data array with multiple rows and columns. Each bank has a bank buffer to hold the most recently used row to speed up DRAM accesses. A memory access to a DRAM bank may either be served in the bank buffer, which is a buffer-hit (fast), or in the bank itself, which is a buffer-miss (slow).

Integrated Memory Controllers (IMC). Each processor package contains one or multiple IMCs. The communications between an IMC and the portion of DRAM it controls are supported by multiple memory channels, each of which serves a set of

DRAM banks. When the processor wants to access the data in the DRAM, it first calculates the bank that stores the data based on the physical address, then it sends the memory request to the IMC that controls the bank. The processor can request data from the IMC in the local package, as well as in a different package via QPI. The IMCs implement a bank priority queue for each bank they serve, to buffer the memory requests to this bank. A bank scheduler is used to schedule requests in the bank priority queue, typically using a First-Ready-First-Come-First-Serve algorithm that gives high scheduling priority to the request that leads to a buffer-hit in the DRAM bank buffer, and then to the request that arrived earliest. Once requests are scheduled by the bank scheduler, a channel scheduler will further schedule them, among requests from other bank schedulers, to multiplex the requests onto a shared memory channel. The channel scheduler usually adopts a First-Come-First-Serve algorithm, which favors the earlier requests. Modern DRAM and IMCs can handle a large amount of requests concurrently, so it is less effective to flood the DRAM and IMCs to generate severe contention, as shown in [160].

4.1.3 Related Work

4.1.3.1 Resource Contention Attacks

Cloud DoS attacks. [145] proposed a DoS attack which can deplete the victim's network bandwidth from its subnet. [44] proposed a network-initiated DoS attack which causes contention in the shared Network Interface Controller. [117] proposed cascading performance attacks which exhaust hypervisor's I/O processing capability. [31] exploited VM migration to degrade the hypervisor's performance. Our work is different as we exploit failure of isolation in the hardware memory subsystem (which has not been addressed by cloud providers), and not attacks on networks or hypervisors.

Cloud resource stealing attacks. [224] proposed the resource-freeing attack, where a malicious VM can steal one type of resource from the co-located victim VM by increasing this VM's usage of other types of resources. [274] designed a CPU resource attack where an attacker VM can exploit the boost mechanism in the Xen credit scheduler to obtain more CPU resource than paid for. Our attacks do not steal extra cloud resources. Rather, we aim to induce the maximum performance degradation to the co-located victim VM targets.

Hardware resource contention studies. [103] studied the effect of trace cache evictions on the victim's execution with Hyper-Threading enabled in an Intel Pentium 4 Xeon processor. [242] explored frequently flushing shared L2 caches on multicore platforms to slow down a victim program. They studied saturation and locking of buses that connect L1/L2 caches and the main memory [242]. [160] studied contention attacks on the schedulers of memory controllers. However, due to advances in computer hardware design, caches and DRAMs are larger and their management policies more sophisticated, so these prior attacks may not work in modern cloud settings.

Timing channels in clouds. Prior studies showed that shared memory resources can be exploited by an attacker to extract crypto keys from the victim VM using cache side-channel attacks in cloud settings [269, 270, 144], or to transmit information, using cache operations [182, 247] or bus activities [244] in covert channel communications between two VMs. Unlike side-channel attacks, our memory DoS attacks aim to *maximize* the effects of resource contention, while resource contention is an unintended side-effect of side-channel attacks. To maximize contention, we addressed various new challenges, e.g., finding which attacks cause greatest resource contention (e.g., exotic bus locking versus memory controller attacks), maximizing the frequency of resource depletion, and minimizing self-contention. To the best of our knowledge, we are the first to show that similar attack strategies (enhanced for resource contention) can be used as availability attacks as well as confidentiality attacks.

4.1.3.2 Eliminating Resource Contention

VM performance monitoring. Public clouds offer performance monitoring services for customers' VMs and applications, e.g., Amazon CloudWatch [2], Microsoft Azure Application Insights [16], Google Stackdriver [23], etc. However, these services only monitor CPU usage, network traffic and disk bandwidth, but not low-level memory usage. To verify if a VM's performance is affected by co-located VMs, we need to measure the VM's performance with and without contention. To measure a VM's performance without contention for reference sampling, past work offer three ways: (1) collecting the VM's performance characteristics before it is deployed in the cloud [77, 273]; (2) measuring the performance of other VMs which run similar tasks [267, 164]; (3) measuring the PROTECTED VM while pausing all other co-located VMs [110, 251]. The drawback of (1) and (2) is that it only works for programs with predictable and stable performance characteristics, and does not support arbitrary programs running in the PROTECTED VM. The problem with (3) is the significant performance overhead inflicted on co-located VMs. In contrast, we use novel *execution throttling* of the co-located VMs to collect the PROTECTED VM's baseline (reference) measurements with negligible performance overhead (Section 4.4.1). While *execution throttling* has been used to achieve resource fairness in prior work [266, 86]; using it to collect reference samples at runtime is, to our knowledge, novel.

QoS-aware VM scheduling. Prior research propose to predict interference between different applications (or VMs) by profiling their resource usage offline and then statically scheduling them to different servers if co-locating them will lead to excessive resource contention [77, 273, 251]. The underlying assumption is that applications (or VMs), when deployed on the cloud servers, will not change their resource usage patterns. Unfortunately, these approaches fall short in defense against malicious

applications, who can reduce their resource uses during the profiling stage, then run memory DoS attacks when deployed, thus evading these QoS scheduling mechanisms.

Load-triggered VM migration. Some studies propose to monitor the resource consumption of guest VMs or the entire server in real-time, and migrate VMs to different processor packages or servers when there is severe resource contention [50, 276, 29, 232]. By doing so these approaches can dynamically balance the workload among multiple packages or servers when some of them are overloaded, and achieve an optimal resource allocation. While they work well for performance optimization of a set of fully-loaded servers, they fail to detect carefully-crafted memory DoS attacks. First, the metrics in their methods cannot be used to detect the existence of memory DoS attacks. These works measure the LLC miss rate [50, 276] or memory bandwidth [29, 232] of guest VMs or the whole server. A high LLC miss rate or memory bandwidth indicates severe resource contention for the VMs or servers. However, a memory DoS attack does not need to cause high LLC miss rate or memory bandwidth in order to degrade a victim's performance. For instance, atomic locking attacks (Section 4.2.3) lock the bus temporarily but frequently, which could incur decreased LLC accesses and LLC misses in the victim VM. Our experiments show the victim VM's LLC miss rate does not change, and its memory bandwidth is even decreased. So such micro-architectural measurement can never reveal severe resource contention, or trigger VM migration in the above approaches. Second, these approaches aim to balance the system's performance. So they cannot guarantee to choose the victim or attacker VMs for migration when they achieve optimal workload placement. For instance, Adaptive LLC cleansing attacks (Section 4.2.2) can increase the victim VMs' LLC miss rate. However, the above approaches have no means to figure out that the victim VM has the strongest desire for migration. It is possible that there exists another VM, which has even higher LLC miss rate than the victim VM, due to its internal execution behaviors, not the interaction and contention with the attacker. So

the above approaches will migrate this VM instead of the victim VM, as this VM has the highest miss rate. Then the victim VM will still suffer LLC cleansing attacks.

Performance isolation. While cloud providers can offer single-tenant machines to customers with high demand for security and performance, disallowing resource sharing by VMs will lead to low resource utilization and thus is at odds with the cloud business model. Another option is to partition memory resources to enforce performance isolation on shared resources (e.g., LLC [238, 188, 131, 73, 130, 8], or DRAM [160, 162, 236]). These works aim to achieve fairness between different domains and provide fair QoS. However, they cannot effectively defeat memory DoS attacks. For cache partitioning, software page coloring methods [131] can cause significant wastage of LLC space, while hardware cache partitioning mechanisms have insufficient partitions (e.g., Intel Cache Allocation Technology [8] only provides four QoS partitions on the LLC). Furthermore, LLC cache partitioning methods cannot resolve atomic locking attacks (Section 4.2.3).

To summarize, existing solutions fail to address memory DoS attacks because they assume benign applications with non-malicious behaviors. Also, they are often tailored to only one type of attack so that they cannot be generalized to all memory DoS attacks, unlike our proposed defense.

4.2 Memory DoS Attacks

4.2.1 Fundamental Attack Strategies

We have classified all memory resources into either storage-based or scheduling-based resources. This helps us formulate the following two fundamental attack strategies for memory DoS attacks:

- **Storage-based contention attack.** The fundamental attack strategy to cause contention on storage-based resources is to *reduce the probability that the victim's data is found in an upper-level memory resource (faster), thus forcing it to fetch the data from a lower-level resource (slower).*
- **Scheduling-based contention attack.** The fundamental attack strategy on a scheduling-based resource is to *decrease the probability that the victim's requests are selected by the scheduler, e.g., by locking the scheduling-based resources temporarily, tricking the scheduler to improve the priority of the attacker's requests, or overwhelming the scheduler by submitting a huge amount of requests simultaneously.*

We systematically show how memory DoS attacks can be constructed on different layers of memory resources (LLC in Section 4.2.2, bus in Section 4.2.3, memory controller and DRAM in Section 4.2.4). For each memory component, we first study the basic techniques the attacker can use to generate resource contention and affect the victim's performance. We measure the effectiveness of the attack techniques on the victim VM with different vCPU locations and program features. Then we propose some practical attacks and evaluate their impacts on real-world benchmarks.

Testbed configuration. To demonstrate the severity of different types of memory DoS attacks, we use a server configuration, representative of many cloud servers, configured as shown in Table 4.1. We use Intel processors, since they are the most common in cloud servers, but the attack methods we propose are general, and applicable to other processors and platforms as well.

In each of the following experiments, we launched two VMs, one as the attacker and the other as the victim. By default, each VM was assigned a single vCPU. We select a mix of benchmarks for the victim: (1) We use a modified *stream* program [153, 160] to explore the effectiveness of the attacks on victims with different features. This program allocates two array buffers with the same size, one as the source and the other as the destination. It copies data from the source to the destination in loops

Table 4.1: Testbed Configuration

Server	Dell PowerEdge R720
Processor Packages	Two 2.9GHz Intel Xeon E5-2667 (Sandy Bridge)
Cores per Package	6 physical cores, or 12 hardware threads with Hyper-Threading
Core-private Level 1 and Level 2 caches	L1 I and L1 D: each 32KB, 8-way set-associative; L2 cache: 256KB, 8-way set-associative
Last Level Cache (LLC)	15MB, 20-way set-associative, shared by cores in package, divided into 6 slices of 2.5MB each; one slice per core
Physical memory	Eight 8GB DRAMs, divided into 8 channels, and 1024 banks
Hypervisor	Xen version 4.1.0
VM's OS	Ubuntu 12.04 Linux, with 3.13 kernel

repeatedly, either in a sequential manner (resulting a program with *high memory locality*) or in a random manner (*low memory locality*). We chose this benchmark because it is memory-intensive and allows us to alter the size of memory footprints and the locality of memory resources. (2) To fully evaluate the attack effects on real-world applications, we choose 8 macro benchmarks (6 from SPEC2006 [22] and 2 from PARSEC [48]) and cryptographic applications based on OpenSSL as the victim program. Each experiment was repeated 10 times, and the mean values and standard deviations are reported.

4.2.2 Cache Contention (Storage Resources)

Of the storage-based contention attacks, we found that the LLC contention results in the most severe performance degradation. The root vulnerability is that an LLC is shared by all cores of the same CPU package, without access control or quota enforcement. Therefore a program in one VM can evict LLC cache lines belonging to another VM. Moreover, inclusive LLCs (e.g., most modern Intel LLCs) will propagate these cache line evictions to core-private L1 and L2 caches, further aggravating the interference between programs (or VMs) in CPU caches.

4.2.2.1 Contention Study

Cache cleansing. To cause LLC contention, the adversary can allocate a memory buffer to cover the entire LLC. By accessing one memory address per memory block in the buffer, the adversary can cleanse the entire cache and evict all of the victim's data from the LLC to the DRAM.

The optimal buffer used by the attacker should *exactly map* to the LLC, which means it can fill up each cache set in each LLC slice without *self-conflicts* (i.e., evicting earlier lines loaded from this buffer). For example, for a LLC with n^s slices, n^c sets in each slice, and n^w-way set-associativity, the attacker would like $n^s \times n^c \times n^w$ memory blocks to cover all cache lines of all sets in all slices. There are two challenges that make this task difficult for the attacker: the host physical addresses of the buffer to index the cache slice are unknown to the attacker, and the mapping from physical memory addresses to LLC slices is not publicly known.

Mapping LLC cache slices: To overcome these challenges, the attacker can first allocate a 1GB Hugepage which is guaranteed to have continuous host physical addresses; thus he need not worry about virtual to physical page translations which he does not know. Then for each LLC cache set $\mathrm{S^i}$ in all slices, the attacker sets up an empty group $\mathbb{G}^i$, and starts the following loop: (i) select block $\mathrm{A^k}$ from the Hugepage, which is mapped to set $\mathrm{S^i}$ by the same index bits in the memory address; (ii) add $\mathrm{A^k}$ to $\mathbb{G}^i$; (iii) access all the blocks in $\mathbb{G}^i$; and (iv) measure the average access latency per block. A longer latency indicates block $\mathrm{A^k}$ causes self-conflict with other blocks in $\mathbb{G}^i$, so it is removed from $\mathbb{G}^i$. The above loop is repeated until there are $n^s \times n^w$ blocks in $\mathbb{G}^i$, which can exactly fill up set $\mathrm{S^i}$ in all slices. Next the attacker needs to distinguish which blocks in $\mathbb{G}^i$ belong to each slice: He first selects a new block $\mathrm{A^n}$ mapped to set $\mathrm{S^i}$ from the Hugepage, and adds it to $\mathbb{G}^i$. This should cause a self-conflict. Then he executes the following loop: (i) select one block $\mathrm{A^m}$ from $\mathbb{G}^i$; (ii) remove it from $\mathbb{G}^i$;

(iii) access all the blocks in $\mathbb{G}^i$; and (iv) measure the average access latency. A short latency indicates block A^m can eliminate the self-conflict caused by A^n, so it belongs to the same slice as A^n. The attacker keeps doing this until he discovers n^w blocks that belong to the same slice as A^n. These blocks form the group that can fill up set S^i in one slice. The above procedure is repeated till the blocks in $\mathbb{G}^i$ are divided into n^s slices for set S^i. After conducting the above process for each cache set, the attacker obtains a memory buffer with non-consecutive blocks that map exactly to the LLC. Such self-conflict elimination is also useful in improving side-channel attacks [144].

To test the effectiveness of cache cleansing, we arranged the attacker VM and victim VM on the same processor package, thus sharing the LLC and all memory resources in lower layers. The adversary first identified the memory buffer that maps to the LLC. Then he cleansed the whole LLC repeatedly. The resulting performance degradation of the victim application is shown in Figure 4.3. The victim suffered from the most significant performance degradation when the victim's buffer size is around 10MB (1.8× slowdown for the high locality program, and 5.5× slowdown for the low locality program). When the buffer size is smaller than 5MB (data are stored mainly in upper-level caches), or the size is larger than 25MB (data are stored mainly in the DRAM), the impact of cache contention on LLC is negligible.

The results can be explained as follows: the maximum performance degradation can be achieved on victims with memory footprint smaller than, but close to, the LLC size, which is 15MB, because the victim suffers the least from self conflicts in LLC and the most from the attacker's LLC cleansing. Moreover, as a low locality program accesses its data in a random order, hardware prefetching is less effective in enhancing the program's access speed. So the program accesses the cache at a relatively lower rate. Its data will be evicted out of the LLC by the attacker with higher probability. That is why the LLC cleansing has a larger impact on low locality programs than on high locality programs.

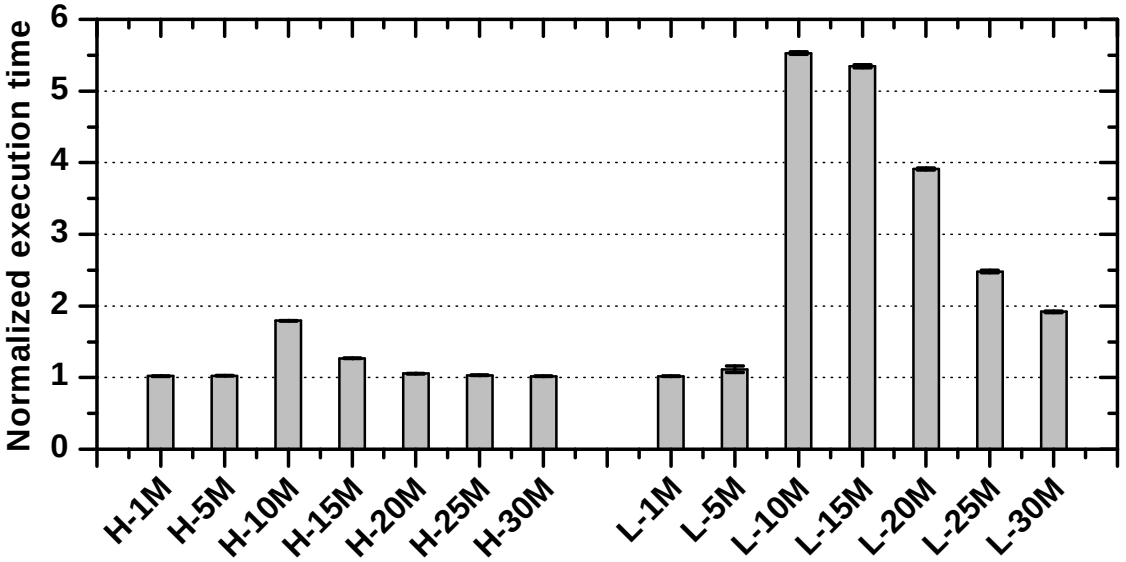


Figure 4.3: Performance slowdown due to LLC cleansing contention. We use "H-x" or "L-x" to denote the victim program has high or low memory locality and has a buffer size of x.

Takeaways. LLC contention is (more) effective when (1) the attacker and victim VMs share the same LLC, (2) the victim program's memory footprint is about the size of LLC, and (3) the victim program has lower memory locality.

4.2.2.2 Practical Attack Evaluation

We improve this attack by increasing the cleansing speed, and the accuracy of evicting (thus contending with) the victim's data.

Multi-threaded LLC cleansing. To speed up the LLC cleansing, the adversary may split the cleansing task into n threads, with each running on a separate vCPU and cleansing only a non-overlapping $1/n$ of the LLC simultaneously. This effectively increases the cleansing speed by n times.

In our experiment, the attacker VM and the victim VM were arranged to share the LLC. The attacker VM was assigned 4 vCPUs. It first prepared the memory buffer that exactly mapped to the LLC. Then he cleansed the LLC with (1) one vCPU; (2) 4 vCPUs (each cleansing 1/4 of the LLC). Figure 4.4 shows that the attack can cause

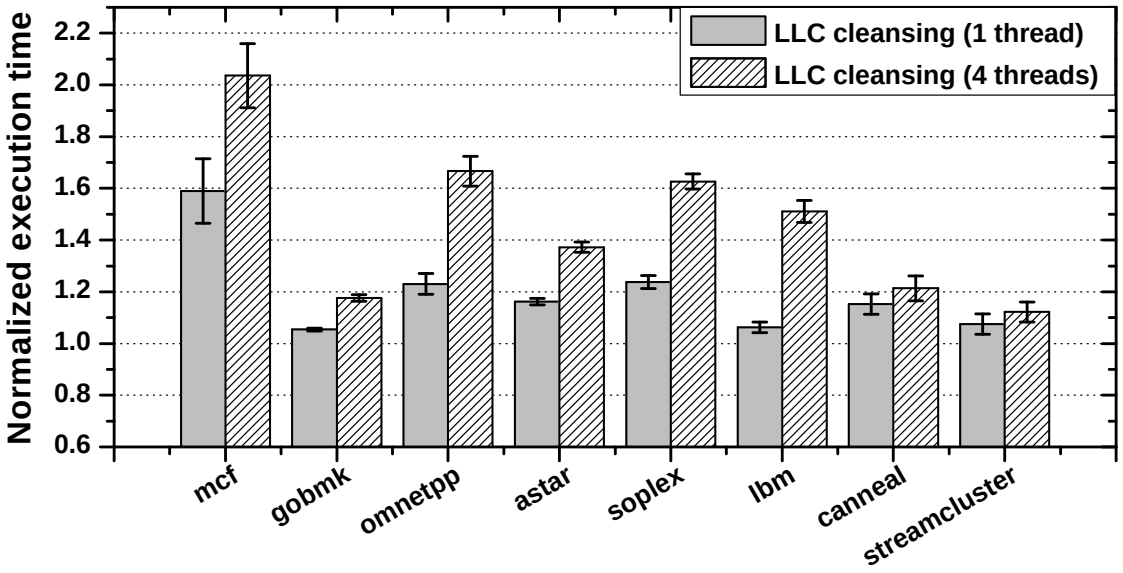


Figure 4.4: Performance slowdown due to multi-threaded LLC cleansing attack

$1.05 \sim 1.6\times$ slowdown to the victim VM when using one thread, and $1.12 \sim 2.03\times$ slowdown when using four threads.

Adaptive LLC cleansing. The basic LLC cache cleansing technique does not work when the victim's program has a memory footprint (<1MB) that is much smaller than an LLC (e.g., 15MB), since it takes a long time to finish one complete LLC cleansing, where most of the memory accesses do not induce contention with the victim. To achieve finer-grained attacks, we developed a cache probing technique to pinpoint the cache sets in the LLC that map to the victim's memory footprint, and cleanse only these selected sets.

The attacker first allocates a memory buffer covering the entire LLC in his own VM. Then he conducts cache probing in two steps: (1) In the Discover Stage, while the victim program runs, for each cache set, the attacker accesses some cache lines belonging to this set and figures out the maximum number of cache lines which can be accessed without causing cache conflicts. If this number is smaller than the set associativity, this cache set will be selected to conduct adaptive cleansing attacks, because the victim has frequently occupied some cache lines in this set; (2) In the

Algorithm 4.1: Adaptive LLC cleansing

```
   Input:
1      cache_set{}: all the sets in the LLC
2      cache_buffer{}: cover the entire LLC
3      cache_assoc_num: the associativity of LLC
4  begin
5      /* Discover Stage */
6      victim_set=∅
7      for each set i in cache_set{} do
8          Find out j, s.t., accessing j cache lines in set i from cache_buffer{} has no cache conflict (low
             accessing time), but accessing j+1 cache lines in set i from cache_buffer{} has cache conflict (high
             accessing time)
9          if j<cache_assoc_num then
10             add i to victim_set{}
11         end
12     end

13     /* Attack Stage: */
14     while attack is not finished do
15         for each set i in victim_set{} do
16             access cache_assoc_num cache lines in set i from cache_buffer{}
17         end
18     end
19 end
```

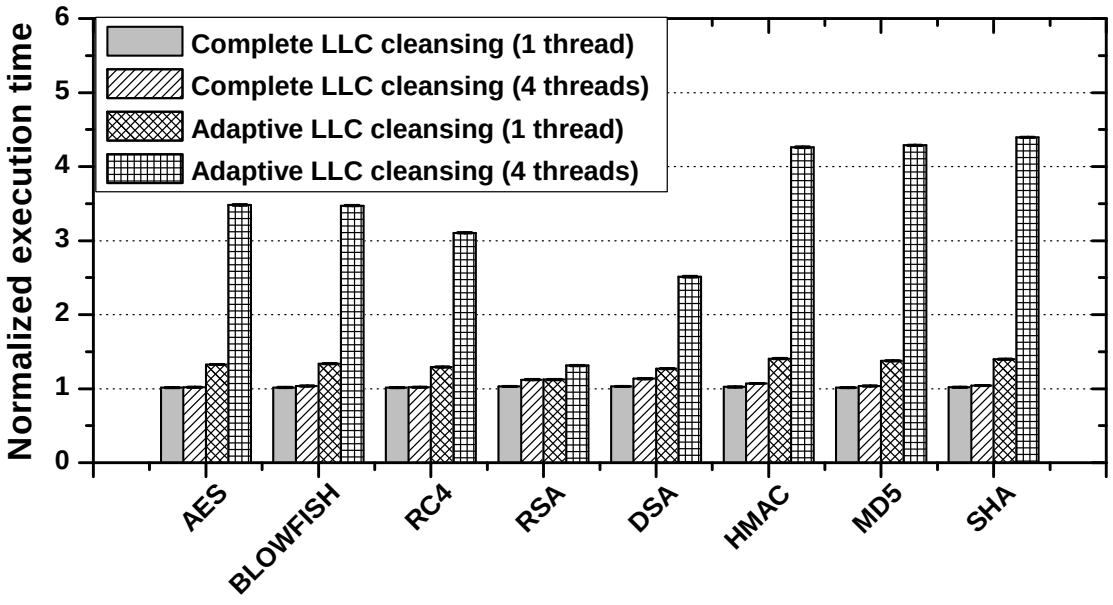


Figure 4.5: Performance slowdown due to adaptive LLC cleansing attacks

Attack Stage, the attacker keeps accessing these selected cache sets to cleanse the victim's data. Algorithm 4.1 shows the steps to perform the adaptive LLC cleansing.

Figure 4.5 shows the results of the attacker's multi-threaded adaptive cleansing attacks against victim applications with cryptographic operations. While the basic

cleansing did not have any effect, the adaptive attacks can achieve around 1.12 to 1.4 times runtime slowdown with 1 vCPU, and up to 4.4× slowdown with 4 vCPUs.

4.2.3 Bus Contention (Scheduling Resources)

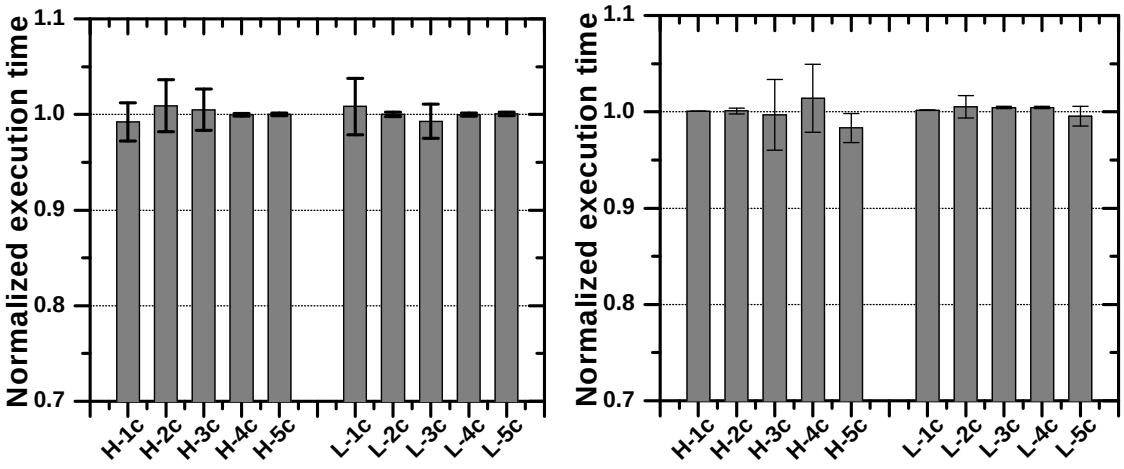
The availability of internal memory buses can be compromised by overwhelming or temporarily locking down the buses. We study the effects of these techniques.

4.2.3.1 Contention Study

Bus saturation. One intuitive approach for an adversary is to create numerous memory requests to saturate the buses [242]. However, the bus bandwidth in modern processors may be too high for a single VM to saturate.

To examine the effectiveness of *bus saturation contention*, we conducted two sets of experiments. In the first set of experiments, the victim VM and the attacker VM were located in the same processor package but on different physical cores (Figure 4.6a). They accessed different parts of the LLC, without touching the DRAM. Therefore the attacker VM causes contention in the ring bus that connects LLC slices without causing contention in the LLC itself. In the second set of experiments, the victim VM and the attacker VM were pinned on different processor packages (Figure 4.6b). They accessed different memory channels, without inducing contention in the memory controller and DRAM modules. Therefore the attacker and victim VMs only contend in buses that connect LLCs and IMCs, as well as the QPI buses. The attacker and victim were assigned increasing number of vCPUs to cause more bus traffic. Results in Figure 4.6 show that these buses were hardly saturated and the impact on the victim's performance was negligible in all cases.

Bus locking. To deny the victim from being scheduled by a scheduling resource, the adversary can temporarily lock down the internal memory buses. Intel processors provide locked atomic operations for managing shared data structures between multi-



(a) Same package (b) Different packages

Figure 4.6: Performance slowdown due to bus saturation contention. We use "H-xc" or "L-xc" to denote the configuration that the victim program has high or low locality, and both of the attacker and victim use x cores to contend for the bus.

processors [9]. Before Intel Pentium (P5) processors, the locked atomic operations always generate LOCK signals on the internal buses to achieve operation atomicity. So other memory accesses are blocked until the locked atomic operation is completed. For processor families after P6, the bus lock is transformed into a cache lock: the cache line is locked instead of the bus and the cache coherency mechanism is used to ensure operation atomicity. This causes much smaller scheduling lockdown times.

However, we have found two exotic atomic operations the adversary can still use to lock the internal memory buses: (1) *Locked atomic accesses to unaligned memory blocks*: the processor has to fetch two adjacent cache lines to complete this unaligned memory access. To guarantee the atomicity of accessing the two adjacent cache lines, the processors will flush in-flight memory accesses issued before, and block memory accesses to the bus, until the unaligned memory access is finished. (2) *Locked atomic accesses to uncacheable memory blocks*: when uncached memory pages are accessed in atomic operations, the cache coherency mechanism does not work. Hence, the memory bus must be locked to guarantee atomicity. Listings 4.1 and 4.2 show the codes for

Listing 4.1: Attack using unaligned atomic operations

```
1  char *buffer = mmap(0, BUFFER_SIZE, PROT_READ|PROT_WRITE,
       MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
2
3  int x = 0x0;
4  int *block_addr = (int *)(buffer+CACHE_LINE_SIZE-1);
5  while (1) {
6    __asm__(
7      "lock; xaddl %%eax, %1\n\t"
8      :"=a"(x)
9      :"m"(*block_addr), "a"(x)
10     :"memory");
11 }
```

Listing 4.2: Attack using uncached atomic operations

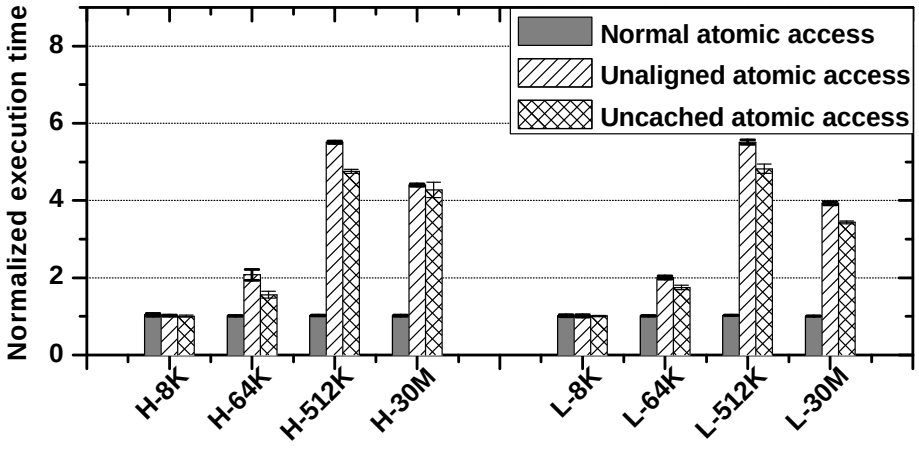
```
1  char *buffer = mmap(0, BUFFER_SIZE, PROT_READ|PROT_WRITE,
       MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
2  syscall(__NR_UnCached, (unsigned long)buffer);
3  int x = 0x0;
4  int *block_addr = (int *)buffer;
5  while (1) {
6    __asm__(
7    "lock; xaddl %%eax, %1\n\t"
8    :"=a"(x)
9    :"m"(*block_addr), "a"(x)
10   :"memory");
11 }
```

issuing unaligned and uncached atomic operations. The two programs keep conducting the addition operation of a constant (`x`) and a memory block (`block_addr`) (line 5 – 11): in line 7, the `lock` prefix indicates this operation is atomic. The instruction `xaddl` indicates this is an addition operation. The first operand is the register `eax`, which stores `x` (line 9). The second operand is the first parameter of line 9 (data denoted by the address `block_addr`). The results will be loaded to the register `eax` (line 8). In Listing 4.1, we set this memory block as unaligned (line 4). In Listing 4.2, we added a new system call to set the page table entries of the memory buffer as cache disabled (line 2).

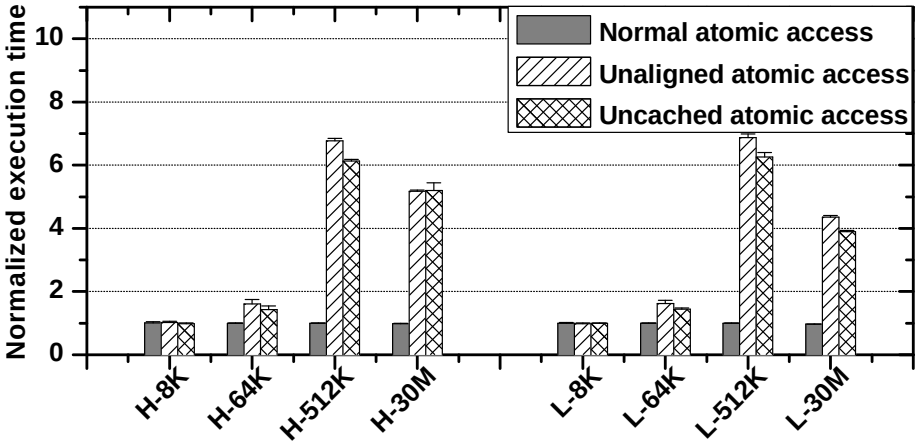
To evaluate the effects of *bus locking contention*, we chose the footprint size of the victim program as (1) 8KB, with which the L1 cache was under-utilized, (2) 64KB, with which the L1 cache was over-utilized but the L2 cache was under-utilized, (3) 512KB, with which the L2 cache was over-utilized but the LLC was under-utilized, and (4) 30MB, with which the LLC was over-utilized. The attacker VM kept issuing unaligned atomic or uncached atomic memory accesses to lock the memory buses. For comparison, we also run another group of experiments, where the attacker kept issuing normal locked memory accesses. We considered two scenarios: (1) the attacker and victim shared the same processor package, but run on different cores; (2) they were scheduled on different processor packages. The normalized execution time of the victim program is shown in Figure 4.7.

We observe that the victim's performance was significantly affected when the its buffer size was larger than the L2 caches. This is because the attacker who kept requesting exotic atomic memory accesses was only able to lock the buses within its physical cores, the ring buses around the LLCs in each package, the QPI, and the buses from each package to the DRAM. So when the victim's buffer size was smaller than the L2 cache, it fetched data from the private caches in its own core without being affected by the attacker. However, when the victim's buffer size was larger than the L2 caches, its access to the LLC would be delayed by the bus locking operations, and the performance is degraded (up to 6× slowdown for high locality victim programs and 7× slowdown for low locality victim programs).

Takeaways. We explored two approaches to bus contention. Saturating internal buses is unlikely to cause noticeable performance degradation. Bus locking shows promise when the victim program makes heavy use of the shared LLC or lower layer memory resources, whenever the victim VM and attacker VM are on the same processor package or different packages.



(a) Same package



(b) Different packages

Figure 4.7: Performance slowdown due to bus locking contention. We use "H-x" or "L-x" to denote the victim program has high or low memory locality and has a buffer size of x.

4.2.3.2 Practical Attack Evaluation

To evaluate the effectiveness of *atomic locking attacks* on real-world applications, we scheduled the attacker VM and victim VM on different processor packages. The attacker VM kept generating atomic locking signals by (1) requesting unaligned atomic memory accesses, or (2) requesting uncached atomic memory accesses. The normalized execution time of the victim program is shown in Figure 4.8. We observe that the

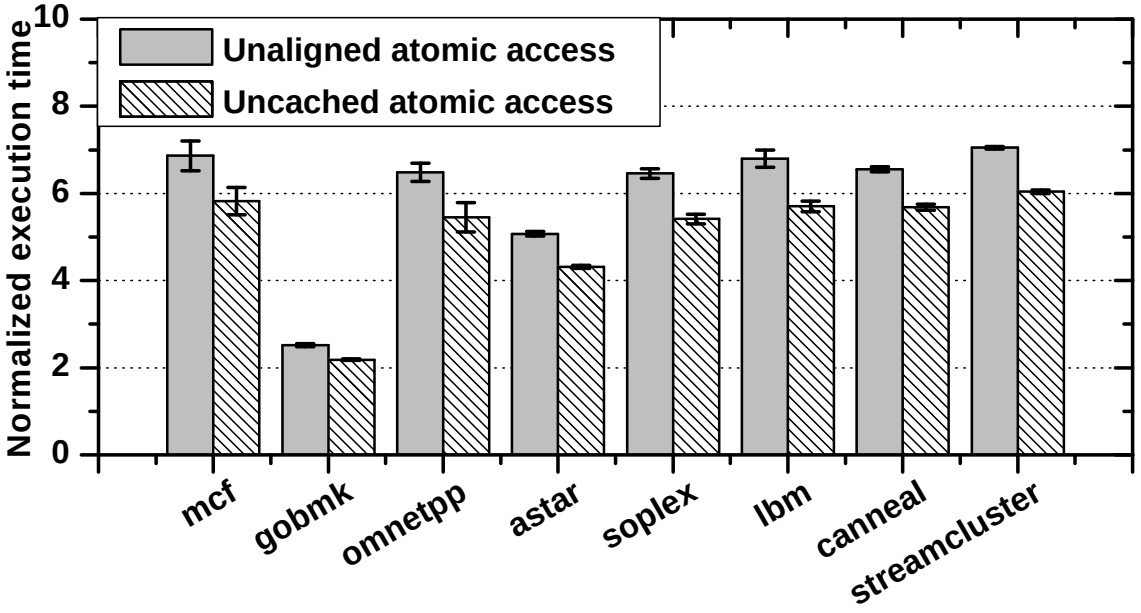


Figure 4.8: Performance slowdown due to bus locking attacks.

victim's performance can be degraded as much as 7 times when the attacker conducted exotic atomic operations.

4.2.4 Memory Contention (Combined Resources)

An IMC uses the bank scheduler and channel scheduler to select the memory requests for each DRAM access. Therefore an adversary may contend on these two schedulers by frequently issuing memory requests that result in bank buffer hits to boost his priority in the scheduler. Moreover, each memory bank is equipped with only one bank buffer to hold the recently used bank row, so the adversary can easily induce storage-based contention on bank buffers by frequently occupying them with his own data.

4.2.4.1 Contention Study

Memory flooding. Since channel and bank schedulers use First-Come-First-Serve policies, an attacker can send a large amount of memory requests to flood the target

memory channels or DRAM banks. These requests will contend on the scheduling-based resources with the victim's memory requests. In addition, the attacker can issue requests in sequential order, so sequential accesses will hit the same row in DRAM banks. This can achieve high row-hit locality and thus high priority in the bank scheduler, to further increase the effect of flooding. Furthermore, when the adversary keeps flooding the IMCs, these memory requests can also evict the victim's data out of the DRAM bank buffers. The victim's bank buffer hit rate is decreased and its performance is further degraded.

To demonstrate the effects of DRAM contention, we configure one attacker VM to operate a memory flooding program, which kept accessing memory blocks in the same DRAM bank directly without going through caches (i.e., uncached accesses). The victim VM did exactly the same with either high or low memory locality. We conducted two sets of experiments: (1) The two VMs access the same bank in the same channel (Same bank in Figure 4.9); (2) the two VMs access two different banks in the same channel (Same channel in Figure 4.9). To alter the memory request rate issued by the two VMs, we also changed the number of vCPUs in the attacker and victim VMs. The normalized execution time of the victim program is shown in Figure 4.9.

Three types of contention were observed in these experiments. First, channel scheduling contention was observed when the attacker and the victim access different banks in the same channel. It was enhanced with increased number of attacker and victim vCPUs, thus increasing the memory request rate (around 1.2× slowdown for "H-5c" and "L-5c"). Second, bank scheduling contention was also observed when the attacker and victim accessed the same DRAM bank. When the memory request rate was increased, the victim's performance was further degraded by an additional 70% and 25% for "H-5c" and "L-5c", respectively. Third, contention in DRAM bank buffers was observed when we compare the results of "Same bank" in Figure 4.9 between high

locality and low locality victim programs — low locality victims already suffer from row-misses and the additional performance degradation in high locality victims is due to bank buffer contention (1.9× slowdown for H_5c verses 1.45× slowdown for L_5c).

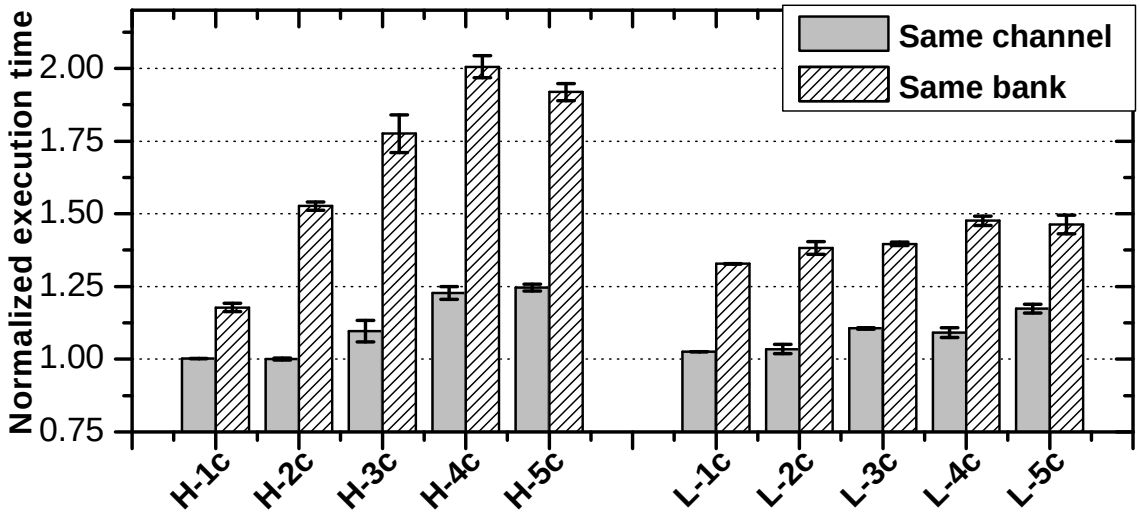


Figure 4.9: Performance slowdown due to memory channel and bank contention. We use "H-xc" or "L-xc" to denote the configuration that the victim program has high or low locality, and both of the attacker and victim use x cores to contend for the bus.

We consider the overall effect of memory flooding contention. In this experiment, the victim VM runs a high locality or low locality stream benchmark on its only vCPU. The attacker VM allocates a memory buffer with the size 20× that of the LLC and runs a stream program which keeps accessing memory blocks sequentially in this buffer to generate contention in every channel and every bank. To increase bus traffic, the attacker employed multiple vCPUs to perform the attack simultaneously. The performance degradation, as we can see in Figure 4.10, was significant when the victim's memory accesses footprint was mostly in the DRAM, and more vCPUs of the attacker VM were used in the attack. The attacker can use 8 vCPUs to induce about 1.5× slowdown to the victim with the buffer size larger than LLC.

Takeaways. Contention can be induced in channel schedulers, bank schedulers and bank buffers between different programs from different processor packages. This

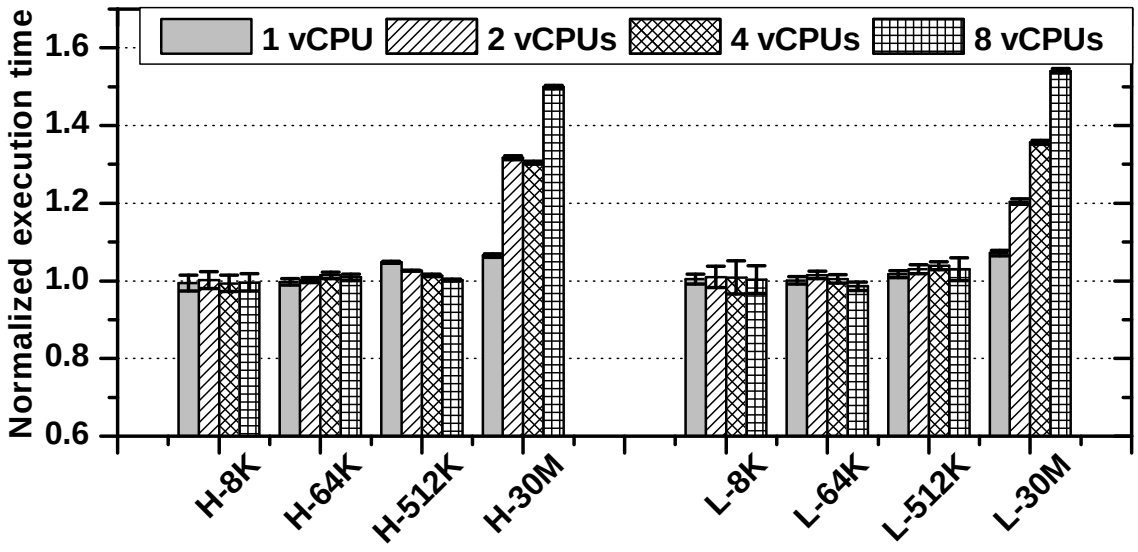


Figure 4.10: Performance slowdown due to memory flooding contention. We use "H-x" or "L-x" to denote the victim program has high or low memory locality and has a buffer size of x.

contention is especially significant when the victim program's memory footprint is larger than the LLC.

4.2.4.2 Practical Attack Evaluation

We evaluate two advanced memory flooding attacks.

Multi-threaded memory flooding. The attacker can use more threads to increase the memory flooding speed, as we demonstrated earlier. We evaluated this attack on real-world applications. The attacker and victim VMs are located in two different processor packages, so they only share the IMCs and DRAM. The attacker VM issues frequent, highly localized memory requests to flood every DRAM bank and every channel. To increase bus traffic, the attacker employed 8 vCPUs to perform the attack simultaneously. Figure 4.11 shows that the victim experiences up to a 1.22× runtime slowdown when the attacker uses 8 vCPUs to generate contention (Complete Memory Flooding bars).

Adaptive memory flooding. For a software program with smaller memory footprint, only a few memory channels will be involved in its memory accesses. We developed a *novel* approach with which an adversary may identify memory channels that are more frequently used by a victim program. To achieve this, the attacker needs to reverse engineer the unrevealed algorithms that map the physical memory addresses to memory banks and channels, in order to accurately direct the flows of the memory request flood.

Mapping DRAM banks and channels: The attacker can leverage methods due to Liu et al. [147] to identify the bits in physical memory addresses that index the DRAM banks. The attacker first allocates a 1GB Hugepage with continuous physical addresses, which avoids the unknown translations from guest virtual addresses to machine physical addresses. Then he selects two memory blocks from the Hugepage whose physical addresses differ in only one bit. He then flushes these two blocks out of caches and accesses them from the DRAM alternatively. A low latency indicates these two memory blocks are served in two banks as there is no contention on bank buffers. In this way, the attacker is able to identify all the bank bits. Next, the attacker needs to identify the channel bits among the bank bits. We design a *novel* algorithm which is shown in Algorithm 4.2 to achieve this goal. The attacker selects two groups of memory blocks from the Hugepage, whose bank indexes differ in only one bit. The attacker then allocates two threads to access the two groups simultaneously. If the different bank index bit is also a channel index bit, then the two groups will be in two different channels, and a shorter access time will be observed since there is no channel contention.

Then the attacker performs two stages: in the Discover Stage, the attacker keeps accessing each memory channel for a number of times and measures his own memory access time to infer contention from the victim program. By identifying the channels with a longer access time, the attacker can detect which channels are heavily

Algorithm 4.2: Discovering channel index bits

```
   Input:
1      bank_bit{} // bank index bits
2      memory_buffer{} // a memory buffer
   Output:
3      channel_bit{}
4  begin
5      channel_bit{}=∅
6      for each bit i ∈ bank_bit{} do
7          buffer_A{}=memory_buffer{}
8          buffer_B{}=memory_buffer{}
9          for each memory block d_a ∈ buffer_A{} do
10             m_a = physical address of d_a
11             if (m_a's bit i) ≠ 0 then
12                 delete d_a from buffer_A{}
13                 break
14             end
15             for each bit j ∈ bank_bit{} and i ≠ j do
16                 if (m_a's bit j) ≠ 0 then
17                     delete d_a from buffer_A{}
18                     break
19                 end
20             end
21         end
22         for each memory block d_b ∈ buffer_B{} do
23             m_b = physical address of d_b
24             if (m_b's bit i) ≠ 1 then
25                 delete d_b from buffer_B{}
26                 break
27             end
28             for each bit j ∈ bank_bit{} and i ≠ j do
29                 if (m_b's bit j) ≠ 0 then
30                     delete d_b from buffer_B{}
31                     break
32                 end
33             end
34         end
35         thread_A: // access buffer_A in an infinite loop
36         while (true) do
37             for each memory block d_a ∈ buffer_A{} do
38                 access d_a (uncached)
39             end
40         end
41         thread_B:// access buffer_B N times and measure time
42         for i=0 to N-1 do
43             for each memory block d_b ∈ buffer_B{} do
44                 access d_b (uncached)
45             end
46         end
47         total_time = thread_B's execution time;
48         if total_time<Threshold then
49             add i to channel_bit{}
50         end
51     end
52     return channel_bit{}
53 end
```

Algorithm 4.3: Adaptive memory flooding

```
   Input:
1      memory_channel{}: all the channels in the memory
2      memory_buffer{}
3  begin
4      /* Discover Stage */
5      victim_channel=∅
6      for each channel i in memory_channel{} do
7          access the addresses belonging to channel i from memory_buffer{}, and measure the total time
             (repeat for a number of times)
8          if total time is high then
9              add i to victim_channel{}
10         end
11     end

12     /* Attack Stage */
13     while attack is not finished do
14         for each channel i in victim_channel{} do
15             access the addresses belonging to channel i from memory_buffer{}
16         end
17     end
18 end
```

used by the victim. Note that the attacker only needs to discover the channels used by the victim, but does not need to know the exact value of channel index bits for a given channel. In the Attack Stage, the attacker floods these selected memory channels. Algorithm 4.3 shows the two steps to conduct adaptive memory flooding attacks.

Figure 4.11 shows the results when the attacker VM uses 8 vCPUs to generate contention in selected memory channels which are heavily used by the victim. These adaptive memory flooding attacks cause 3% $\sim$ 44% slowdown while indiscriminately flooding the entire memory causes only 0.07 $\sim$ 22% slowdown.

4.3 Case Studies in Amazon EC2

We now evaluate our memory DoS attacks in a real cloud environment, Amazon EC2. We provide two case studies: memory DoS attacks against distributed applications, and against E-Commerce websites.

Legal and ethical considerations. As our attacks only involve memory accesses within the attacker VM's own address space, the experiments we conducted in this

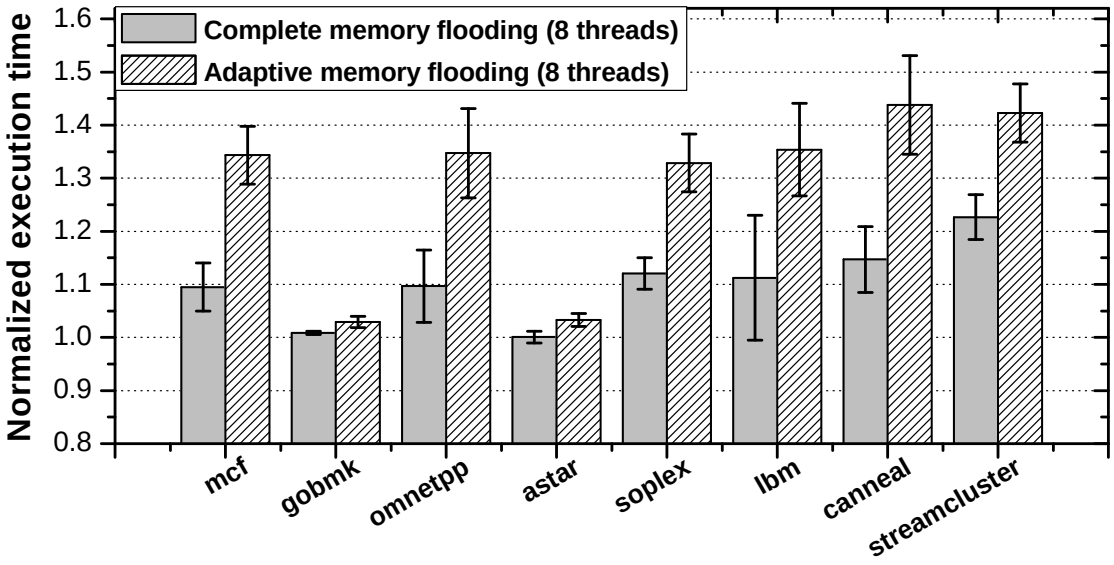


Figure 4.11: Performance slowdown due to multi-threaded and adaptive memory flooding attacks.

section conformed with the EC2 customer agreement. Nevertheless, we put forth our best efforts in reducing the duration of the attacks to minimally impact other users in the cloud.

VM configurations. We chose the same configuration for the attacker and victim VMs: t2.medium instances with 2 vCPUs, 4GB memory and 8GB disk. Each VM ran Ubuntu Server 14.04 LTS with Linux kernel version 3.13.0-48-generic, in full virtualization mode. All VMs were launched in the us-east-1c region. Information exposed through `lscpu` indicated that these VMs were running on 2.5GHz Intel Xeon E5-2670 processors, with a 32KB L1D and L1I cache, a 256KB L2 cache, and a shared 25MB LLC.

For all the experiments in this section, the attacker employs exotic atomic locking (Section 4.2.3) and LLC cleansing attacks (Section 4.2.2), where each of the 2 attacker vCPUs was used to keep locking the memory and cleansing the LLC. Memory contention attacks (Section 4.2.4) are not used since they cause much lower performance degradation (availability loss) to the victim.

VM co-location in EC2. The memory DoS attacks require the attacker and victim VMs to co-locate on the same machine. Past work [182, 226, 248] have proven the feasibility of such co-location attacks in public clouds. While cloud providers adopt new technologies (e.g., Virtual Private Cloud [4]) to mitigate prior attacks in [182], new ways are discovered to test and detect co-location in [226, 248]. Specifically, Varadarajan et al. [226] achieved co-location in Amazon EC2, Google Compute Engine and Microsoft Azure with low-cost (less than $8) in the order of minutes. They verified co-location with various VM configurations, launch delay between attacker and victim, launch time of day, datacenter location, etc. Xu et al . [248] used similar ideas to achieve co-location in EC2 Virtual Private Cloud. We also applied these techniques to achieve co-location in Amazon EC2. In our experiments, we simultaneously launched a large number of attacker VMs in the same region as the victim VM. A machine outside EC2 under our control sent requests to static web pages hosted in the target victim VM. Each time we select one attacker VM to conduct memory DoS attacks and measure the victim VM's response latency. Delayed HTTP responses from the victim VM indicates that this attacker was sharing the machine with the victim.

4.3.1 Attacking Distributed Applications

We evaluate memory DoS attacks on a multi-node distributed application deployed in a cluster of VMs, where each VM is deployed as one node. We show how much performance degradation an adversary can induce to the victim cluster with minimal cost, using a single co-located attacker VM.

Experiment settings. We used Hadoop as the victim system. Hadoop consists of two layers: MapReduce for data processing, and Hadoop Distributed File System (HDFS) for data storage. A Hadoop cluster includes a single master node and multiple slave nodes. The master node acts as both the Job Tracker for scheduling map or reduce jobs and the NameNode for hosting HDFS indexes. Each slave node acts as

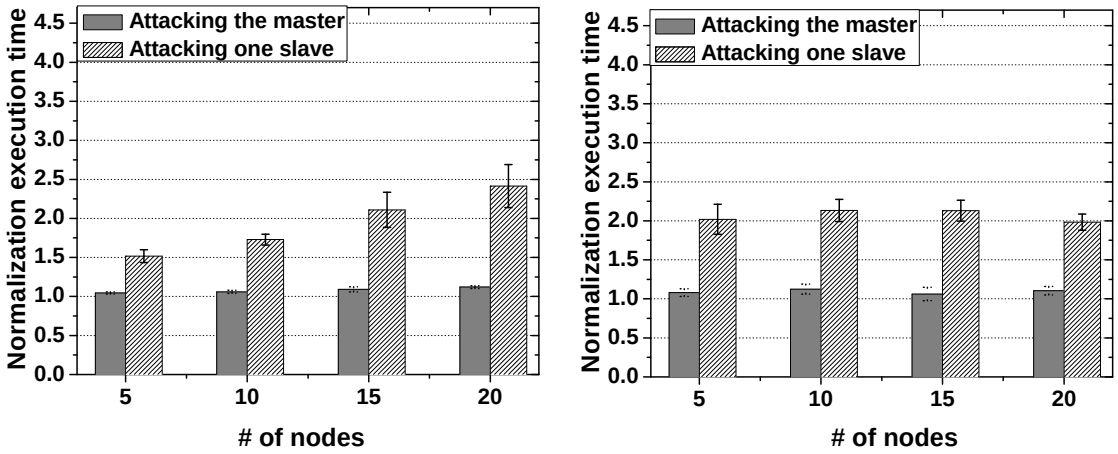
both the Task Tracker for conducting the map or reduce operations and the DataNode for storing data blocks in HDFS. We deployed the Hadoop system with different numbers of VMs (5, 10, 15 or 20), where one VM was selected as the master node and the rest were the slave nodes.

The attacker only used *one* VM to attack the cluster. He either co-located the malicious VM with the master node or one of the slave nodes. We ran four different Hadoop benchmarks to test how much performance degradation the single attacker VM can cause to the Hadoop cluster. Each experiment was repeated 5 times. Figure 4.12 shows the mean values of normalized execution time and one standard deviation.

MRBench: This benchmark tests the performance of the MapReduce layer of the Hadoop system: it runs a small MapReduce job of text processing for a number of times. We set the number of mappers and reducers as the number of slave nodes for each experiment. Figure 4.12a shows that attacking a slave node is more effective since the slave node is busy with the map and reduce tasks. In a large Hadoop cluster with 20 nodes, attacking just one slave node introduces 2.5× slowdown to the entire distributed system.

TestDFSIO: We use TestDFSIO to evaluate HDFS performance. This benchmark writes and reads files stored in HDFS. We configured it to operate on n files with the size of 500MB, where n is the number of slave nodes in the Hadoop cluster. Figure 4.12b shows that attacking the slave node is effective: the adversary can achieve about 2× slowdown.

NNBench: This program is also used to benchmark HDFS in Hadoop. It generates HDFS-related management requests on the master node of HDFS. We configured it to operate on $200n$ small files, where n is the number of slave nodes in the Hadoop cluster. Since the master node is heavily used for serving the HDFS requests, attacking the master node can introduce up to 3.4× slowdown to the whole Hadoop system, as shown in Figure 4.12c.



(a) MRBench (b) TestDFSIO

(c) NNBench (d) TeraSort

Figure 4.12: Performance slowdown of the Hadoop applications due to memory DoS attacks.

Terasort: We use this benchmark to test the overall performance of both MapReduce and HDFS layers in the Hadoop cluster. TeraSort generates a large set of data and uses map/reduce operations to sort the data. For each experiment, we set the number of mappers and reducers to n, and the size of data to be sorted to $100n$ MB, where n is the number of slave nodes in the Hadoop cluster. Figure 4.12d shows that attacking the slave node is very effective: it can bring 2.8 $\sim$ 3.7 $\times$ slowdown to the entire Hadoop system.

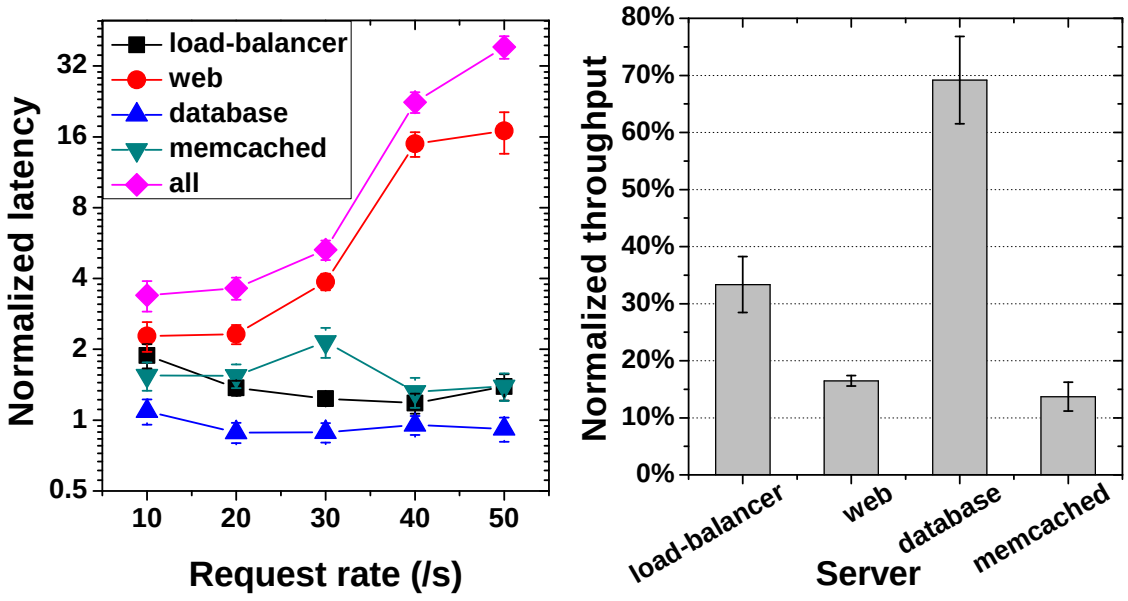
Summary. The adversary can deny working memory availability to the victim VM and thus degrade an important distributed system's performance with minimal costs: it can use just one VM to interfere with one of 20 nodes in the large cluster. The slowdown of a single victim node can cause up to 3.7× slowdown to the whole system.

4.3.2 Attacking E-Commerce Websites

A web application consists of load balancers, web servers, database servers and memory caching servers. Memory DoS attacks can disturb an E-commerce web application by attacking various components.

Experiment settings. We chose a popular open source E-commerce web application, Magento [13], as the target of the attack. The victim application consists of five VMs: a load balancer based on Pound for balancing network requests; two Apache web servers to process and deliver web requests; a MySQL database server to store customer and merchandise information; and a Memcached server to speed up database transactions. The five VMs were hosted on different cloud servers in EC2. The adversary is able to co-locate his VMs with one or multiple VMs that host the victim application. We measure the application's latency and throughput to evaluate the effectiveness of the attack.

Latency. We launched a client on a local machine outside of EC2. The client employed httperf [25] to send HTTP requests to the load balancer with different rates (connections per second) and we measured the average response time. We evaluated the attack from one or all co-located VMs. Each experiment was repeated 10 times and the mean and standard deviation of the latency are reported in Figure 4.13a. This shows that memory contention on database, load balancer or memcached servers do not have much impact on the overall performance of the web application, with only up to 2× degradation. This is probably because these servers were not heavily used in these cases. Memory DoS attacks on web servers were the most effective (17×



(a) Latency (b) Throughput

Figure 4.13: Latency and throughput of the Magento application due to memory DoS attacks.

degradation). When the adversary can co-locate with all victim servers and each attacker VM induces contention with the victim, the web server's HTTP response time was delayed by 38×, for a request rate of 50 connections per second.

Server throughput. Figure 4.13b shows the results of another experiment, where we measured the throughput of each victim VM individually, under memory DoS attacks. We used ApacheBench [1] to evaluate the load balancer and web servers, SysBench [24] to evaluate the database server and memtier_benchmark [14] to evaluate the memcached server. This shows memory DoS attacks on these servers were effective: the throughput can be reduced to only 13% $\sim$ 70% under malicious contention by the attacker.

Summary. The adversary can compromise the quality of E-commerce service and cause financial loss in two ways: (1) long response latency will affect customers' satisfaction and make them leave this E-commerce website [176]; (2) it can cause throughput degradation, reducing the number of transactions completed in a unit

time. The cost for these attacks is relatively cheap: the adversary only needs a few VMs to perform the attacks, with each t2.medium instance costing $0.052 per hour.

4.4 Defense against Memory DoS Attacks

We propose a novel, general-purpose approach to detecting and mitigating memory DoS attacks in the cloud. Unlike some past work, our defense does not require prior profiling of the memory resource usage of the applications. Our defense can be provided by the cloud providers as a new security service to customers. We denote as PROTECTED VMs those VMs for which the cloud customers require protection. To detect memory DoS attacks, lightweight statistical tests are performed frequently to monitor performance changes of the PROTECTED VMs (Section 4.4.1). To mitigate the attacks, *execution throttling* is used to reduce the impact of the attacks (Section 4.4.2). A novelty of our approach is the combined use of two existing hardware features: *event counting* using hardware performance counters controllable via the Performance Monitoring Unit (PMU) and *duty cycle modulation* controllable through the `IA32_CLOCK_MODULATION` Model Specific Register (MSR).

4.4.1 Detection Method

The key insight in detecting memory DoS attacks is that *such attacks are caused by abnormal resource contention between* PROTECTED VM*s and attacker VMs, and such resource contention can significantly alter the memory usage of the* PROTECTED VM*, which can be observed by the cloud provider.* We postulate that the statistics of accesses to memory resources, by a phase of a software program, follow certain probability distributions. When a memory DoS attack happens, these probability distributions will change. Figure 4.14 shows the probability distributions of the PROTECTED VM's memory access statistics, without attacks (black), and with two kinds of attacks (gray

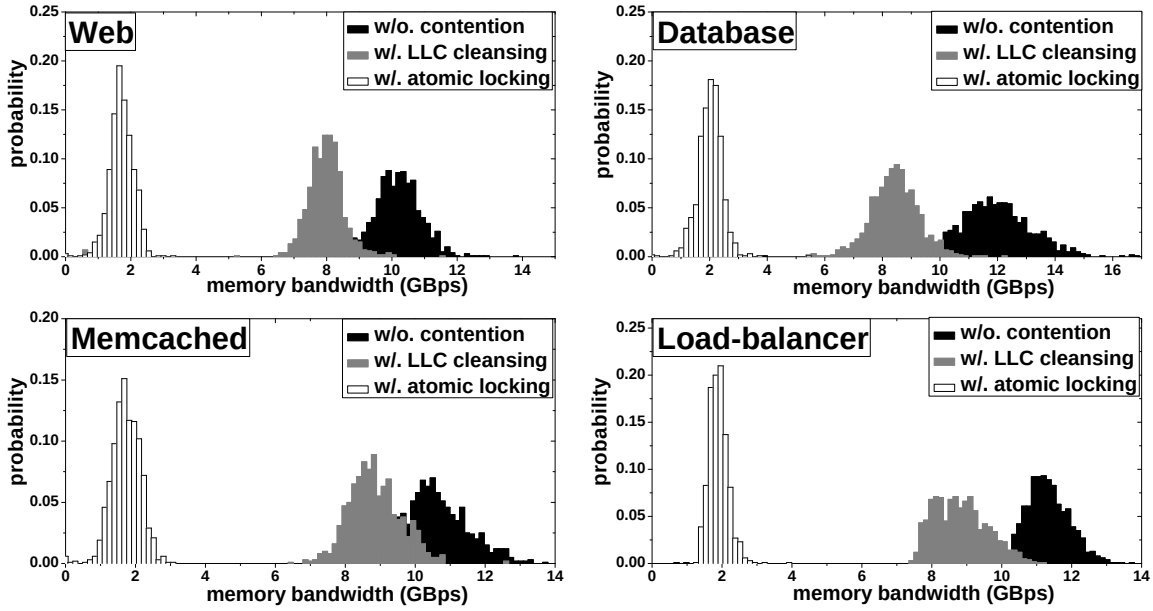


Figure 4.14: Probability distributions of the PROTECTED VM's memory bandwidth.

and shaded), when it runs one of four applications introduced in Section 4.3.2, i.e., the Apache web server, Mysql database, Memcached and Pound load-balancer. When an attacker is present, the probability distribution of the PROTECTED VM's memory access statistics (in this case, memory bandwidth in GigaBytes per second) changes significantly.

In practice, only samples drawn from the underlying probability distribution are observable. Therefore, the provider's task is to collect two sets of samples: $[X_1^{\mathrm{R}}, X_2^{\mathrm{R}}, ..., X_{n^{\mathrm{R}}}^{\mathrm{R}}]$ are reference samples collected from the probability distribution when we are sure that there are no attacks; $[X_1^{\mathrm{M}}, X_2^{\mathrm{M}}, ..., X_{n^{\mathrm{M}}}^{\mathrm{M}}]$ are monitored samples collected from the PROTECTED VM at runtime, when attacks may occur. If these two sets of samples are not drawn from the same distribution, we can conclude that the performance of the PROTECTED VM is hindered by its neighboring VMs. When the distance between the two distributions is large, we may conclude the PROTECTED VM is under some memory DoS attacks.

We propose to use the two-sample Kolmogorov-Smirnov (KS) tests [152], as a metric for whether two samples belong to the same probability distribution. The KS

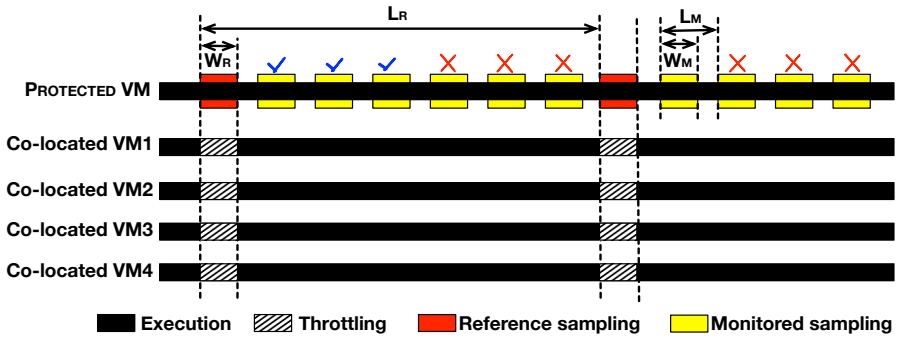
statistic is defined in Equation 4.1, where $F_n(x)$ is the empirical distribution function of the samples $[X_1, X_2, ..., X_n]$, and *sup* is the supremum function (i.e., returning the maximum value). Superscripts M and R denote the monitored samples and reference samples, respectively. n^M and n^R are the number of monitored samples and reference samples.

$$D_{n^{\mathrm{M}},\, n^{\mathrm{R}}} = \sup_x \mid F^{\mathrm{M}}_{n^{\mathrm{M}}}(x) - F^{\mathrm{R}}_{n^{\mathrm{R}}}(x) \mid \tag{4.1}$$

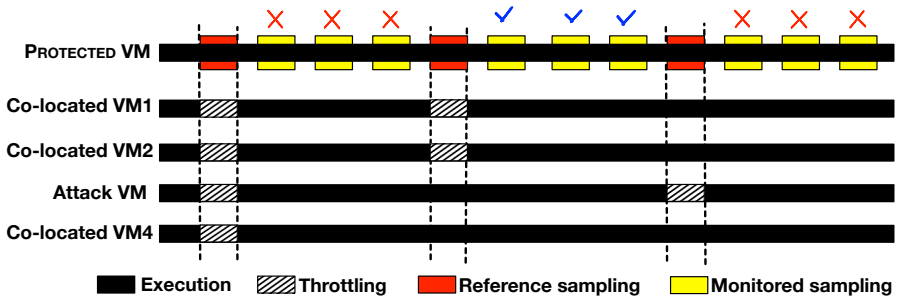
$$D^{\alpha}_{n^{\mathrm{M}},\, n^{\mathrm{R}}} = \sqrt{\frac{n^{\mathrm{M}} + n^{\mathrm{R}}}{n^{\mathrm{M}} \times n^{\mathrm{R}}}} \sqrt{-0.5 \times ln(\frac{\alpha}{2})} \tag{4.2}$$

Null hypothesis for KS test. We establish the null hypothesis that currently monitored samples are drawn from the same distribution as the reference samples. Benign performance contention with non-attacking, co-tenant VMs will not alter the probability distribution of the PROTECTED VM's monitored samples significantly, so the KS statistic is small and the null hypothesis is held. Equation 4.2 introduces α: We can reject the null hypothesis with confidence level $1 - \alpha$ if the KS statistic, $D_{n^{\mathrm{M}},\, n^{\mathrm{R}}}$, is greater than predetermined critical values $D^{\alpha}_{n^{\mathrm{M}},\, n^{\mathrm{R}}}$. Then, the cloud provider can assume, with confidence level $1 - \alpha$, that a memory DoS attack exists, and trigger a mitigation strategy.

While monitored samples, X^{M}_i, are simply collected at runtime, reference samples, X^{R}_i, ideally should be collected when the PROTECTED VM is not affected by other co-located VMs. The technical challenge here is that if these samples are collected offline, we need to assume the memory access statistics of the VM never change during its life time, which is unrealistic. If samples are collected at runtime, all the co-locating VMs need to be paused during sample collection, which, if performed frequently, can cause significant performance overhead to benign, co-located VMs.



(a) Monitoring the PROTECTED VM.



(b) Identifying co-located VM3 as the attacker VM.

Figure 4.15: Illustration of monitoring the PROTECTED VM (a) and identifying the attack VM (b). The blue "✔" means the null hypothesis is accepted; while the red "✘" means the null hypothesis is rejected.

Pseudo Isolated Reference Sampling. To address this technical challenge, we use *execution throttling* to collect the reference samples at runtime. The basic idea is to throttle down the execution speed of other VMs, but maintain the PROTECTED VM's speed during the reference sampling stage. This can reduce the co-located VMs' interference without pausing them.

Execution throttling is based on a feature provided in Intel Processors called *duty cycle modulation* [9], which is designed to regulate each core's execution speed and power consumption. The processor allows software to assign "duty cycles" to each CPU core: the core will be active during these duty cycles, and inactive during the non-duty cycles. For example, the duty cycle of a core can be set from 16/16 (no throttling), 15/16, 14/16, ..., down to 1/16 (maximum throttling). Each core uses

a model specific register (MSR), `IA32_CLOCK_MODULATION`, to control the duty cycle ratio: bit 4 of this MSR denotes if the duty cycle modulation is enabled for this core; bits 0-3 represent the number of 1/16 of the total CPU cycles set as duty cycles.

In execution throttling, the execution speed of other VMs will be throttled down and very little contention is induced to the PROTECTED VM. As such, reference samples collected during the execution throttling stage are drawn from a quasi contention-free distribution.

Figure 4.15a illustrates the high-level strategy for monitoring PROTECTED VMs. The reference samples are collected during the reference sampling periods (W_R), where other VMs' execution speeds are throttled down. The monitored samples are collected during the monitored sampling periods (W_M), where co-located VMs run normally, without execution throttling. KS tests are performed right after each monitored sample is collected, and probability distribution divergence is estimated by comparing with the most recent reference samples. Monitored samples are collected periodically at a time interval of L_M, and reference samples are collected periodically at a time interval of L_R. We can also randomize the intervals L_M and L_R for each period to prevent the attacker from reverse-engineering the detection scheme and scheduling the attack phases to avoid detection.

If the KS test results reject the null hypothesis, it may be because the PROTECTED VM is in a different execution phase with different memory access statistics, or it may be due to memory DoS attacks. To rule out the first possibility, double checking automatically occurs since reference samples are re-collected and updated after a time interval of L_R. If deviation of the probability distribution still exists, attacks can be confirmed.

4.4.2 Mitigation Method

The cloud provider has several methods to mitigate the attack. One is VM migration, which can be achieved either by reassigning the vCPUs of a VM to a different CPU package, when the memory resource being contended is in the same package (e.g., LLC), or by migrating the entire VM to another server, when the memory resource contended is shared system-wide (e.g., memory bus). However, such VM migration cannot completely eliminate the attacker VM's impact on other VMs.

An alternative approach is to identify the attacker VM, and then employ *execution throttling* to reduce the execution speed of the malicious VM, while meanwhile the cloud provider conducts further investigation and/or notifies the customer of the suspected attacker VM of observed resource abuse activities.

Identifying the attacker VM. Once memory DoS attacks are detected, to mitigate the threat, the cloud provider needs to identify which of the co-located VMs is conducting the attack. Here we propose a novel approach to identify malicious VMs based on *selective execution throttling in a binary search manner*: First, half of the co-located VMs keep normal execution speed while the rest of VMs are throttled down during reference sampling periods (Figure 4.15b, 2nd Reference Sampling period). If in this case, reference samples and monitored samples are drawn from the same distribution, then there are malicious VMs among the ones not throttled down during the reference sampling period. Then, we select half of the remaining VMs to be throttled while all the other VMs are in normal speed, to collect the next reference samples. In Figure 4.15b, this is the 3rd Reference Sampling period, where only VM3 is throttled. Since the subsequent monitored samples have a different distribution compared to this Reference Sample, VM3 is identified as the attack VM. Note that if there are multiple attacker VMs on the server, we can use the above procedure to find one VM each time and repeat it until all the attacker VMs are found. By organizing

Algorithm 4.4: Identifying and mitigating the attacker VMs that cause severe resource contention.

```
   Input:
1      VM[1,...,n]        /* set of co-tenant VMs */

2  function IdentifyAttacker(sub_VM)
3      /* sub_VM: set of VMs to identify */
4      if sub_VM.length() = 1 then
5          return sub_VM[0]
6      else
7          imin = 0
8          imax = sub_VM.length()-1
9          imid = ⌈(imin+imax)/2⌉
10         ThrottleDown(sub_VM[0,...,imid-1])
11         reference_sample = DataCollect()
12         ThrottleUp(sub_VM[0,...,imid-1])
13         monitor_sample = DataCollect()
14         result = KSTest(reference_sample, monitor_sample)
15         if result = Reject then
16             return IdentifyAttacker(sub_VM[0,...,imid-1])
17         else
18             return IdentifyAttacker(sub_VM[imid,...,imax])
19         end
20     end
21 end

22 begin
23     vm = IdentifyAttacker(VM)
24     ThrottleDown([vm])
25 end
```

this search for the attacker VM or VMs as a binary search, the time taken to identify the source of memory contention is O(log n), where n is the number of co-tenant VMs on the PROTECTED VM's server. Algorithm 4.4 shows how to detect attacker VMs using this *selective execution throttling*.

4.4.3 Implementation

We implement a prototype system of our proposed defense on the OpenStack platform. Figure 4.16 shows the defense architecture overview. We adopt the *CloudMonatt* architecture from Chapter 3 and [262] (Details about the integration of this defense in *CloudMonatt* will be illustrated in Section 7.1). Specifically, the system includes three types of servers. The Cloud Controller is the cloud manager that manages the VMs. It has a `Policy Validation Module` to receive and analyze customers' requests. It also has a `Response Module`, which can throttle down the attacker VMs' execution

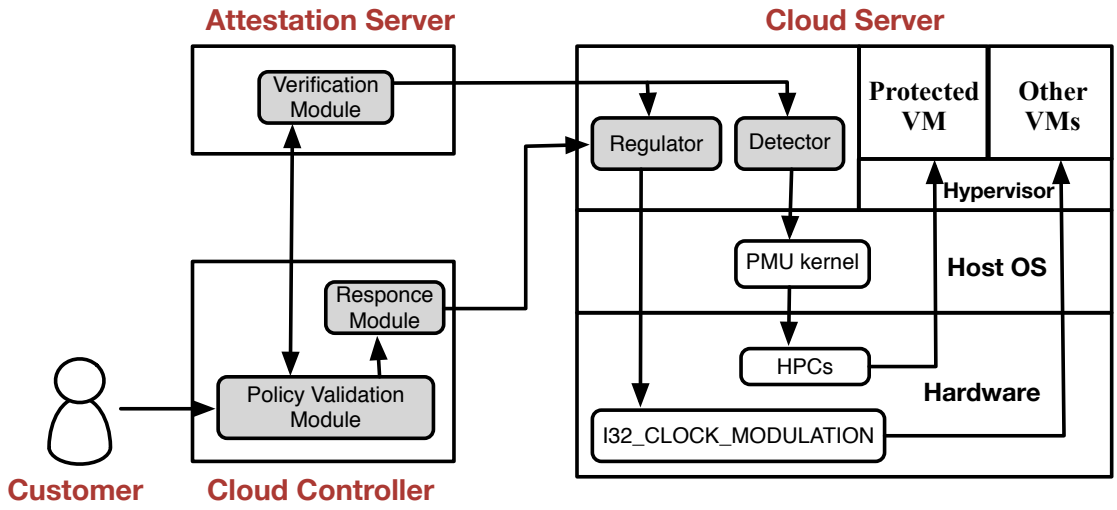


Figure 4.16: Architecture overview.

speed to mitigate memory DoS attacks. The Attestation Server is a centralized server for monitoring and detection of memory DoS attacks. It has a `Verification Module` that receives PROTECTED VM's performance probability distribution, detects memory DoS attacks and identifies malicious VMs.

On each of the cloud servers, we use the KVM hypervisor which is the default setup for OpenStack. Other virtualization platforms, such as Xen and HyperV, can also be used. Two software modules are installed on the host OS. A `Detector` measures the memory access characteristics of the PROTECTED VM using Performance Monitoring Units (PMU), which are commonly available in most modern processors. A PMU provides a set of Hardware Performance Counters to count hardware-related events. In our implementation, we use the linux kernel API `perf_event` to measure the memory access statistics for the number of *LLC accesses* per sampling period. A `Regulator` is in charge of controlling VMs' execution speed. It uses the `wrmsr` instruction to modify the `IA32_CLOCK_MODULATION` MSR to control the duty cycle ratio.

In our implementation, the parameters involved in reference and monitored sampling are as follows: $W_R = W_M = 1s$, $L_M = 2s$, $L_R = 30s$. These values were selected to strike a balance between the performance overhead due to execution throttling

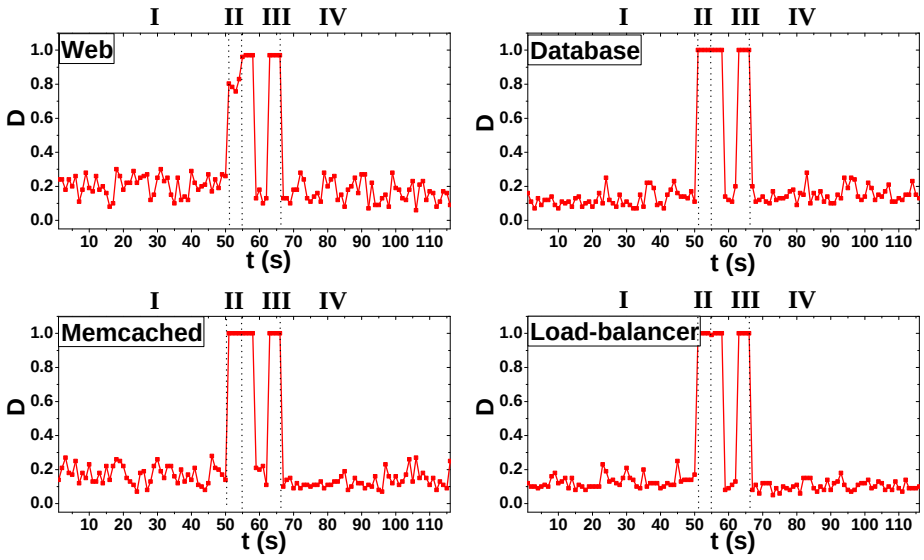
and detection accuracy. In each sampling period, $n = 100$ samples are collected, with each collected during a period of 10ms. We choose 10ms because it is short enough to provide accurate measurements, and long enough to return stable results. In the KS tests, the confidence level, $1 - \alpha$, is set as 0.999, and the threshold to reject the null hypothesis is $D^{\alpha} = 0.276$ (given $\alpha = 0.001$). If 4 consecutive KS statistics larger than 0.276 are observed (the choice of 4 is elaborated in Section 4.4.4), it is assured that the PROTECTED VM's memory access statistics have been changed. Then to confirm that such changes are due to memory DoS attacks, reference samples will be refreshed and the malicious VM will be identified.

4.4.4 Evaluation

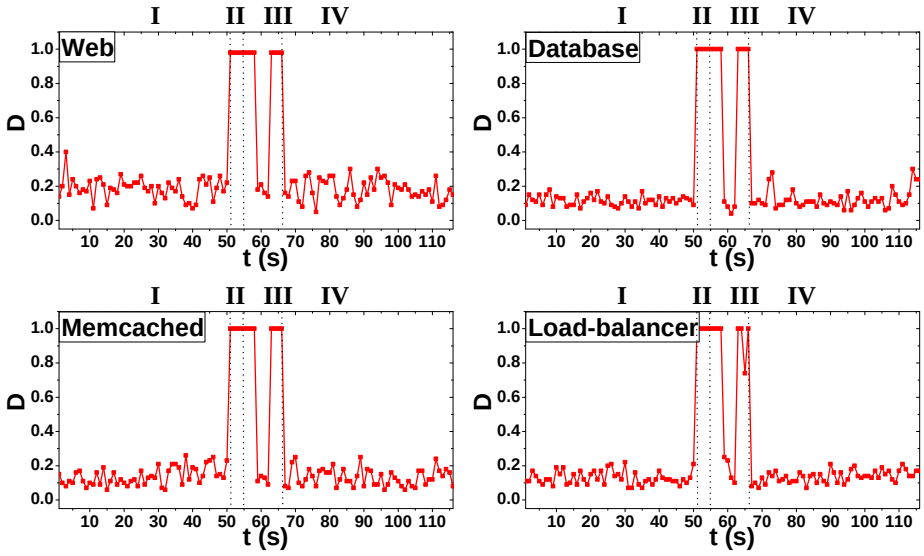
Our lab testbed comprised three servers. A Dell R210II Server (equipped with one quad-core, 3.30GHz, Intel Xeon E3-1230v2 processor with 8MB LLC) was configured as the Cloud Controller as well as the Attestation Server. Two Dell PowerEdge R720 Servers (one has two six-core, 2.90GHz Intel Xeon E5-2667 processors with 15MB LLC, the other has one eight-core, 2.90GHz Intel Xeon E5-2690 processor with 20MB LLC) were deployed to function as VM hosting servers.

Detection accuracy. We deployed a PROTECTED VM sharing a cloud server with 8 other VMs. Among these 8 VMs, one VM was an attacker VM conducting a multi-threaded LLC cleansing attack with 4 threads (Section 4.2.2), or an atomic locking attack (Section 4.2.3). The remaining 7 VMs were benign VMs running common linux utilities. The PROTECTED VM runs one of the web, database, memcached or load-balancer applications in the Magento application (Section 4.3.2). The experiments consisted of four stages; the KS statistics of each of the four workloads during the four stages under the two types of attacks are shown in Figure 4.17.

In stage I, the PROTECTED VM runs while the attacker is idle. The KS statistic in this stage is relatively low. So we accept the null hypothesis that the memory accesses



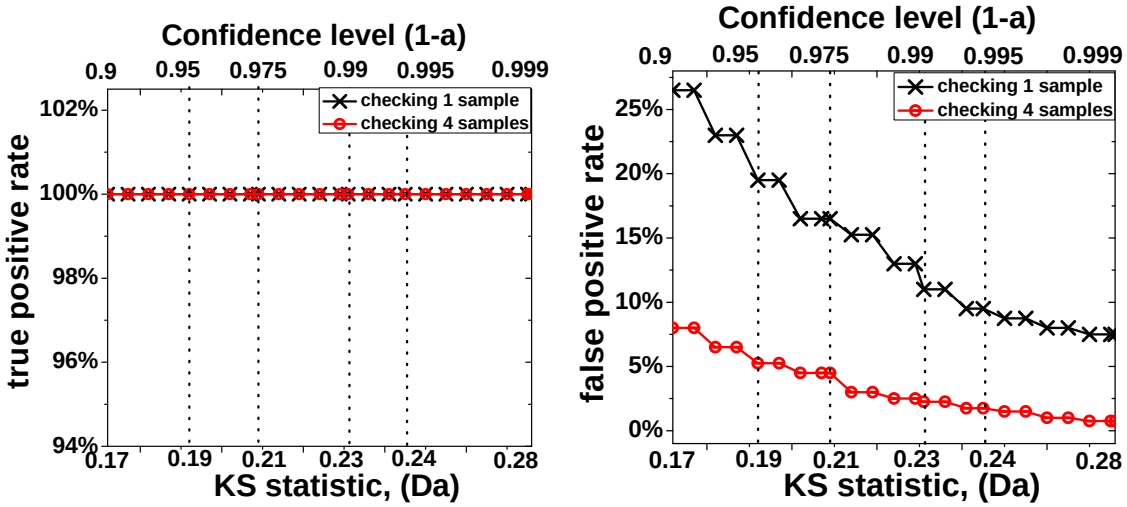
(a) LLC cleansing attack



(b) Atomic locking attack

Figure 4.17: KS statistics of the PROTECTED VM for detecting and mitigating memory DoS attacks.

of the reference and monitored samples follow the same probability distribution. In stage II, the attacker VM conducts the LLC cleansing or atomic locking attacks. We observe the KS statistic is much higher than 0.276. The null hypothesis is rejected, signaling detection of potential memory DoS attacks. In stage III, the cloud provider runs *three* rounds of reference resampling to pinpoint the malicious VM. Resource



(a) True positive

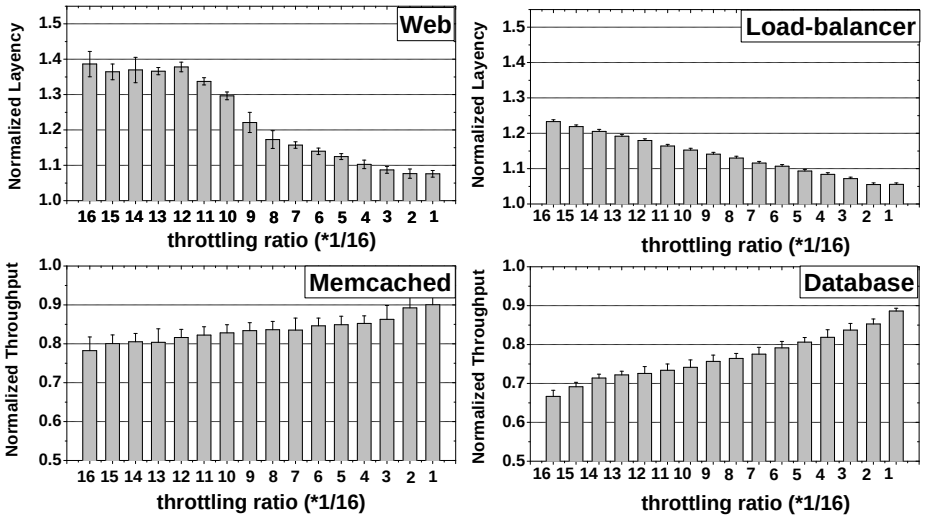
(b) False positive

Figure 4.18: Detection accuracy.

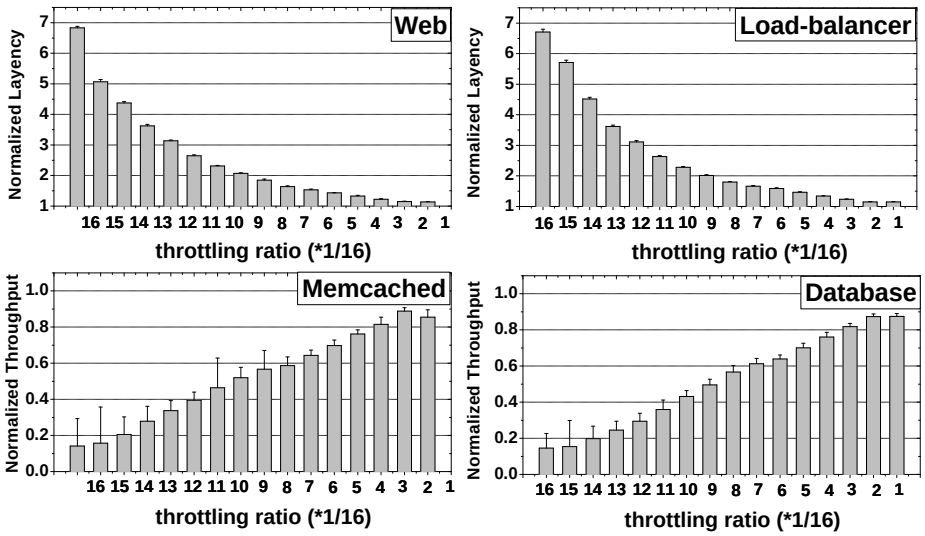
contention mitigation is performed in stage IV: the cloud provider throttles down the attacker VM's execution speed. After this stage, the KS statistic falls back to normal which suggests that the attacks are mitigated.

We also evaluated the false positive rates and false negative rates of two different criteria for identifying a memory access anomaly: 1 abnormal KS statistic (larger than the critical value D^{α}) or 4 consecutive abnormal KS statistics. Figure 4.18a shows the true positive rate of LLC cleansing and atomic locking attack detection, at different confidence levels $1 - \alpha$. We observe that the true positive rate is always one (thus zero false negatives), regardless of the detection criteria (1 vs 4 abnormal KS tests). Figure 4.18b shows the false positive rate, which can be caused by background noise due to other VMs' executions. This figure shows that using 4 consecutive abnormal KS statistics significantly reduces the false positive rate.

Effectiveness of mitigation. We evaluated the effectiveness of execution throttling based mitigation. The PROTECTED VM runs the cloud benchmarks from the Magento application while the attacker VM runs LLC cleansing or atomic locking attacks. We chose different duty cycle ratios for the attacker VM. Figures 4.19a and 4.19b show



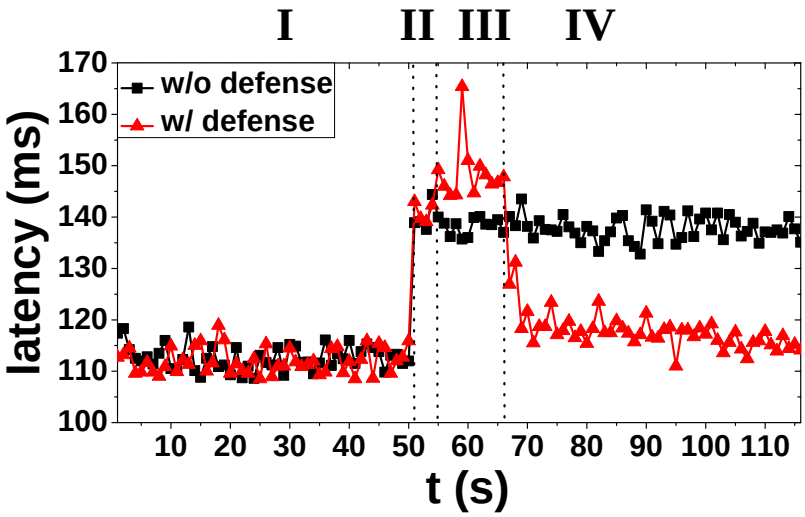
(a) Throttling LLC cleansing attacks



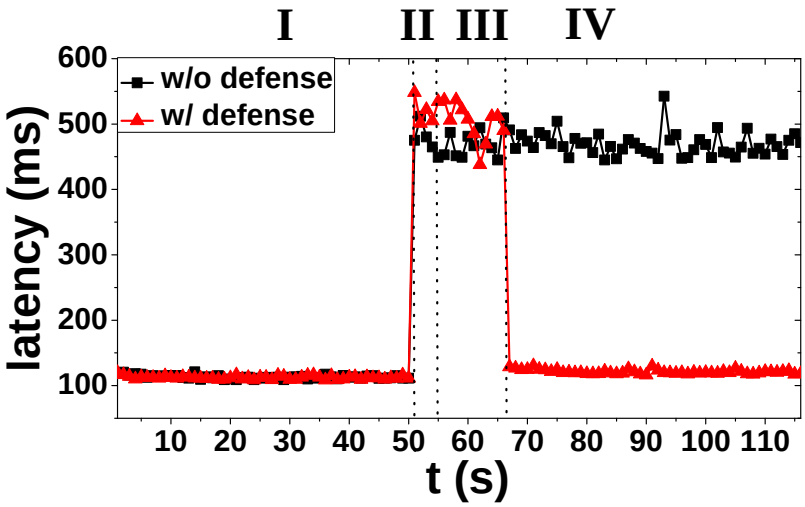
(b) Throttling atomic locking attacks

Figure 4.19: Normalized performance of the PROTECTED VM with throttling of memory DoS attacks.

the normalized performance of the PROTECTED VM with different throttling ratios, under LLC cleansing and atomic locking attacks, respectively. The x-axis shows the duty cycle ($x \times 1/16$) given to the co-located VMs, going from no throttling on the left to maximum throttling on the right of each figure. The y-axis shows the PROTECTED VM's response latency (for web and load-balancer) or throughput (for memcached and database) normalized to the ones without attack. A high latency or a small throughput indicates that the performance of the PROTECTED VM is highly



(a) LLC cleansing attack



(b) Atomic locking attack

Figure 4.20: Request latency of Magento Application

taffected by the attacker VM. We can see that a smaller throttling ratio can effectively reduce the attacker's impact on the victim's performance. When the ratio is set as 1/16, the victim's performance degradation caused by the attacker is kept within 12% (compared to 23% $\sim$ 50% degradation with no throttling) for LLC cleansing attacks. It is within 14% for atomic locking attacks (compared to 7$\times$ degradation with no throttling).

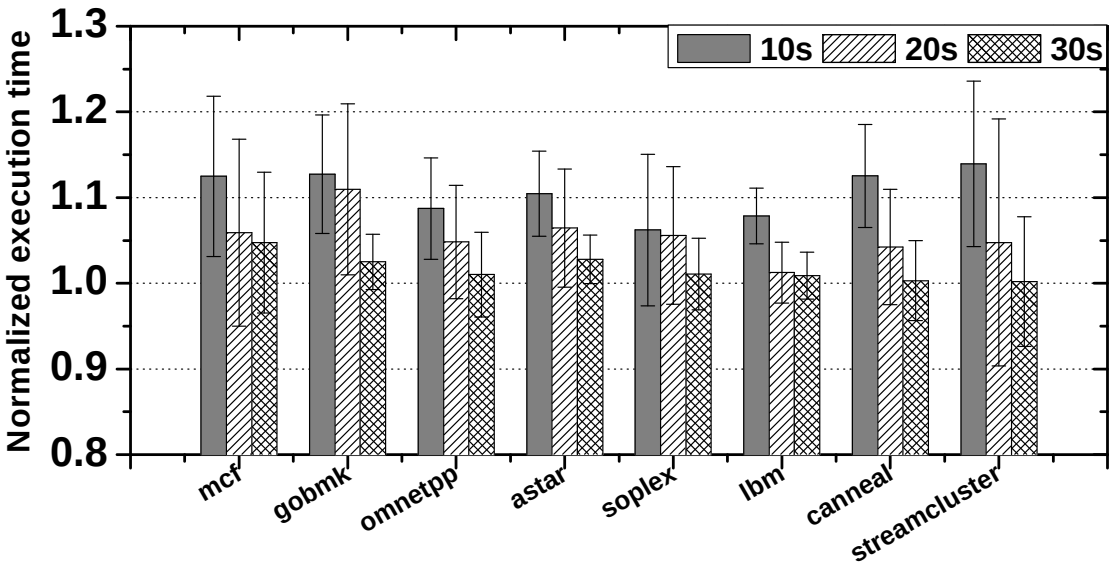


Figure 4.21: Performance overhead of co-located VMs due to monitoring.

Latency increase and mitigation. We chose a latency-critical application, the Magento E-commerce application as the target victim. One Apache web server was selected as the PROTECTED VM, co-locating with an attacker and 7 benign VMs running linux utilities. Figure 4.20 shows the response latency with and without our defense. The detection phase does not affect the PROTECTED VM's performance (stage I), since the PMU collects monitored samples without interrupting the VM's execution. In stage II, the attack occurs and the defense system detects the PROTECTED VM's performance is degraded. In stage III, attacker VM identification is done. After throttling down the attacker VM in stage IV, the PROTECTED VM's performance is not affected by the memory DoS attacks. The latency during the attack in Phase II increases significantly, but returns to normal after mitigation in Phase IV.

We also evaluated the performance overhead of co-located VMs due to *execution throttling* in the detection step. We launched one VM running one of the eight SPEC2006 or PARSEC benchmarks. Then we periodically throttle down this VM every 10s, 20s or 30s. Each time throttling lasted for 1s (the same value for W_R and W_M used earlier). The normalized performance of this VM is shown in Figure 4.21.

We can see that when the server throttles this VM every 10s, the performance penalty can be around 10%. However, when the frequency is set to be 30s (our implementation choice), this penalty is smaller than 5%.

4.5 Chapter Summary

We presented memory DoS attacks, in which a malicious VM intentionally induces memory resource contention to degrade the performance of co-located victim VMs. We proposed several advanced techniques to conduct such attacks, and demonstrate the severity of the resulting performance degradation. Our attacks work on modern memory systems in cloud servers, for which prior attacks on older memory systems are often ineffective. We evaluated our attacks against two commonly used applications in a public cloud, Amazon EC2, and show that the adversary can cause significant performance degradation to not only co-located VMs, but to the entire distributed application.

We then designed a novel and generalizable method that can detect and mitigate all known memory DoS attacks. Our approach collects the PROTECTED VM's reference and monitored behaviors at runtime using the Performance Monitor Unit. This is done by establishing a pseudo isolated collection environment by using the duty-cycle modulation feature to throttle the co-resident VMs for collecting Reference samples. Statistical tests are performed to detect differing performance probability distributions between Reference and Monitored samples, with desired confidence levels. Our evaluation shows this defense can detect and defeat memory DoS attacks with very low performance overhead.

Chapter 5

Detection and Mitigation of Confidentiality Vulnerabilities

In Chapter 3 we show the covert-channel attacks and their detection in *CloudMonatt*. In this chapter, we consider another form of attacks on the confidentiality property: cross-VM side-channel attacks. We present *CloudRadar*, a system to detect, and then mitigate, cache-based side-channel attacks in multi-tenant cloud systems (most parts of this chapter have been published in [264]). *CloudRadar* operates by correlating two events: first, it exploits signature-based detection to identify when the protected virtual machine (VM) executes a cryptographic application (e.g., encryption/decryption, hash); at the same time, it uses anomaly-based detection techniques to monitor the co-located VMs to identify abnormal cache behaviors that are typical during cache-based side-channel attacks. We show that correlation in the occurrence of these two events offer strong evidence of side-channel attacks. Upon attack detection, *CloudRadar* can use VM migration to mitigate side-channel information leakage. *CloudRadar* is designed as a lightweight patch to existing cloud systems, which does not require new hardware support, or any hypervisor, operating system, or application modifications. We demonstrate a prototype implementation of *CloudRadar* in the OpenStack cloud

framework. Our evaluation suggests *CloudRadar* achieves negligible performance overhead with high detection accuracy.

5.1 Background

In a multi-tenant cloud server, virtualization technology is used to provide strong resource isolation between different VMs so each VM's memory content is not accessible to other co-tenant VMs. However, confidentiality breaches due to cross-VM side-channel attacks become a major concern. These attacks often operate on shared hardware resources and extract sensitive information, such as cryptographic keys, by making inferences on the observed side-channel events due to resource sharing. CPU caches are popular attack medium used in cross-VM side-channel attacks. Several prior work have shown the possibilities of cross-VM secret leakage via different levels of CPU caches [269, 255, 122, 270, 107, 144, 120].

Mitigating side-channel attacks in the cloud is challenging. Past work on defeating side-channel attacks have some practical drawbacks: they mostly require significant changes to the hardware [238, 239, 83, 142], hypervisors [229, 200, 131, 271, 225, 139] or guest OSes [271], making them impractical to be deployed in current cloud datacenters. Other work have proposed to mitigate these attacks in cloud contexts by periodic VM migrations to reduce the co-location possibility between victim VMs and potential malicious VMs [272, 159]. These heavy-weight approaches cannot effectively prevent side-channel leakage unless performed very frequently, making them less practical as VM co-location takes on the order of minutes [226] while side-channel attacks can be done on the order of milliseconds [255, 144].

In this chapter, we propose to detect side-channel attacks as they occur and prevent information leakage by triggering VM migration upon attack detection. However, side-channel attack detection is non-trivial. To do so, we must overcome several technical

challenges in the application of traditional detection techniques, like signature-based detection and anomaly-based detection, to side-channel attacks. Signature-based side-channel detection exploits pattern recognition to detect known attack methods [78, 68, 114]. While low in false negatives for existing attacks, it fails to recognize new attacks; anomaly-based detection flags behaviors that deviate significantly from the established normal behaviors as attacks, which can potentially identify new attacks in addition to known ones. However, differentiating side-channel attacks from normal applications is difficult as these attacks just perform normal memory accesses which resemble some memory intensive applications.

To overcome these challenges, we design *CloudRadar*, a real-time system to detect the existence of cross-VM side-channel attacks in clouds. There are two key ideas behind *CloudRadar*: first, *the victim has unique micro-architectural behaviors when executing cryptographic applications that need protection from side-channel attacks.* So the cloud provider is able to identify the occurrence of such events using a signature-based detection method. Second, *the attacker VM creates an anomalous cache behavior when it is stealing information from the victim.* Such anomaly is inherent in all side-channel attacks due to the intentional cache contention with the victim to induce side-channel observations. By correlating these two types of events, *CloudRadar* is able to detect the stealthy cache side-channel attacks with high fidelity.

We implement *CloudRadar* as a lightweight extension to virtual machine monitors. Specially, it (1) utilizes the existing host system facilities to collect micro-architectural features from hardware performance counters that are available in all modern commodity processors, and (2) non-intrusively interacts with the existing virtualization framework to monitor the VM's cache activities while inducing little performance penalty. Our evaluations show that it effectively detects side-channel attacks with high true positives and low false positives.

Compared to past work, *CloudRadar* has several advantages. First, *CloudRadar* focuses on the root causes of cache-based side-channel attacks and hence is hard to evade using different attack code, while maintaining a low false positive rate. Our approach is able to detect different types of side-channel attacks and their variants with a simple method. Second, *CloudRadar* is designed as a lightweight patch to existing cloud systems, which does not require new hardware support or hypervisor/OS modifications. Therefore *CloudRadar* can be immediately integrated into modern cloud fabric without making drastic changes to the underlying infrastructure. Third, *CloudRadar* exploits hardware performance counters to monitor VM activities, which detects side-channel attacks within the order of milliseconds with negligible performance overhead. Finally, *CloudRadar* requires no changes to the guest VM or the applications running in it, and thus is transparent to cloud customers.

5.1.1 Related Work

5.1.1.1 Cache Side-Channel Attacks

In cache-based side-channel attacks, the adversary exfiltrates sensitive information from the victim via shared CPU caches. The sensitive information are usually associated with cryptographic operations (e.g., signing or decryption), but may also be extended to other applications [270]. Such sensitive information are leaked through secret-dependent control flows or data flows that lead to attacker-observable cache use patterns. The adversary, on the other hand, may exploit several techniques to manipulate data in the shared cache to deduce the victim's cache use patterns, and thereby make inference on the sensitive information that dictates these patterns. Two cache manipulation techniques are well-known for side-channel attacks:

Prime-Probe attacks: The adversary allocates an array of cacheline-sized, cacheline-aligned memory blocks so that these memory blocks can exactly fill up a set of targeted

cache sets. Then the adversary repeatedly performs two attack stages: in the PRIME stage, the adversary reads each memory block in the array to evict all the victim's data in these cache sets. The adversary waits for some time interval before performing the PROBE stage, in which he reads each memory block in the array again, and measures the time of memory accesses. Longer access time indicates one or more cache misses, which means this cache set has been accessed by the victim between the PRIME and PROBE stages. The adversary will repeat these two steps for a large number of times to collect traces that, hopefully, overlap with the victim's execution of cryptographic operations, for offline analysis. This technique was first proposed by Percival [172], and then applied to the cloud environment in [182, 269, 144, 120].

Flush-Reload attacks: This type of attacks assumes identical memory pages can be shared among different VMs, so that the adversary and victim VMs may share the same pages containing cryptographic code or data. The adversary carefully selects a set of cacheline-sized, -aligned memory blocks from these shared pages. Then he also conducts two stages repeatedly: in the FLUSH stage, the adversary flushes the selected blocks out of the entire cache hierarchy (e.g., using the *clflush* instruction). Then it waits for a fixed interval in which the victim might issue the critical instructions and fetch them back to the caches. In the RELOAD stage, the adversary reloads these memory blocks into the caches and measures the access time. A short access time for one memory block indicates a cache hit, so this block has been accessed by the victim during the interval. By repeating these two stages the adversary can obtain traces of the victim's memory accesses and deduce the confidential data. This FLUSH-RELOAD technique was first proposed in [109], and further demonstrated in different virtualized platforms with different variants [122, 270, 107, 106].

5.1.1.2 Defenses Against Side-channel Attacks

Previous studies propose to defeat cache-based side-channel attacks in one of these ways:

Constant access time: A solution often suggested by software researchers is to achieve constant time access or constant time encryption/decryption for any key and any plaintext data. This is often very hard to achieve in practice, due to variability in cache time and other hardware features. One brute-force way is to disable the cache mechanism [168]. This makes each access time constant and there is no information leakage via timing differences. However, this will incur a huge unacceptable performance degradation to applications.

Program transformation: Past work [71, 74, 181, 151] designed methods to automatically transform a program at compiling time to eliminate secret-dependent control flows or data flows, which could cause side-channel vulnerabilities. These approaches attempted to eliminate leakage sources of all timing channels, and usually yielded high performance overhead.

Partitioning caches: One straightforward approach is to prevent the cache sharing by dividing the cache into different zones by sets or ways for different VMs. This can be achieved by hardware [238, 83, 141, 250] or software methods [200, 131, 275].

Randomization: This idea is to add randomization to the attacker's measurements, making it hard for him to get accurate information based on his observations. This includes random memory-to-cache mappings [238, 239], randomized cache prefetching [142], timers [229, 139] and cache states [271].

Avoiding co-location: New VM placement policies were designed [113, 36] to reduce the co-location probability between victim and attacker VMs. Zhang et al. [272] and Moon et al. [159] frequently migrated the VMs to add difficulty of VM co-location for the attackers.

These approaches, when applied in the cloud setting, require significant modification of computing infrastructure, and thus are less attractive to cloud providers for practical adoption. In our study, we aim to build atop existing cloud framework a lightweight side-channel attack detection system to detect, and then mitigate, the attacks as they take place, and doing this without modifying guest OS, hypervisor or hardware.

5.1.1.3 Intrusion Detection Using Hardware Performance Counters

Hardware performance counters are a set of special-purpose registers built into x86 (e.g., Intel and AMD) and ARM processors. They work along with event selectors which specify certain hardware events, and update a counter after a hardware event occurs. Most modern processors provide a Performance Monitor Unit (PMU) that enables applications to control performance counters. One of the basic working modes of PMUs is the interrupt-based mode. Under this working mode, an interrupt is generated when the occurrences of a given event exceed a predefined threshold or a predefined amount of time has elapsed. Therefore, it makes both event-based sampling and time-based sampling possible.

Performance counters were originally designed for software debugging and system performance tuning. Recently, researchers exploited performance counters to detect security breaches and vulnerabilities [150, 257, 245, 234, 78, 220, 38, 235]. The intuition is that the performance counters can reveal programs' execution characteristics, which can further reflect the programs' security states. Besides, performance counter detection introduces negligible performance overhead to the programs. Related to, but different from our work are signature-based side-channel attack detection methods using performance counters [78, 68, 114], which, unfortunately, could be easily evaded by smarter attackers by slightly changing the cache probing pattern.

5.1.2 Threat Model and Assumptions

We focus on cross-VM side-channel threats in public IaaS clouds based on Last Level Caches (LLC) that are shared between processor cores. We assume the adversary is a legitimate user of the cloud service who is able to launch VMs in the cloud and has complete control of his VMs. We further assume the attacker is able to co-locate one of his VMs on the same server as the victim VM, and the two VMs will share the same processor package, thus the LLC, with non-negligible probability. We consider both Prime-Probe side-channel attacks and Flush-Reload side-channel attacks, which represent all known LLC side channels in modern computer systems.

5.2 Detection Method

5.2.1 Design Challenges and Overview

There are several technical challenges in the application of traditional detection techniques, like signature-based detection and anomaly-based detection, to side-channel attacks.

Signature-based detection approaches are widely used techniques in detecting network intrusion and malware, by comparing monitored application or network characteristics with pre-identified attack signatures. Similarly, to detect side-channel attacks, signatures of side-channel attacks must be generated from all known side-channel attack techniques and used to compare with events collected from production systems. Prior work [78, 68] have preliminarily explored such ideas. Particularly, Demme et al. [78] demonstrated in a simplified experiment setting that classification algorithms could successfully differentiate normal programs from Prime-Probe attack programs. The advantage of this approach is that they have a high true positive rate in detecting known attacks. However, such a detection method is very fragile and easy

to evade by clever attackers. It also fails to recognize unknown attacks, with only subtle changes from existing ones. For instance, the attacker can change the memory access pattern (e.g., sequential order, access frequency) in a PRIME-PROBE attack to evade signature-based detection.

In anomaly-based detection, the normal behaviors of benign applications are modeled and any substantial deviation from such models are detected as attacks. To detect side-channel attacks using such techniques, one can build models for benign application behaviors. Then, for each VM to be monitored, we check if its behaviors conform to the models in the database. Compared to signature-based detection, anomaly-based detection can potentially identify "zero-day" attacks in addition to known ones. However, the difficulty of applying the anomaly-based approach to side-channel attacks stems from the challenge of precisely modeling benign application activities. Cache side-channel attacks resemble benign memory intensive applications, and therefore they are difficult to differentiate. False positive or false negative rates can be extremely high due to imprecise application behavior modeling. We are not aware of successful side-channel detection methods that are based on anomaly detection.

CloudRadar combines both anomaly-based and signature-based techniques to detect side-channel attacks. The only features used by *CloudRadar* are hardware event values read from the Hardware Performance Counters available in commercial processors. The key insight that motivates *CloudRadar* is derived from prior research in side-channel attacks: in cache side-channel attacks, to effectively exfiltrate secret information from the victim's sensitive execution, the attacker needs to repeatedly conduct side-channel activities (e.g., PRIME-PROBE or FLUSH-RELOAD) and deduce cache usage based on the execution time of his own interferring memory activities. This enables him to make inferences on the victim's cache usage by looking at the statistics of his own cache hits and cache misses. As such, the attacker's cache use patterns must be different when the victim executes sensitive operations so that the

attacker can differentiate them in his own analysis. Our intuition is that *if such distinction can be detected by the attacker using timing channels, it can be detected by the cloud provider using Hardware Performance Counters.* Thus *CloudRadar* monitors all suspected VMs running on a cloud server and collects their cache use patterns using Hardware Performance Counters. Once anomalies in cache use patterns are detected by *CloudRadar*, these anomalies will be correlated with the sensitive operations (usually cryptographic operations) in the co-located protected VM (i.e., VMs owned by customers paying for such services). Strong correlation will serve as a good indicator of cache-based side-channel attacks.

Two key *technical challenges* in our design are (1) identifying the execution of the protected VM's sensitive operations without asking the customers to modify their applications, and (2) detecting untrusted VM's abnormal cache use patterns. We aim to achieve both by using only values read from Hardware Performance Counters. To do so, we *first* propose to use signature-based techniques to detect sensitive applications of the protected VM, because they are conducted by honest parties and will not attempt to evade detection intentionally—a perfect target of signature-based detection techniques. *Second*, we propose to use anomaly-based detection techniques to detect abnormal cache patterns due to side-channel activities, as they are expected to vary due to different attack techniques and intensity. As side-channel attack detection is done via correlation with sensitive operations, false positives that are common challenges in anomaly detection techniques can be ruled out. We will highlight our design of these two components in Section 5.2.2 and 5.2.3.

5.2.2 Signature Detection of Cryptographic Applications

As sensitive operations that are targeted by side-channel attacks are usually cryptographic operations, we consider detection of cryptographic applications in this section. Our working hypothesis here is that all cryptographic applications have unique signa-

tures that can be easily identified by Hardware Performance Counters. In this section, we validate our hypothesis by a set of preliminary experiments.

5.2.2.1 Cryptographic Signature Generation

To generate signatures for detecting cryptographic applications, we need to select a proper hardware performance feature that uniquely characterizes a certain execution phase [198] of such applications.

Feature selection. Modern processors allow a large number of events to be measured and reported by Hardware Performance Counters. The signature generated from a proper hardware event should satisfy two requirements: (1) *uniqueness*: the signatures of different applications should be highly distinguishable; (2) *repeatability*: the signature of a cryptographic application should be identical each time it is generated, regardless of the platform's configurations and the inputs.

We consider different events from three main categories: CPU events, cache events and kernel software events. We use the Fisher Score [85] to test the *repeatability* and *uniqueness* of these events in identifying cryptographic applications. The Fisher Score is one of the most widely used methods to select features quickly. It finds the optimal feature so that the distances between data points in the feature space of different classes are maximized, while the distances between data points in the same class are minimized.

To test the *uniqueness* of an event, we use Hardware Performance Counters to measure the number of this event every 100μs during the execution of six representative cryptographic applications (i.e., asymmetric cryptography: ElGamal and DSA from GnuPG; symmetric cryptography: AES and 3DES from OpenSSL; hash: HMAC from OpenSSL and SHA512 from GnuPG). We select 10 consecutive counter values (collected from $10 \times 100\mu$s) from the beginning of each application to form a timing sequence as one training data point. We repeat this 100 times for each cryptographic

application. For each hardware event we considered, we calculate the Fisher Score using 600 training data points from the six cryptographic applications to test the uniqueness of this event in distinguishing different applications. Table 5.1 (Inter-class F-Score column) shows the results. Note a larger inter-class F-Score indicates a better *uniqueness* of this event. We can see some CPU events (instructions, branches and mispredicted branch instructions) and cache events (L1I fetch misses) are better candidates for signature generation. They vary significantly for different cryptographic applications. The events that rarely happen during the cryptographic execution (e.g., context switches and page faults), or remain identical for different cryptographic applications (e.g., CPU cycles or clock) fail to satisfy the *uniqueness* requirement.

To test the *repeatability* of an event, we repeat the above experiments on three servers with different hardware and software configurations. For each cryptographic application, we calculate the Fisher Score from 300 training data points collected from three servers. Table 5.1 (Intra-class F-Score column) shows the average Fisher Score of the six cryptographic programs. A smaller Intra-class F-Score indicates the signature with this event is more repeatable. We are able to find some events with good repeatability (e.g., instructions, branches and mispredicted branch instructions).

Based on the inter-class and intra-class Fisher Scores, we can choose the features with both good uniqueness and repeatability for signature matching. For instance, we can use *instructions*, *branch instructions* and *mispredicted branch instructions* to conduct multi-feature classification. Further evaluations in Section 5.5 show one single feature (i.e., *branch instructions*) is already enough to give good accuracy. So we will collect the number of *branch instructions* as the feature to generate signatures in the following sections.

Phase selection. It has been shown in prior studies that programs run in different phases [198]. Therefore, another question we need to solve is which phase of the cryptographic application we should use to generate the signature. The selected

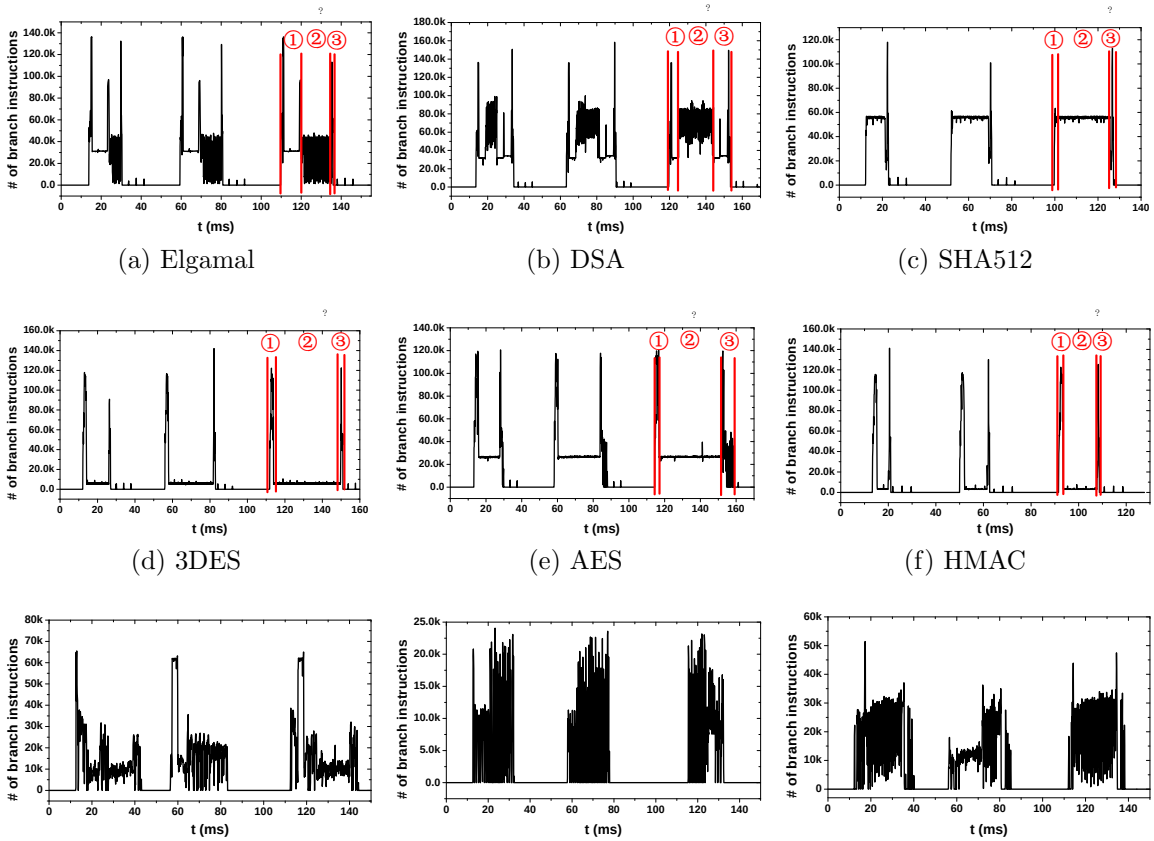
Category	Events	Inter-class F-Score	Intra-class F-Score
CPU events	instructions	1.49	0.13
	branch instructions	1.55	0.14
	mispredicted branch instructions	1.11	0.15
	CPU cycles	0.01	0.30
Cache events	L1D load accesses	0.37	0.72
	L1D load misses	0.69	0.42
	L1I fetch misses	1.14	0.20
	LLC load accesses	0.79	0.31
	LLC load misses	0.05	0.36
	iTLB load accesses	0.55	0.27
	iTLB load misses	0.23	0.21
	dTLB load accesses	0.22	0.63
	dTLB load misses	0.36	0.62
Software events	context switches	0.00	0.00
	page faults	0.00	0.00
	CPU clock	0.01	0.50

Table 5.1: Fisher Scores for different events.

phase should be able to distinguish cryptographic applications from non-cryptographic applications. It should also be independent of the inputs.

We conducted the following experiments: we ran the same six cryptographic applications as above. For each cryptographic application, the cryptographic keys and input message (for signing or encryption) are randomly chosen each time the applications are executed. We exploit the Hardware Performance Counters to record the number of *branch instructions* taking place in the program within 100μs windows. Figure 5.1 shows the profiling results for each cryptographic application. For comparison, we also show the profiling results for three non-cryptographic applications: Apache, Mysql and the Network File System (NFS).

We observe that the cryptographic applications have different behaviors from the non-cryptographic ones. Each cryptographic application exhibits three distinguishable stages, labeled in Figure 5.1. (1) The first stage initializes the program and variables. Specifically, it analyzes the application's parameters, allocates buffers for the input and output messages, retrieves keys from passphrase or salts, and sets up the cipher context. This stage does not depend on the inputs. (2) The second stage computes the



(a) Elgamal (b) DSA (c) SHA512

(d) 3DES (e) AES (f) HMAC

(g) Apache (h) Mysql (i) NFS

Figure 5.1: Signatures of different applications based on the number of branches

cryptographic operations (e.g., multiply or square operations, checking lookup tables, etc.), the characteristics of which are input dependent: the duration of this stage is linearly related to the length of the plaintext/ciphertext, and the pattern depends on the values of the cryptographic key and the plaintext/ciphertext blocks. (3) The last stage ends the application, frees the memory buffer and reports the results. We chose the first stage as the signature to represent a crypto application, because it is input independent. The Fisher Score in Table 5.1 were also generated for this stage.

5.2.2.2 Cryptographic Application Detection

To detect the execution of the sensitive applications, *CloudRadar* only requires the customers to provide the signature generated offline using Hardware Performance

Counters (not necessarily on the same hardware) or simply the executables for the service provider to generate the signature. At runtime, *CloudRadar* keeps monitoring the protected VM using the same set of Hardware Performance Counters. It then compares the data points collected at runtime with the signature of the cryptographic application. If a signature match is found, *CloudRadar* will assume the cryptographic application is being executed by the protected VM (In fact, our evaluation in Section 5.5 shows high fidelity of this approach).

Because the cryptographic signatures and runtime measurements are temporal sequences of performance counter values, we cast the signature detection problem as a time series analysis problem: i.e., measuring the similarity between the two sequences that represent the signature and the runtime measurement, respectively. We adopt the Dynamic Time Warping (DTW) algorithm [187] to calculate the distance between the two sequences. DTW is able to measure the similarity between temporal sequences which may vary in speed: it tries different alignments between these sequences and finds the optimal one that has the shortest distance. This distance is called the DTW distance. We chose the DTW algorithm because the runtime sequence may be slightly stretched or shrunk due to the difference of the computing environment (e.g., CPU models, running speed, interruption, etc.). DTW is powerful enough to find the similarity between two temporal sequences even with distortion.

Algorithm 5.1 shows how to calculate the DTW distance between the signature sequence (s, of length n) and measurement sequence (p, of length m). We set an adjustable locality constraint parameter (w) to reduce computing time complexity. We consider a $n \times m$ matrix, where the (i, j) element of this matrix corresponds to the distance between $s[i]$ and $p[j]$. Then we retrieve the optimal path from (0, 0) to (n, m) through the matrix that has the minimal total cumulative distance. This warping path can be found using a dynamic programming method, and the minimal total

Algorithm 5.1: Calculate the normalized DTW distance between signature and measurement sequences

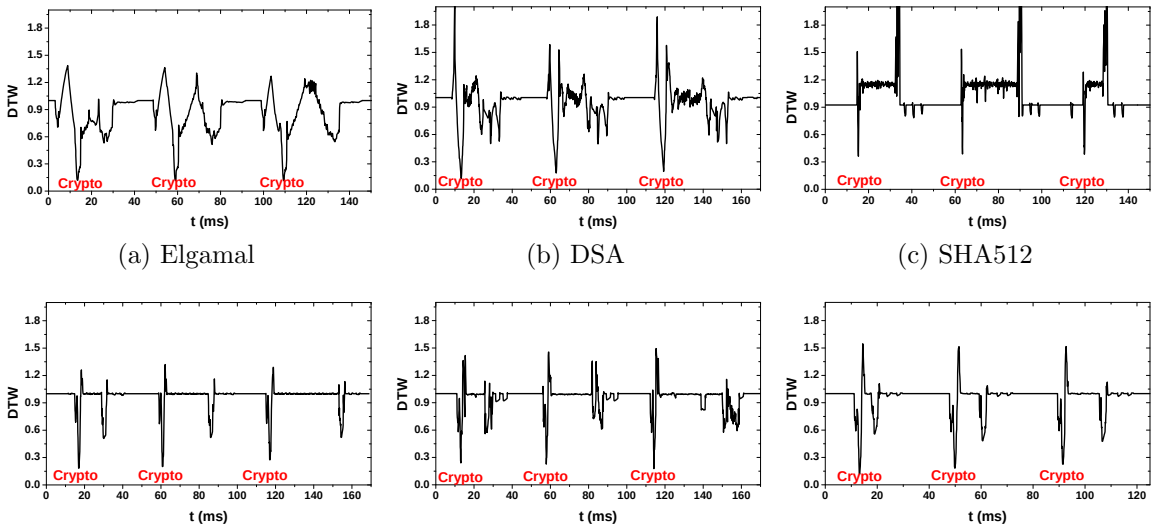
```
   Input:
1      s[1, ..., n]                  /* signature sequence */
2      p[1, ..., m]                  /* measurement sequence */
3      w                             /* locality constraint */

4  begin
5      /* calculate DTW between s[] and p[]*/
6      D[0, ..., n][0, ..., m]
7      w = max(w, |n − m|)
8      for i = 0 to n do
9          for j = 0 to m do
10             D[i][j] = +∞
11         end
12     end
13     D[0][0] = 0
14     for i = 1 to n do
15         for j = max(1, i-w) to min(m, i+w) do
16             cost = |s[i] − p[j]|
17             D[i][j] = cost + min(D[i-1][j], D[i][j-1], D[i-1][j-1])
18         end
19     end

20     return D[n][m]/∑_{i=1}^{n} |s[i]|
21 end
```

cumulative distance is the DTW distance. Finally, we normalize the DTW distance to the distance between the signature sequence and origin.

Figure 5.2 shows the normalized DTW distance of different cryptographic programs. We observe that occurrence of cryptographic programs yields very small DTW distances, which indicates a signature match. We defer a more systematic evaluation of the signature-based cryptographic program detection technique to Section 5.5.

5.2.3 Anomaly Detection of Side-channel Activities

The cache usage patterns that *CloudRadar* monitors for anomaly detection are characterized by the *cache hit* count and the *cache miss* count measured by the Hardware Performance Counters: In Prime-Probe side-channel attacks, the attacker Probes certain cache sets and measures if there are cache misses via timing the accesses to the sets after the victim executes. It is expected that cache misses will be higher than normal when the protected VM executes the cryptographic operations, since cache misses will be the tell-tale signal for the attacker to detect these operations in the first



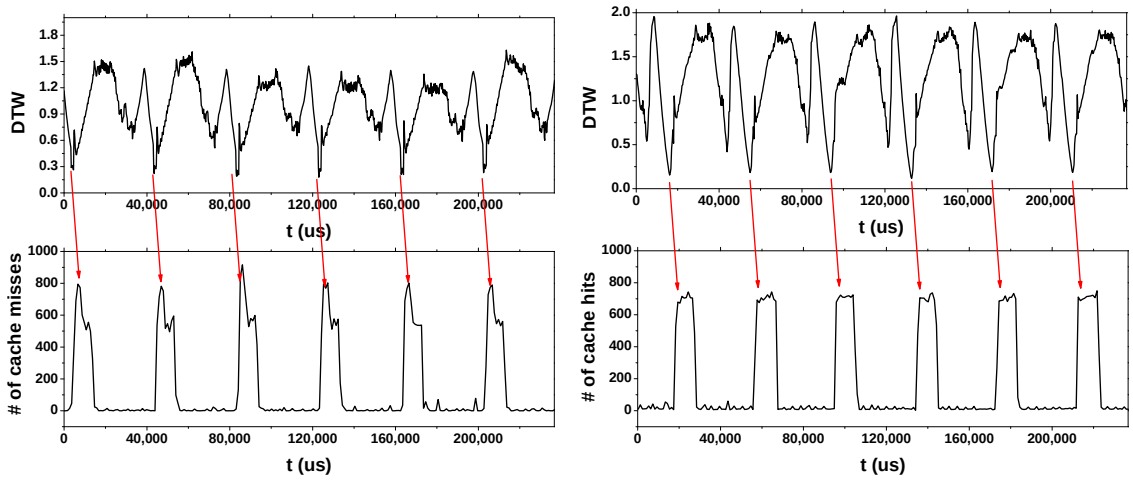
(a) Elgamal (b) DSA (c) SHA512

(d) 3DES (e) AES (f) HMAC

Figure 5.2: DTW distances of different cryptographic programs. The lowest distance indicates a signature match.

place. In Flush-Reload side-channel attacks, the attacker Reloads certain cache lines and tries to detect cache hits. Cache hits should occur more frequently during the protected VM's sensitive operations.

To validate this hypothesis, we conducted a set of experiments to show that abnormal cache activities in the untrusted VM can be correlated with the protected VM's sensitive operations. We first consider a Prime-Probe attack against the ElGamal cipher [144]. Figure 5.3a shows the DTW distance (low distance indicates a signature match) between the runtime sequence and the signature sequence observed on the protected VM (top figure), correlates with the attacker VM's high cache miss counts (bottom figure). We next consider a Flush-Reload attack against the RSA cipher [255]. Figure 5.3b shows the low DTW distance of the protected VM correlates with the high cache hit counts of the attacker VM. We align the top figures and the bottom figures according to timestamps. Strong correlation can be observed in both set of experiments, which suggest that this method can be used for side-channel attack detection.



(a) Prime-Probe attack (b) Flush-Reload attack

Figure 5.3: Monitoring cache activities under side-channel attacks

To describe our detection algorithm more precisely, when *CloudRadar* detects that the victim VM starts executing crypto applications (a low DTW distance), two short sub-sequences are selected from the entire monitored runtime sequences in the untrusted VMs: $\mathbb{S}$, data points of size w before the DTW distance reaches its minimum, and $\mathbb{S}'$, data points of size w after the minimum points of DTW distance, where w is a parameter of the detection system. If *CloudRadar* detects that the difference between any value in $\mathbb{S}'$ and any value in $\mathbb{S}$ is larger than a pre-determined threshold T, *CloudRadar* will raise an alarm of a possible side-channel attack. This rule can be formally expressed in Equation 5.1. We will further evaluate this side-channel detection method in Section 5.5.

$$\text{Alarm: } v' - v > T, \quad \forall v \in \mathbb{S},\ v' \in \mathbb{S}' \tag{5.1}$$

5.3 Mitigation Methods

When a side-channel adversary is detected, *CloudRadar* will cut off the side channel by migrating the identified adversary VM to a different processor socket. There can

be other ways to mitigate the threat. For instance, the cloud provider can migrate the adversary VM to a different server when the server has only one processor socket. This triggered migration will be more effective than past work [272, 138, 159]. The cloud provider can also partition the LLC between the victim VM and the adversary VM. It can even shut down the adversary VM and block the attacker's account. Side-channel detection makes mitigation solutions easier.

5.4 Architecture

5.4.1 Architecture Overview

CloudRadar is provided by the cloud operator as a security service to the customers who are willing to pay extra cost for better security, as in the *CloudMonatt* cloud framework [124, 262]. Figure 5.4 shows the architecture of *CloudRadar*, and the workflow of detecting side-channel attacks. We implement *CloudRadar* into the *CloudMonatt* framework (Details about the integration of this defense in *CloudMonatt* will be illustrated in Section 7.1). Three types of servers, the Cloud Controller, the Attestation Server and regular cloud servers, are relevant to our discussion.

The Cloud Controller is the cloud manager, responsible for taking VM detection requests and servicing them for each customer. The Attestation Server is a dedicated server to manage the provided security services and coordinate the interaction between the Cloud Controller and the cloud servers. The `Signature Database` is used to store signatures of crypto programs.

CloudRadar's functionality within a cloud server is tightly integrated with the host OS. As shown in Figure 5.4, *CloudRadar* consists of three modules, with each one running on a dedicated core. The `Victim Monitor` is responsible for collecting the protected VM's runtime events, which will be fed to `Signature Detector` to detect the cryptographic programs using our signature-based technique; The `Attacker Monitor`

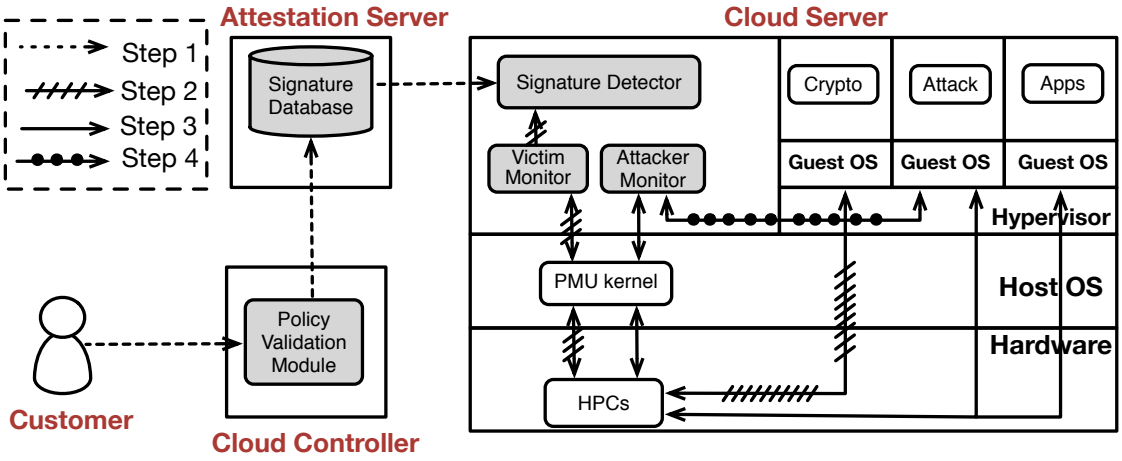


Figure 5.4: Architecture Overview of *CloudRadar*

is responsible for collecting cache activities of the other VMs, using anomaly-based detection approach to identify side-channel attackers. We used the Linux *perf_event* kernel API for the PMU to manage the Hardware Performance Counters, therefore no change is needed to the hypervisor itself.

5.4.2 System Operations

CloudRadar includes four steps, as shown in Figure 5.4 with different paths. Each step is described below:

Step 1: generating cryptographic signature. In this step, the customer who seeks side-channel detection services for his protected VM can indicate to the Cloud Controller what sensitive applications to be protected, by providing the signatures generated offline using Hardware Performance Counters (not necessarily on the same hardware) or simply the executables. Then the Cloud Controller will run these crypto programs on a dedicated server with the same configuration as the Cloud Server that hosts the protected VM, and use Hardware Performance Counters to generate the signatures for the customer. The signatures will be stored in the `Signature Database`

in the Attestation Server for future reference. They will also be sent to the cloud server that hosts this VM.

Step 2: detecting cryptographic applications. This step takes place at run-time. In this step, the `Victim Monitor` monitors the protected VM using Hardware Performance Counters. It periodically (e.g., every 100μs) records the event counts (e.g., *branch instructions*) as a time sequence, while the `Signature Detector` keeps comparing the most recent window of data points in the sequence with the signature. If a signature match is found, the `Signature Detector` can identify the protected VM is performing a cryptographic application, and signal this result to the `Attacker Monitor`.

Step 3: monitoring cache activities. This step happens concurrently with Step 2. The `Attacker Monitor` exploits Hardware Performance Counters to monitor all untrusted VMs simultaneously. One challenge is that not enough performance counters are available on the servers to monitor all VMs, if this number is large: most of the Intel and AMD processors support up to six counters, and the number of counters does not scale with the number of cores. So when there are a lot of VMs on the server, the `Attacker Monitor` cannot monitor them concurrently.

To solve this problem, we use a time-domain multiplexing method: the `Attacker Monitor` identifies active vCPUs that share LLC with the protected VM as the *monitored* vCPUs, and then measures each of them in turn. Specifically, in each period, the `Attacker Monitor` uses a kernel module to check the state and CPU affinity of each vCPU of each VM from its *task_struct* in the kernel. The `Attacker Monitor` marks the vCPUs in the *running* state that are sharing the same LLC with the protected VM as *monitored*. Then it sets up Hardware Performance Counters to measure each *monitored* vCPU's cache misses and hits in turn. When the `Attacker Monitor` is notified that a cryptographic application is happening in the protected VM, it will compare each *monitored* vCPU's cache misses and hits before and during

the cryptographic application, as specified in Section 5.2.3. If one vCPU has an abrupt increase in the number of cache misses or hits during the cryptographic application, the `Attacker Monitor` will flag an alarm.

Step 4: eliminating side channels. Once the `Attack Monitor` notices that one co-tenant VM has abnormal cache behavior exactly when the protected VM executes cryptographic applications, it will raise alarm for side-channel attacks. It will adopt the methods from Section 5.3 to cut off the cache side channels. In addition, the Attestation Server will report this incident to the Cloud Controller for further processing, such as shut down the malicious VM or eventually block the attacker's account.

5.5 Evaluation

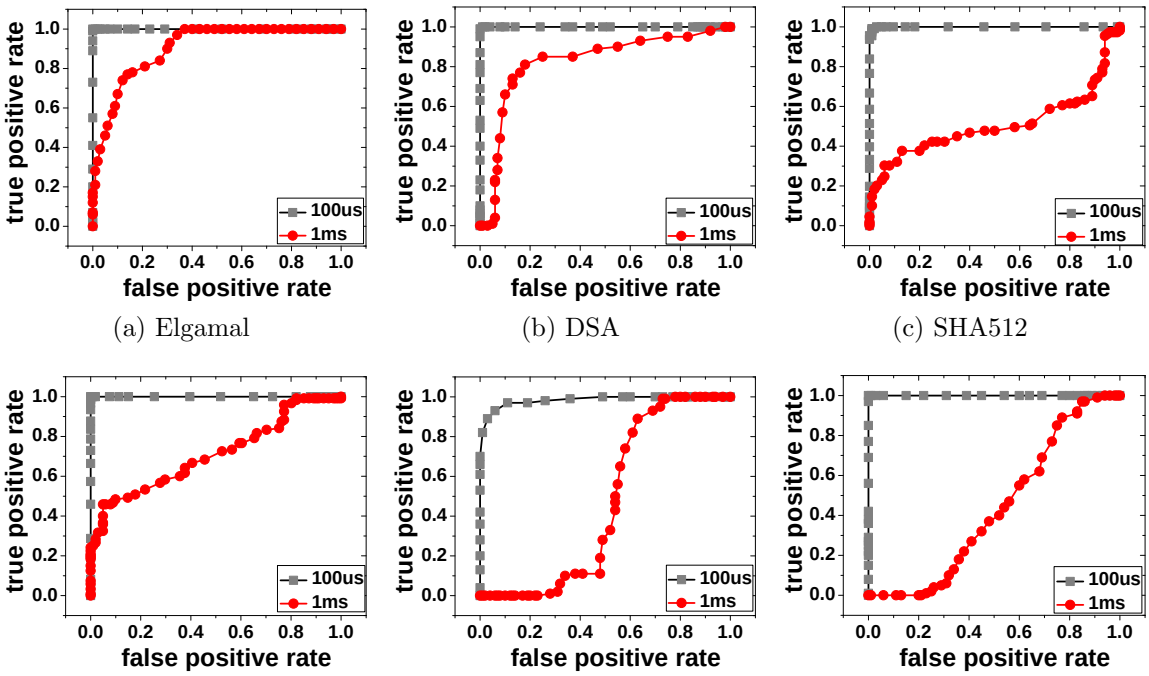
We used four servers to evaluate the security and performance of *CloudRadar*. A Dell R210II Server (equipped with one quad-core, 3.30GHZ, Intel Xeon E3-1230v2 processor with 8GB LLC) is configured as the Controller Server as well as the Attestation Server. Two Dell PowerEdge R720 Servers are deployed as the host cloud servers: one is equipped with one eight-core, 2.90GHz Intel Xeon E5-2690 processor with 20GB LLC; one is equipped with two six-core, 2.90GHz Intel Xeon E5-2667 processors with 15GB LLC. We also use another Dell 210II server as the client machine outside of the cloud system to communicate with cloud applications. Each VM in our experiments has one virtual CPU, 4GB memory and 30GB disk size. We choose Ubuntu 14.04 Linux, with 3.13 kernel as the guest OS.

5.5.1 Detection Accuracy

We measure the detection accuracy of cryptographic signature detection and cache anomaly detection.

Accuracy of cryptographic application detection. To detect a cryptographic application, we used the *branch instruction* counts as the signature. We consider the detection of a cryptographic application as a binary classification, and measure its true positive rate and false positive rate. True positive happens when a cryptographic application is correctly identified as such. We used the same six cryptographic applications from Section 5.2.2.1. *CloudRadar* first generates a signature for each application. In the detection phase, the victim VM generates a random memory block and feeds it to the crypto application. We run the experiment 100 times, and measure the number of times *CloudRadar* can correctly identify the cryptographic program under different thresholds. False positive is defined as non-cryptographic applications identified as cryptographic programs. We select 30 common linux commands and utilities which do not contain cryptographic operations. In each experiment the victim VM runs these commands in a random order. We repeated the experiment 100 times and measured the number of times false positives take place under different thresholds. We plot the ROC (Receiver Operating Characteristic) curves to show the relationship between the true positive rate and false positive rate.

We explored the effect of changing performance counter sampling granularities (i.e., interval between taking performance counter values) on detection accuracy. We choose two different sampling granularities: 100μs and 1ms. Figure 5.5 shows the ROC curves of the six cryptographic applications under these two granularities. From this figure we can see 100μs gives better accuracy than 1ms: *CloudRadar* can achieve close to 100% true positive rate with zero false positive rate when the DTW threshold is set between 0.3 and 0.4. For 1ms, Elgamal and DSA application can be detected with less accuracy, while SHA512, AES, HMAC and 3DES cannot be differentiated from non-cryptographic applications with reasonable false positive and false negative rates at the same time.



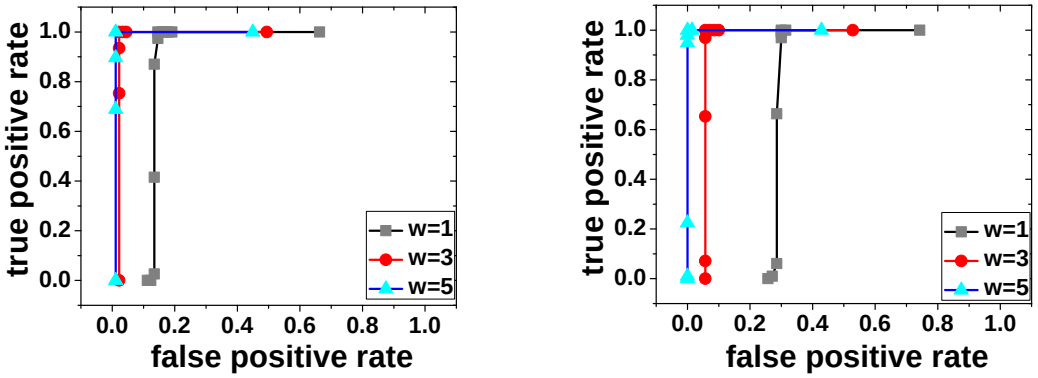
(a) Elgamal (b) DSA (c) SHA512

(d) 3DES (e) AES (f) HMAC

Figure 5.5: ROC curve of crypto detection under two sampling intervals.

The optimal sampling granularity depends on the length of the cryptographic application's initialization stage: if the sampling period is much shorter than the initialization stage, the signature will contain more data points, thus yielding more accurate results. In our experiments, the initialization stages of Elgamal, DSA, SHA512, AES, HMAC and 3DES last for 10ms, 5ms, 1.6ms, 2ms, 2ms and 2ms respectively. So a granularity of 100μs can give good results for all the six applications, while 1ms granularity performs worse, especially for SHA512, AES, HMAC and 3DES whose signatures only contain two data points.

Accuracy of cache side-channel attack detection. We measured the true positive rate and false positive rate of side-channel attack detection. True positive is the case where side-channel attacks are correctly identified. We tested the Prime-Probe attack [144] and Flush-Reload attack [255]. False positive is defined as benign programs that are falsely identified as attacks. We select different common linux



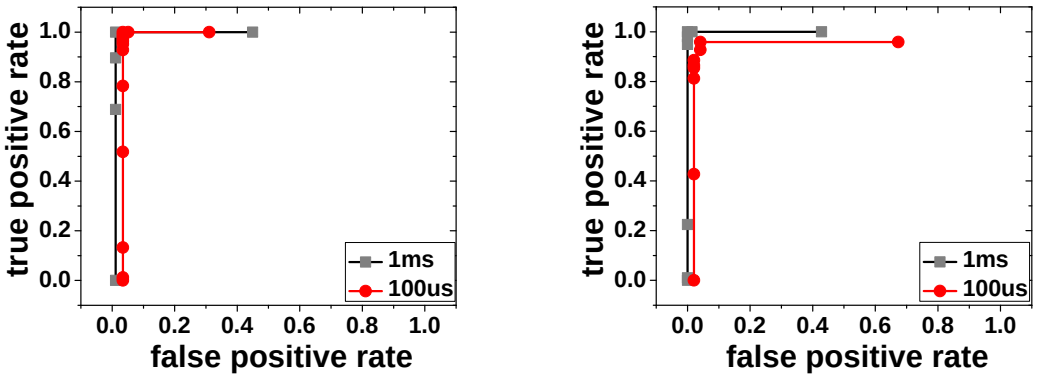
(a) Prime-Probe attack (b) Flush-Reload attack

Figure 5.6: ROC curve of attack detection under different window lengths.

commands and utilities as benign applications. We changed the threshold and drew the ROC curves to show the relations between true positive and false positive rates.

We first considered different window sizes w for $\mathbb{S}$ and $\mathbb{S}'$ (Setion 5.2.3). Figure 5.6 shows the attack detection accuracy under three window sizes: $w = 1$, 3 and 5. In these experiments, we set the sampling granularity as 1ms (this sampling rate is different from that of signature detection). From these results we see that *CloudRadar* has an excellent true positive rate: with appropriate thresholds ($100 \sim 300$ events per 1ms), the true positive rate can be 100%. However, it also has false positives. When $w = 1$, the false positive rate can be as high as $20\% \sim 30\%$. False positives are caused by the coincidence that a benign application experiences a phase transition at exactly the same time as the victim application executes a crypto operation. *CloudRadar* will observe changes in the benign application's cache behavior and think it is due to interference with the victim. Then it will flag this benign VM as malicious. We can increase w to reduce the false positive rate without affecting the true positive rate: when $w = 5$, the false positive rate is close to 0 while true positive rate is 100%.

We also tested different sampling granularities. Figure 5.7 shows the ROC curves of detecting two attacks under two different sampling intervals: 1ms and 100μs. The window size is 5 data points. We can see the 1ms interval is better than the 100μs.



(a) Prime-Probe attack (b) Flush-Reload attack

Figure 5.7: ROC curve of attack detection under different sampling intervals.

This is because when the sampling interval is small, the number of cache events occurring within a sampling period is comparable to the measurement noise. So the measurements under this sampling granularity are not very accurate. It is interesting to note that we need different granularities to sample the victim's CPU events (100μs) and attacker's LLC events (1ms). This is because the victim's CPU events occur more frequently than the attacker's LLC events. So at the granularity of 100μs, sampling the victim can give finer information, while sampling the attacker will introduce large Signal-to-noise ratio (SNR), making the results less accurate.

5.5.2 Performance

Detection latency. Table 5.2 reports the detection latency of *CloudRadar* under different window sizes w and sampling granularities. This detection latency is defined as the period from the time the victim VM starts to execute sensitive operations (i.e., start of the second stage in Figure 5.1) to the time an alarm for side-channel attacks is flagged. We see that *CloudRadar* can identify the attack on the order of milliseconds. Considering side-channel attackers usually need at least several cryptographic operations to steal the keys, this small latency can achieve our *real-time*

(μs)	granularity = 1ms			granularity = 100μs		
	$w = 1$	$w = 3$	$w = 5$	$w = 1$	$w = 3$	$w = 5$
Prime-Probe	1021.41	3065.86	5110.04	120.49	361.97	603.03
Flush-Reload	1021.50	3064.38	5107.57	122.48	363.27	605.30

Table 5.2: Detection latency (μs) under different window sizes and sampling intervals

Workload	Description	Metric
Data Analytics	Perform machine learning analysis in a hadoop system	1/throughput
Data Caching	Stress a Memcached system	latency
Data Serving	YCSB benchmarks on Apache Cassandra system	1/throughput
Graph Analytics	Perform graph analysis using PowerGraph	completion time
Media Streaming	Stress a Darwin Streaming Server using Faban	1/throughput
Software Testing	Perform parallel software testing using Cloud9	completion time
Web Search	Stress the Nutch search engine using Faban	latency
Web Serving	Stress the Apache, Nginx and Mysql servers using Faban	1/throughput

Table 5.3: CloudSuite Benchmarks

design goal. We also observe that smaller window sizes and finer granularity can effectively reduce the detection latency, at the cost of slightly lower accuracy.

Performance overhead. We selected a mix of benchmarks and real-world applications to evaluate the performance of *CloudRadar*. Our benchmarks can be categorized into three types: (1) crypto programs (AES, SHA, HMAC, Blowfish and MD5 from OpenSSL; ElGamal, RSA and DSA from GnuPG); (2) CPU benchmarks (mcf, gobmk, omnetpp, astar, soplex and lbm from SPEC2006; canneal and streamcluster from PARSEC); (3) cloud applications from CloudSuite [89] (data analytics, data caching, data serving, graph analytics, media streaming, software testing, web searching and web serving) (see Table 5.3).

We tested the performance penalty due to *CloudRadar* and show the normalized performance of each of the benchmark applications in Figure 5.8 (results are average of 5 runs, error bars show one standard deviation). The results suggest *CloudRadar* has little impact on the performance of the monitored VM: even in the worst case, performance overhead is within 5%.

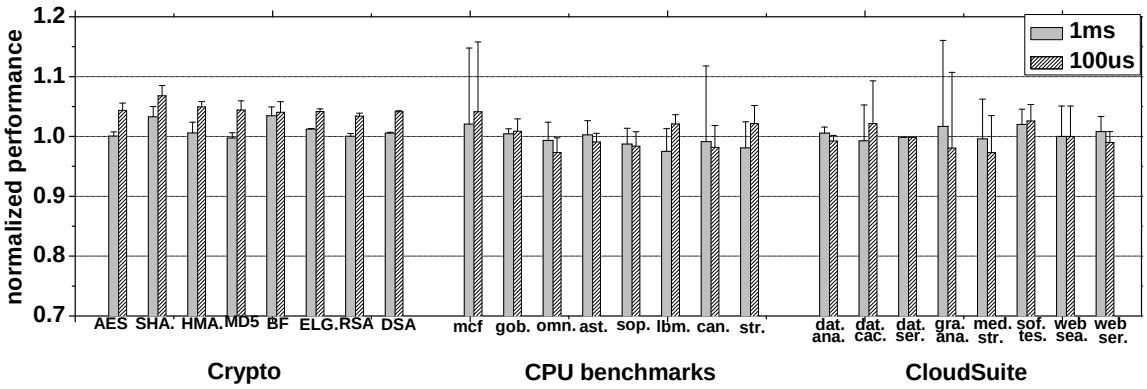


Figure 5.8: Performance of different benchmarks under *CloudRadar*

5.6 Discussions

5.6.1 Detecting Other Side Channels

One can extend *CloudRadar* to detect cache-based side-channel attacks in other cloud models (e.g., PaaS [270]), or in non-virtualization environments. The only change we need to make is to use Hardware Performance Counters to monitor the processes or threads instead of VMs. Besides, this method can be applied to other micro-architectural side-channel attacks that exploit resource contention. We can use Hardware Performance Counters to count the corresponding events that the attacker uses to retrieve information. For instance, we can monitor the DRAM bandwidth event to detect the DRAM side-channel attacks in [236]. Generalization of this method beyond cache-based side-channel attacks can be future work.

5.6.2 Potential Evasive Attacks

There can be potential evasive attacks against *CloudRadar*. To evade the detection of *CloudRadar*, a side-channel attacker can try to reduce the cache probing speed, so the abnormal increase in cache misses or hits may not be observed by *CloudRadar*. However, the attacker needs a much longer time to recover the keys, making side-

channel attacks more difficult and less practical. An attacker can also try to evade the detection by adding noise to *CloudRadar*'s observations. However, such noise can also blur the attacker's observations, making it more difficult to extract side-channel information. How to design efficient evasive attacks and how to detect such attacks can also be future work.

5.6.3 Limitations

CloudRadar may be limited in several aspects. First, each of its three modules (`Victim Monitor`, `Attacker Monitor` and `Signature Detector`) requires an exclusive use of one physical CPU core as they keep conducting data collection and analysis at full CPU speed. This can potentially reduce the server's capacity for hosting VMs. However, as many cloud servers today are equipped with dozens of CPU cores, the impact is not as big as one might imagine. Besides, public clouds usually have low server utilization ($< 20\%$) for preserving VMs' QoS [146]. So using three cores will not affect most VMs' performance. Second, due to the limited number of Hardware Performance Counters available in modern processors, *CloudRadar* has to multiplex the monitoring for each VM using the same counter. When the number of *monitored* vCPUs scales up, *CloudRadar* may miss attacks. We expect future generations of processors will incorporate more Hardware Performance Counters and *CloudRadar* can make use of different counters to monitor different VMs at the same time.

5.7 Chapter Summary

This chapter presents *CloudRadar*, a real-time detection system to detect cache-based side-channel attacks in clouds. *CloudRadar* leverages the existing Hardware Performance Counter feature to both monitor a victim VM's cryptographic operations and capture a potential attacker VM's abnormal behavior during this time. *CloudRadar*

is designed as a lightweight extension to the cloud system and does not require new hardware, hypervisor/OS or application modifications. The feasibility of *CloudRadar* is validated by our implementation on the OpenStack framework. Our evaluation shows *CloudRadar* can detect cache-based side-channel attacks with high fidelity, while introducing little overhead to the cloud applications.

Chapter 6

Detection and Mitigation of Integrity Vulnerabilities

In addition to binary attestation of startup integrity (Chapter 3), *CloudMonatt* can also detect and mitigate system integrity breaches during runtime. In this chapter, we show how existing techniques (e.g., virtual machine introspection) can be readily integrated into the *CloudMonatt* framework to protect VMs' system integrity. Specifically, we present *CloudGuard*, a framework that provides runtime protection of VM system integrity as a security service to customers. *CloudGuard* has the following features. First, it provides a rich set of protection options for customers. Second, it leverages the VM introspection technique to achieve more trustworthy and reliable integrity protection. Third, it offers prompt mitigation solutions once security breaches are detected, so customers do not have to worry about deploying security measures by themselves. We show a concrete implementation of *CloudGuard* on the OpenStack open source software. We select and implement several security tools from past work to enhance different aspects of the security of guest VMs. We evaluate our implementation and show that the performance overhead of the monitored VM caused by *CloudGuard* is within 8%, which is acceptable.

6.1 Background

Customers are concerned about the security of their VMs running in the remote datacenter. They expect that their VMs are protected from being compromised by remote network attackers or malicious programs in the VMs during runtime. However, with the proliferation of malware and attacks, customers' data and computations in the cloud face tremendous integrity threats. First, the large code sizes of operating systems inevitably introduce vulnerabilities that adversaries can exploit to compromise the VMs. The Common Vulnerabilities and Exposures (CVE) database [7] keeps reporting OS bugs that enable attackers to take control of the VMs. Past work [39, 261] showed that VM images in public clouds may contain lots of malware and software vulnerabilities. Second, modern attacks are designed to be more sophisticated and harder to defeat. For instance, advanced malware can leverage social engineering techniques to trick victim users into offering access to their systems [27]. Malware can easily spread in a variety of ways, e.g., phishing websites, phishing email attachments, etc. Third, cloud computing is a type of network-based computing service. Customers deliver their computation tasks and data to the remote datacenters via networks. They also deploy cloud applications for end users to use through networks. Since malware is usually delivered over networks, virtual machines in the cloud can also get infected with the malware. Once the malware intrudes into the VMs, they can compromise the VMs' data and code.

It is necessary for public cloud providers to protect the VMs' system integrity for customers. One typical solution is to install security tools (e.g., anti-virus applications, intrusion detection systems (IDS), firewalls, etc.) inside VMs to detect and defeat potential vulnerabilities. Amazon Web Services introduces Inspector [3] to realize this type of security service. Amazon Inspector requires customers to install a security agent inside their VMs. The security agent monitors the activities (e.g., network,

file system, process activities) inside the VM. It can identify threats from the CVE database, system mis-configurations, authentication vulnerabilities, insecure network protocols, etc. Microsoft Azure designed Antimalware [15] to achieve similar functions. Antimalware also requires customers to install security tools inside the VM. These tools are able to identify and remove viruses, spyware and other malicious software, and alert users when malicious or unwanted software attempts to install itself or run inside the VMs.

However, the above solutions have one big drawback: since the security tools are located in the vulnerable system, they are highly susceptible to attacks [112, 84, 219, 40, 90]. Once attackers intrude into the VM and gain root privilege, they can easily discover and compromise these security tools, making them ineffective in protecting the VM's security. To overcome this, researchers proposed to use Virtual Machine Introspection (VMI) [95] to detect vulnerabilities inside a VM. The basic idea is to move the security tools from the guest VM to the hypervisor or host OS. These tools then have root privilege to monitor the internal execution, state and data of the guest VM. They can produce more trustworthy and reliable results, as malware running in the guest OS cannot tamper with the hypervisor layer.

In this chapter, we present *CloudGuard*, a cloud architecture to protect a VM's system integrity at runtime. *CloudGuard* has several features. First, we leverage the VM introspection technique to achieve trustworthy and reliable system integrity protection. We identify the necessary architectural components and design to apply VM introspection into public clouds. We demonstrate an end-to-end prototype implementation of *CloudGuard* in OpenStack to show how the commercial cloud provider can use the VM introspection to protect customers' VMs. Second, similar to Security-on-Demand cloud frameworks [124, 262], *CloudGuard* allows customers to choose different types of protections as desired for their VMs. For instance, a customer who deploys web applications in his VM can request monitoring and controls

on network sockets. Another customer who frequently runs programs from untrusted sources can request malware scanning. Third, *CloudGuard* can automatically detect and mitigate potential vulnerabilities on behalf of customers. *CloudGuard* exploits the VM introspection functionality to actively change the VM's state and data, thus fixing the security breaches for the customers, where possible. By doing so, the cloud provider will conduct both detection and mitigation for customers. This achieves a one-click deployment of security protection for VMs and makes cloud service more attractive to security-aware customers.

To better illustrate the features of *CloudGuard*, we implement several security measures from past work to protect a guest VM from different security threats. The first one is a kernel rootkits scanner. We check the integrity of critical kernel pointers and codes at VM runtime and restore them once compromised. We also conduct cross-view validation to detect hidden processes or network sockets. The second case is anti-malware. We are able to conduct static analysis on each program's image, as well as dynamic analysis on the program's execution traces. Such information can help us identify malware using signature-based or anomaly-based detection approaches. The third case is firewalls. We can filter and control any inbound and outbound network connections to protect the VM's network activities.

6.1.1 Related Work

Garfinkel and Rosenblum [95] first proposed the method of virtual machine introspection in detecting attacks. Since then, many architectures and tools have been designed to advance the virtual machine introspection techniques. Payne et al. [169] designed XenAccess, a monitoring library for VMI on Xen. Jiang et al. [126] designed a honeypot tool based on virtual machine introspection to detect and analyze network-based attacks. It leverages the binary translation technique to intercept syscalls and collect the associated context information. Quynh and Takefuji [178] designed XenKIMONO

to detect kernel-level rootkits in Xen-based servers. Jones et al. [128] introduced Lycosid to detect hidden malicious processes. Payne et al. [170] designed Lares, to realize the event notification technique by placing hooks inside the introspected VM. Dinaburg et al. [79] designed Ether, which conducts the malware analysis using virtual machine introspection. Lengyel et al. [136] built DRAKVUF, a dynamic malware analysis system for Windows OS. It achieves fidelity and stealth using virtual machine introspection, and scalability using VM cloning. This chapter attempts to explore the possibility of applying VM introspection into public clouds, instead of designing new introspection techniques. Actually due to the rapid development of VM introspection, this technique is now quite mature. We will exploit some opensource tools and methods from past work [169, 178, 128] to show the integration of VM introspection in commercial clouds.

One big challenge of virtual machine introspection is the semantic gap between the guest OS and the hypervisor. Jiang et al. [127] cast the guest VM's view of the OS into the hypervisor to systematically reconstruct internal semantic views of a VM from the outside in a non-intrusive manner. Srinivasan et al. [204] designed a process out-grafting method, which migrated a suspect process from inside the monitored VM to a secure VM which runs the security monitoring tool. This can achieve isolation and removes the semantic gap. Dolan-Gavitt et al. [81] designed Virtuoso to automatically convert in-guest programs into out-of-guest programs that reproduced the same behaviors. Fu et al. [93] designed VM-Space Traveler, which automatically identified the critical data of the monitored VM and redirected the data from the monitored VM to a secure VM for monitoring. It also automatically converted in-guest inspection tools to an introspection tool. Carbone [59] inserted function calls into the introspected VM from the hypervisor to obtain OS information, thus bridging the semantic gap. How to eliminate the semantic gap in VM introspection is orthogonal to our work. These methods can easily be integrated into the *CloudGuard* framework.

Some work designed VMI frameworks for public clouds, which are close to this chapter. Yao et al. [254] designed CryptVMI, which encrypted introspection requests and results to achieve confidentiality. Baek et al. [37] virtualized the introspection interface and provided this to customers as a service to enhance VM security. However, these work only gave high-level system abstractions without specific use scenarios about what security protections the cloud provider can offer to customers. In this chapter, we use concrete examples and implementation to show how *CloudGuard* enables customers to select and enjoy different introspection services on demand with great flexibility.

6.2 VM System Integrity Vulnerabilities

6.2.1 Kernel-level Rootkits

The OS kernel acts as the core trusted component in a virtual machine, providing critical scheduling, memory partitioning, I/O and networking functions for applications inside the VM. Unfortunately the OS kernel may not be trustworthy, and is frequently compromised. If the kernel is compromised by malware (e.g., kernel rootkits), the whole system will become vulnerable. Hence, it is critical to protect the OS kernel of a VM.

Rootkits are malicious software that allow attackers to control the system and hide their existence from the victim users. To install a rootkit into the kernel, the attacker usually needs to first obtain root privilege. Then the attacker inserts the rootkit into a kernel module which will be loaded into the kernel, or directly writes the rootkit into the kernel memory. Once installed, the rootkit can compromise the system integrity in two ways: it presents fake information to the victim users to mask its existence; it also modifies the system's control flow to establish backdoors for the

attacker to access the system in the future without authentication. Rootkits usually employ the following techniques to achieve the above goals [178].

Modifying jump-tables. Jump-tables are widely adopted by many OS kernels. Basically a jump-table is a list of entry points that serves as addresses of kernel functions. Reference to one entry in jump-tables can be done via a numbered index. Typical examples of jump-tables are the System-call Table and the Interrupt Descriptor Table. A System-call table stores an array of function pointers, in which each pointer corresponds to a system call handler that user-space processes can use to invoke kernel functions and services. An Interrupt Descriptor Table (IDT) is used to transfer the execution of a program to special software routines that handle interrupts, which might be raised during the normal course of operation by hardware or to signal exceptional conditions, such as a page fault or a hardware failure.

A rootkit can easily hijack the jump-table, modify some function pointers in the table to redirect them to its own handlers, which intentionally conduct malicious behaviors, and then jump back to the desired system call or interrupt handler. This trick is widely used in many kernel rootkits.

Modifying kernel codes. Instead of changing the function pointers in the jump-tables, the rootkit can change the kernel functions directly. For instance, a rootkit can hijack a system call or interrupt handler by changing the first few bytes of its code to the `jmp` instruction, which will jump to the rootkit's malicious codes. By doing so the integrity of the jump-table is maintained, but the malicious codes can still be executed. This technique can evade the rootkit detectors that only check jump-table integrity.

Modifying critical objects. Besides the jump-tables or kernel codes, a rootkit can also modify some critical kernel objects. By doing so, the rootkit can hide the existence of malicious processes or network connections. For instance, the proc filesystem (procfs) is a special filesystem in Linux that establishes communication

between kernel space and user space. It presents information about OS kernel and system to the user program. It also enables users to change kernel parameters at runtime. However, a rootkit can tamper with the proc filesystem, and cause the system to present wrong information to the users, thus hiding malicious processes or network backdoors from victim users.

6.2.2 User-level Malware

In addition to kernel rootkits, malicious software in the user space can also compromise the integrity of a victim system.

Malware includes a variety of hostile software with different features. For instance, a computer virus usually spreads by inserting itself into other programs, and these infected programs will perform malicious actions on the system. A worm is a stand-alone malware program that actively transmits itself over a network to infect other systems. A trojan is a malicious program which mispresents itself to appear useful and interesting, but actually performs unauthorized actions. A spyware secretly monitors user's activity, extracts sensitive information and reports it to the hackers.

The most common way for malware to proliferate is through Internet: primarily by e-mail and the world wide web. They are usually spread by some forms of social engineering. For instance, a victim user can be tricked to download illegitimate applications repackaged to appear as normal applications, click on phishing links that download and install malware, or open email attachments with malicious files. Besides, malware can also be delivered via physical devices, e.g., infected USB sticks or CDs.

Once malware gets into the victim's system, it usually exploits security bugs in the design of the applications, plugins or operating systems to conduct malicious behaviors. These behaviors include self-replication in different parts of the file system, installing applications that capture keystokes or commandeer system resources, blocking access

to files, applications or even the system itself for ransom, bombarding a browser or desktop with ads, or breaking essential system components.

6.2.3 Network-level Application Attacks

As the network becomes increasingly important in the cloud infrastructure, the attacks on the network also increase. Network-based attacks caused by malicious or unauthorized users can cause severe disruption to the system.

Network adversaries can breach the VM's integrity in several ways. First, when the victim system deploys network-based applications or services, remote adversaries can exploit the vulnerabilities inside the target applications to further control the system. For instance, the SQL injection technique is used by the adversary to embed malicious codes into the SQL statement as the input to the SQL database. If the application does not handle the input correctly, the malicious codes will be executed to conduct malicious actions in the database, such as disclosing or tampering with existing data. Another example is Cross-Site Scripting (XSS): the adversary can inject client-side scripts into web pages, which can bypass access control policies and take control of the web applications. Second, a network-based adversary can exploit the system's vulnerabilities to intrude into the system. For instance, a server usually enables network accesses via remote connection protocols like SSH, RDP (Remote Desktop Protocol), and VNC (Virtual Network Computing). If the system uses weak authentication and passwords, the attacker can just use the brute-force method to conduct password guessing attacks and break into the system. Third, network adversaries can abuse the cloud services or compromise the VMs to deploy them as botnets to attack other systems, e.g., email spamming, Distributed Denial-of-Service attacks, etc.

6.3 Detection and Mitigation

In this section, we describe some existing methods to detect and mitigate the integrity vulnerabilities inside a VM.

6.3.1 Kernel-level Rootkits

Attacker model. We focus on the kernel-level rootkits that aim to compromise the kernel by hijacking the kernel control flow or data structures. We assume that the attacker can gain the root privilege by exploiting some OS vulnerabilities. Then the attacker is able to insert kernel modules inside the kernel space and change critical function addresses, codes, or data objects. By compromising the kernel the rootkits are able to mask the existence of malicious processes, files and network sockets.

There are various ways to defeat kernel rootkits. The first method is to protect the system from being infected with the rootkits. Since the installation of kernel rootkits requires the attacker to obtain root privileges, we can enhance the system security and reduce its vulnerabilities that the attacker can exploit to conduct privilege escalation attacks. By doing so, the attacker has no opportunity to insert the rootkits into the OS kernel. But the continuing rise in privilege escalation attacks indicates that this is hard to achieve.

An alternative solution is to protect the critical kernel data, functions and objects from being compromised by the rootkits. For example, we can set the *Non-Writable* attribute for the critical memory regions that store the jump-tables or critical kernel functions. Then the attacker is not able to modify the data inside these regions. Another defense is to set the writable regions of the system's memory as *Non-Executable*. This can prevent attackers from injecting malicious codes into the system's memory. However, if the rootkits take control of the operating system, they can eliminate these protection bits in the page table entries and make the critical memory regions

vulnerable again. Ge et al. [96] designed SPROBES to monitor if the permissions in the page table entries are compromised or bypassed by the rootkits. This can prevent rootkits from disabling the permission protections and then changing kernel data and functions stealthily.

A third defense is to monitor the victim system to detect and mitigate the integrity breach caused by kernel rootkits. Below we describe common detection and mitigation methods widely used by past work.

6.3.1.1 Detection Methods

We detect kernel rootkits inside a VM in the following steps. First, we check the integrity of the jump-tables ([175, 178, 148, 28]). Then we check the integrity of critical kernel functions and handlers ([175, 178, 148]). Then we use cross-view validation to check the integrity of critical kernel objects ([237, 178, 128]).

Integrity checking of jump-tables. Since rootkits usually modify the jump-tables and redirect the function pointers to their own handlers, this step checks the integrity of function addresses in the jump-tables. We consider two popular jump-tables: the System-call Table and the Interrupt Descriptor Table. For each table, we find its location in the memory, walk through each entry in the table and check whether the pointer in the entry is the "good value" (the "good" value can be obtained from an intact OS of the same version). A mismatch indicates that rootkits have changed this function to its own handler.

Integrity checking of kernel codes. We check the memory regions of the kernel that store critical functions and handlers. We calculate the hashes of the memory regions and compare them with pre-calculated "good values". As the hash function guarantees the unique value for these kernel regions, any mismatch indicates the memory section has been modified, and the kernel is being attacked by a rootkit.

Cross-view validation of critical objects. The rootkits usually compromise the kernel objects and present wrong information to user-space applications, thus hiding their presence. To detect the hidden objects, we can get a class of objects from two perspectives. One is the untrusted view which is obtained from user space in the VM, while the other one is the trusted view obtained from the kernel space that is unlikely to have been subverted by an attacker. Then we check the consistency between the two views. If an object appears in the trusted view but not in the untrusted view, we can conclude that this object has been hidden.

We consider two cases. The first one is *hidden process detection*. We obtain two views of process lists and cross-check the consistency between them. The untrusted view is from the output of user-space applications that display process lists, while the trusted view is from the kernel, e.g., the task list of the scheduler. If one process exists in the trusted view, but not in the untrusted view, we know that this process is hidden from the victim by a rootkit.

The second case is *hidden network socket detection*. Similarly we detect this by comparing the socket lists from the kernel and from user-space applications. Any network sockets in the kernel view list but not the user-space view list will be identified as *hidden sockets*.

6.3.1.2 Mitigation Methods

Once we detect rootkits intrusion, the cloud provider can inform the cloud customers of such security breaches. If pre-authorized by the users' SLA in his Security on Demand [124] requirements, the cloud provider can also eliminate the rootkits for the customers using the following methods.

Repairing the compromised jump-tables. If the rootkits change the function pointers to malicious handlers, the cloud provider can restore the original correct

addresses of these functions from an intact OS of the same version. Then the jump-table integrity breach is fixed and malicious functions will not be invoked any more.

Repairing the compromised kernel codes. If the hash values of the critical kernel codes do not match the "good values", then we can simply remove the compromised codes and restore the known correct ones from an intact OS. This can fix the kernel code integrity breach caused by rootkits.

Fixing compromised critical objects. When we detect hidden objects (e.g., processes, network sockets), we can kill these malicious hidden objects and prevent them from compromising the system. For hidden processes, we can get the process id and then kill it. For network sockets, we can configure firewalls to prevent connections to this stealthy malicious socket. We can also find out the malicious process that established this socket and kill this process. Then this network socket will be disabled.

6.3.2 User-level Malware

Attacker model. We consider the malware launched from the user space. We assume that the launch of malware follows the common routine as normal programs, i.e., creating correct memory address spaces, invoking the correct system calls to start execution, etc. We also assume that at runtime the malware do not compromise the process management and execution routine inside the kernel. So the execution traces (e.g., system calls, APIs) revealed to the OS always truly reflect the malware's runtime behaviors.

One typical solution to defeating malware is to use anti-malware software. Anti-malware software is designed to protect the victim system by detecting and removing malware inside the system.

6.3.2.1 Detection Methods

Malware detection techniques can be classified into two categories: anomaly-based detection and signature-based detection. Anomaly-based detection usually consists of two phases: a training phase and a detection phase. During the training phase the detector attempts to learn the normal behavior. In the detection phase, the detector monitors the inspected program and checks if its behavior deviates from the normal behavior. A key advantage of anomaly-based detection is its ability to detect zero-day attacks. However, it can have a high false positive rate and it is also difficult to determine what is the "normal" behavior during the training phase. Signature-based detection requires the detector to have knowledge of the characterization and features of the various malware, called malware signatures. Then the detector judges if the program under inspection is malicious by looking for signatures inside the program. Signature-based detection has high detection accuracy of known attacks, but it cannot detect zero-day attacks.

Each method can use either static or dynamic analysis to detect malware. The two methods are described below:

Dynamic analysis. Dynamic analysis leverages runtime information of the program under inspection. It usually happens during or after program execution. Such runtime information includes the network payloads [233], traffic patterns [222, 194, 87], system calls traces [135, 116, 166], or program states [193]. The detector gathers these runtime behaviors from the program's execution and compares them with the normal models in anomaly-based detection or malware signatures in signature-based detection to judge the maliciousness of this program.

Static analysis. Static analysis detects malware by examining the program source code or binary executable. This analysis is usually done before the program executes. A key advantage of static analysis is that it can detect malware without having to allow

it to execute on the victim system and cause damage. For anomaly-based detection, static features usually include the structural byte composition or distribution [140], the control flow graph or the APIs used [46, 47]. For detecting malware, we check if the byte/API distribution, or control flow graph follows normal behaviors. For signature-based detection, static features can be classified into two categories: *byte patterns* [214, 70, 69] and *hash sums* [75]. The signature of *byte patterns* contains one or more constant sequences of bytes, possibly separated by gaps of varying size. For detecting malware, we identify the matching sequences in the inspected program. The *hash sum* signature is the hash values (e.g., MD5) over complete files or parts of files. For detecting malware, we check if the inspected program has the same hash value.

6.3.2.2 Mitigation Methods

When malware is detected, we can also perform some response to prevent it from compromising the system. For static detection, since the program has not been executed we can simply prevent this program from being launched, and delete the malicious files from the system. For dynamic detection, since the procedure happens during the program's execution, we can immediately kill this malicious program to prevent further damage.

6.3.3 Network-level Application Attacks

Attacker model. We consider network attacks in which a remote adversary attempts to get unauthorized access to a server, or inject malicious codes into the server illegally. These include backdoor attacks, SSH brute force attacks, XSS attacks, SQL injection attacks, etc.

6.3.3.1 Detection Methods

One possible solution is to exploit an Intrusion Detection System (IDS) to detect potential vulnerabilities via analyzing the network packets and traffic patterns. An IDS can adopt the signature-based approach, where it compares the contents of network packets with known attack signatures. It can also use the anomaly-based approach, where it identifies abnormal network activities and behaviors.

6.3.3.2 Mitigation Methods

Once we find malicious network activities, we can use firewalls [167, 119] to defeat network-level attacks. Firewalls are widely used to protect a system's network against attacks from the outside network. Usually firewalls monitor and control network traffic based on predetermined security rules. These rules define the access control policies based on source/destination IP addresses, source/destination port numbers, protocol types, etc. Firewalls block network connections that disobey the pre-defined security rules. By cutting off the system's connections with untrusted parties, the system is protected against network-level attacks. Besides, we can also identify and kill the suspicious processes inside the system that send or receive illegal network packets.

A firewall should consider inbound traffic protection as well as outbound traffic protection.

Inbound protection. This is used to protect and filter the incoming traffic to the victim from the outside network. This can prevent remote parties from sending malicious traffic to the victim server.

Outbound protection. This is used to control and manage the outgoing traffic from the victim server to the outside network. This can prevent the server from connecting to malicious websites, or sending out sensitive data [66].

6.4 CloudGuard Architecture

We aim to design an architecture that can provide runtime integrity protection of virtual machines as a service in public clouds to customers who are concerned about security and willing to pay extra cost for better security. We want an architecture that fits into the *CloudMonatt* framework. We first describe the basic requirements that such an architecture should have. Then we introduce *CloudGuard*, its architecture overview and threat model.

6.4.1 Architecture Requirements

In order to support VM runtime protection in public clouds, the architecture must satisfy the following requirements:

(1) *Trustworthiness:* the detection and protection results must be correct and reliable.

(2) *Comprehensiveness:* the architecture should be able to support protections of different aspects of security.

(3) *Automatic Detection:* the architecture should be able to automatically detect potential vulnerabilities without customers' further involvement.

(4) *Automatic Mitigation:* the architecture should be able to automatically mitigate the identified vulnerabilities without customers' further involvement.

(5) *Deployability:* the protection mechanism can be easily integrated into current commercial cloud systems.

(6) *Non-interference:* the protection activity should not interfere with the VM's execution. It should not interrupt VMs or cause non-negligible performance overhead.

We design *CloudGuard* to achieve the above requirements. For requirement (1), *CloudGuard* leverages VM introspection to detect and protect VMs from the hypervisor

layer. This method is trustworthy and resistant to attacks inside the guest VMs. For extra security, we can apply Bastion-like hardware isolation architectures [61] for protecting the hypervisor. For requirement (2), *CloudGuard* can protect customers' VMs from a variety of security breaches. It provides the security services of a rootkit scanner, anti-malware and firewalls that customers can choose from. For requirement (3) and (4), once customers select the integrity protection services, *CloudGuard* automatically deploys security detection as well as mitigation for the VMs. For requirement (5), we integrate *CloudGuard* in the OpenStack cloud software to show its deployability. For requirement (6), our evaluation indicates *CloudGuard* introduces negligible performance cost to the platform and VMs.

6.4.2 Overview

As a security service, *CloudGuard* enables customers to select different protections based on their demands and security requirements. Then it automatically deploys the corresponding actions to monitor the VMs and mitigate the potential vulnerabilities. *CloudGuard* keeps informing customers of their VMs' security status. *CloudGuard* can be integrated into the *CloudMonatt* framework as one type of security protection (Details about the integration of this defense in *CloudMonatt* will be illustrated in Section 7.1). Figure 6.1 shows the architecture, and workflow of *CloudGuard*. It involves four entities:

Cloud customer: During VM launch, the customer places a request for leasing VMs with specific resource requirements and security requirements to the Cloud Controller. At VM runtime, the customer can choose different protection options offered by *CloudGuard*, based on their demands. For instance, a VM which runs web applications may need protection of network activities. Another customer frequently downloading and running applications from untrusted sources may want malware detection. Customers can select automatic mitigation of various attacks when these

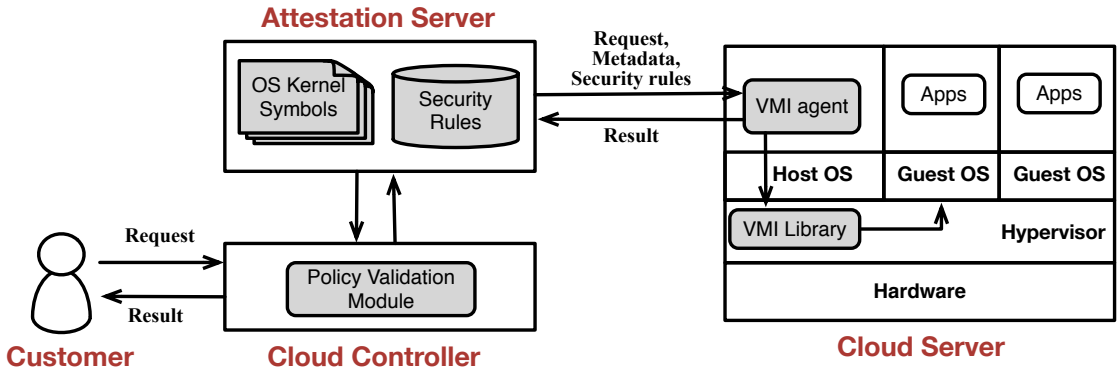


Figure 6.1: Architecture Overview

are detected. This involves giving *CloudGuard* the right to modify the guest OS kernel albeit only to known good values. For each protection, the customer identifies the corresponding security rules and sends the protection requests to the Cloud Controller. He can disable the selected service or add new a service at any time.

Cloud Controller: The Cloud Controller acts as the cloud manager, responsible for taking VM resource and security requests, and servicing them for each customer. When a customer selects a VM protection option, the Cloud Controller forwards the protection request to the Attestation Server. When receiving the results, it sends them back to the customer to keep him informed of the VMs' security health.

Attestation Server: The Attestation Server is a centralized server for managing the introspection services. It communicates with the target cloud server and invokes the introspection. It stores kernel symbol files for different versions of OSes, which map the exported kernel variables and functions to their virtual addresses. Such information are required for VM introspection. If the customer requests a commodity OS, then the Attestation Server can obtain this file from the OS. If the customer uses his own OS, then the Attestation Server will request this mapping file from the customer during VM launch. The Attestation Server also maintains different sets of security rules to judge if a VM is secure or not. Some of these rules are publicly accepted. For instance,

one possible security rule for anti-malware is disallowing the execution of any malware in a public malware database. Another rule for network firewalls can be disallowing the VM's connections to a pool of malicious websites. Customers can also define their own rules for their VMs and pass them to the Attestation Server: they can specify a whitelist of programs that are allowed to be executed in the VM, or IP addresses that are allowed to be connected.

When receiving a VM protection option, the Attestation Server identifies the guest virtual addresses of kernel variables or functions necessary for this protection. Then it sends to the host server the corresponding introspection requests along with these metadata and security rules. When the Attestation Server receives the results from the host server, it sends them back to the Cloud Controller.

Cloud server : The cloud server is the computer that runs the VMs in question. It is also responsible for deploying the desired security services for the customers' VMs. Figure 6.1 shows the architecture of a cloud server which adopts the Xen hypervisor.

CloudGuard leverages the technique of Virtual Machine Introspection (VMI) [95] to monitor and detect vulnerabilities inside a VM. It implements the security tools in the hypervisor layer. Then these tools will not be compromised by malware inside the VMs. Besides, this does not require modification of customers' VMs. Specifically, the cloud server has a `VMI agent` in the host OS, which is responsible for taking and parsing the requests from the Cloud Controller to conduct the corresponding VMI monitoring service and interpreting the introspection results. Once violations against the security rules are detected, the `VMI agent` will mitigate the vulnerabilities for the customers if the customer has selected automatic mitigation in his security SLA. Then it sends back the detection and mitigation results to the Cloud Controller. The cloud server also has a `VMI Library` located in the hypervisor, which provides VMI functions and APIs to the `VMI agent`. The `VMI Library` achieves these functions

using hardware and software virtualization techniques, e.g., pausing/resuming VMs, reading/writing VMs' data and changing memory permissions (Section 6.5.2).

6.4.3 Threat Model

We consider the threat model where hostile applications or services may be running inside the customers' VMs, gaining the guest OS privileges, and the capability of compromising the whole VM. However, we assume that such hostile applications cannot tamper with the host OS and hypervisor layer. Specifically, in each server, the `VMI agent` is used to trigger and manage the introspection functions. The `VMI Library` conducts the actual introspection functions. These two modules should be trusted, as well as their communications. If the `VMI agent` or its communication channel with the `VMI Library` is untrusted, a wrong introspection request can be sent to the `VMI Library`. This violates the integrity of the introspection results. If the `VMI Library` is untrusted, then it can also conduct an incorrect VM introspection. Worse yet, the `VMI Library` has the capability to probe the guest VMs' memory, or change the guest VMs' runtime behaviors, thus the confidentiality, integrity and availability of the guest VMs are vulnerable to a malicious `VMI Library`. So the `VMI agent` and `VMI Library` must be protected and mutually authenticated.

6.5 Implementation

6.5.1 CloudGuard Prototype

We implemented a prototype of *CloudGuard* by integrating it into the OpenStack Newton platform [19]. OpenStack is composed of different modules, with each module supporting a different service. We modified two modules. The first one is *horizon*, which is implemented as OpenStack's dashboard and provides a web-based user interface to

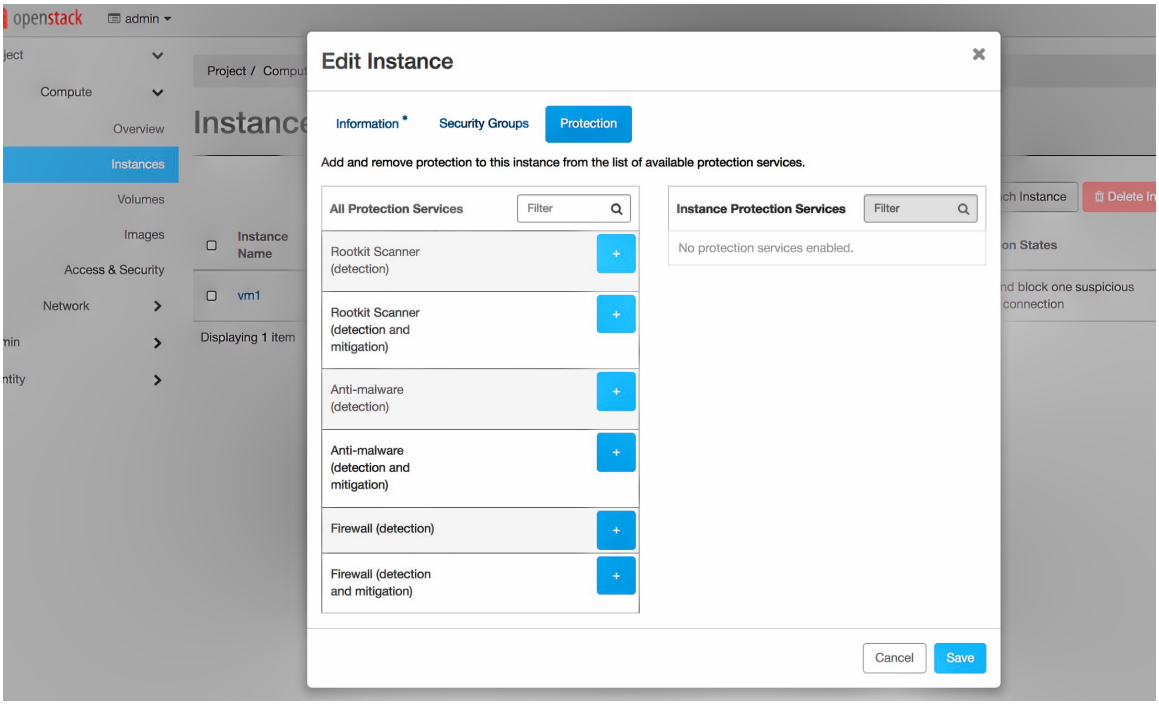


Figure 6.2: A screenshot showing the added protection service in the *CloudGuard* OpenStack Dashboard.

other services of OpenStack. We modified this module to enable customers to select VM protection for their VMs. Figure 6.2 shows the modified dashboard. *CloudGuard* provides different sets of protections (detailed mechanisms of these protections will be described in Section 6.5.3). Customers can start or stop any protection at any time during VM runtime. *CloudGuard* will deploy the corresponding protection on the VM and display the results to customers in the dashboard. The second module is *nova*, which is used to manage computing services in cloud servers. We modified the *novaclient* (nova API client) to pass protection requests from *horizon* to *nova*, as well as results from *nova* to *horizon*. We also added a new file in *nova* to invoke protection service on the host cloud server upon receiving the protection requests.

We integrated the OpenAttestation software tool [17] in the OpenStack system for cloud server authentication and the inter-server communication protocol. The Cloud Controller and cloud servers are exchanging information via the Representational state transfer (RESTful) HTTPs web service [92], which allows the two parties to access and

manipulate textual representations of web resources using a uniform and predefined set of stateless operations, such as GET, POST, PUT, DELETE, etc. On each cloud server, we adopted the open-source LibVMI library [12] as the `VMI Library`. We wrote a software module as the `VMI agent` for managing the introspection activities.

In total, we added 232 lines of Python code in OpenStack, 204 lines of Java code in OpenAttestation, and 1569 lines of C code as `VMI agent` in each server.

6.5.2 VMI Functionalities

Many opensource tools and commercial applications were developed to conduct virtual machine introspection in virtualization platforms (e.g., LibVMI [12], Libbdvmi [11], HVI [49], VMIdbg [258]). These tools exploit hardware and software virtualization support to monitor and audit VMs' memory and activities from outside the guest OS. We describe the functionalities that we use to monitor and mitigate vulnerabilities of VMs.

6.5.2.1 Monitoring VMs

One basic functionality is to access the memory of guest VMs. The challenge is the semantic gap between the high-level data observed by the guest OS and the low-level data observed by the hypervisor. A process inside the guest OS accesses data via its virtual address, which will be translated to the guest physical address by the guest OS, and then the host physical address by the hypervisor. If the hypervisor attempts to access data of a process in a guest VM at a specified guest virtual address, it has to conduct the two levels of address translation without the context of the guest OS.

Address Translation. This can be achieved by the following steps:

1. The hypervisor obtains the base (guest physical) address of the process's page directory. If this process is a kernel process, then the kernel page directory is stored in a fixed known guest physical address. If it is a user-space process, the hypervisor

first gets the process structure list stored in a fixed known guest physical address, and then iterates this list until it finds the given process. From this structure, the hypervisor can get the address of this process's page directory.

2. The hypervisor translates the guest physical address of the page directory into a host physical address, and then takes the desired guest virtual address and translates it to the guest physical address using the page table.
3. The hypervisor then translates the guest physical address into host physical address, and accesses the data from the host physical page.

Accessing guest VM registers. In addition to accessing the guest VM's memory, the hypervisor can also access a VM's (virtual) registers. The hypervisor maintains a set of data structures (e.g., VMCS in Intel processors, VMCB in AMD processors) to save and restore register values for each VM's virtual CPU during a VM context switch. So the hypervisor can easily read or write any register values from its internal structure.

Dynamic event capturing. A more powerful feature of VMI is to capture the occurrence of some critical functions (e.g, syscalls, APIs). When one such function occurs inside a VM, a VM exit will be invoked and the CPU is trapped into the hypervisor. The hypervisor can achieve this functionality using the following steps:

1. *Register the event*: given the address of the monitored function inside the guest virtual memory, the hypervisor can insert a breakpoint at this address using one of the following two methods: (1) ***Extended Page Table (EPT) violation***: the hypervisor can set the memory page containing this function as Non-Executable (NX) in the EPT entry. When the VM executes this function, an EPT violation occurs and the processor is trapped into the hypervisor. (2) ***INT 3 interrupt***: the hypervisor can insert a debugging instruction (`0xCC`) at the address of this function. When the VM calls this function, a software interrupt happens and traps the processor to the hypervisor.

2. *Handle the event*: the hypervisor is notified of this event. It can conduct necessary introspections into this VM's memory. Then it clears the breakpoint in the monitored function to allow the VM to proceed execution without being interrupted again: for EPT violation, the hypervisor clears the NX flag in the EPT entry; for `INT 3` interrupt, the hypervisor clears the debugging instruction. Then the hypervisor sets the Monitor Trap Flag (MTF) and the processor enters the single-step operation mode: the VM can only execute one instruction and then yield to the hypervisor.
3. *Re-register the event*: after the VM executes one instruction (i.e., calling the monitored function), the processor is trapped into the hypervisor. The hypervisor re-registers the event of monitoring the function by resetting the NX flag or re-inserting the `INT 3` instruction to the monitored function. It also resets the Monitor Trap Flag to disable the single-step CPU mode. After all these are done, the VM will continue execution till it calls the monitored function again, and the above procedure is repeated.

6.5.2.2 Modifying VMs

Another technique is to actively change the VMs' memory data or execution paths. With this technique, *CloudGuard* can automatically and promptly take actions to mitigate the integrity breach and prevent further damages once the breach is identified.

Repairing compromised data. Malware can compromise the integrity of the OS by modifying security-critical data. When the guest OS kernel is compromised, the hypervisor can restore the original correct data. For instance, malware can change a kernel function pointer to their own malicious handler. To defeat this, the hypervisor can change the function pointer back to its original one by referring to an intact OS of the same version, then the malicious function will not be invoked.

Bypassing a function. When the guest VM starts to invoke malicious programs or functions, we can modify the VM's execution path to bypass the malicious code. Specifically, in the guest VM when a function $\mathbb{G}$ attempts to call a malicious function $\mathbb{F}$, we aim to bypass the function $\mathbb{F}$ so that the VM will directly return to function $\mathbb{G}$ without executing $\mathbb{F}$.

Figure 6.3 shows the procedure of bypassing a function. By convention, the X86 system usually uses the register `RIP` to store the next instruction pointer, `RBP` to store the base pointer (i.e., the start of the stack), `RSP` to store the stack pointer (i.e., the current location in the stack), and `RAX` to store a function's return value. In the normal case, before calling function $\mathbb{F}$, `RIP` stores the next instruction pointer after $\mathbb{F}$ (denoted as *old_rip* in Figure 6.3 (①)), which is also the return address of $\mathbb{F}$. Then function $\mathbb{G}$ pushes the return address (*old_rip*) into its stack frame pointed to by `RSP` and then jumps to function $\mathbb{F}$ (②). Function $\mathbb{F}$ will establish its own stack frame by pushing function $\mathbb{G}$'s base pointer (stored in `RBP`, denoted as *old_rbp*) to the stack, and directing the base pointer (`RBP`) to the stack pointer (`RSP`) (③). After that $\mathbb{F}$ starts to execute its code within its own stack. When finished, this function will pop off the return address (*old_rip*) and base pointer *old_rbp*, and assign them to `RIP` and `RBP` (④). Then it will jump back to $\mathbb{G}$.

To bypass function $\mathbb{F}$, we can first use the event capturing mechanism in Section 6.5.2.1 to capture the moment that function $\mathbb{G}$ jumps to function $\mathbb{F}$ (i.e., the completion of ②). Then we can bypass step ③ in which function $\mathbb{F}$ establishes its stack, and direct the processor to step ④. Specifically, we pop off the return address *old_rip* from function $\mathbb{G}$'s stack. This value is pointed to by `RSP` in ②. We assign this value to `RIP` (④). We can also assign any return value we want for function $\mathbb{F}$ to `RAX`. By doing so, after entering function $\mathbb{F}$, this function will jump back to the return address directly with our assigned return value. So function $\mathbb{F}$ will be bypassed and never executed.

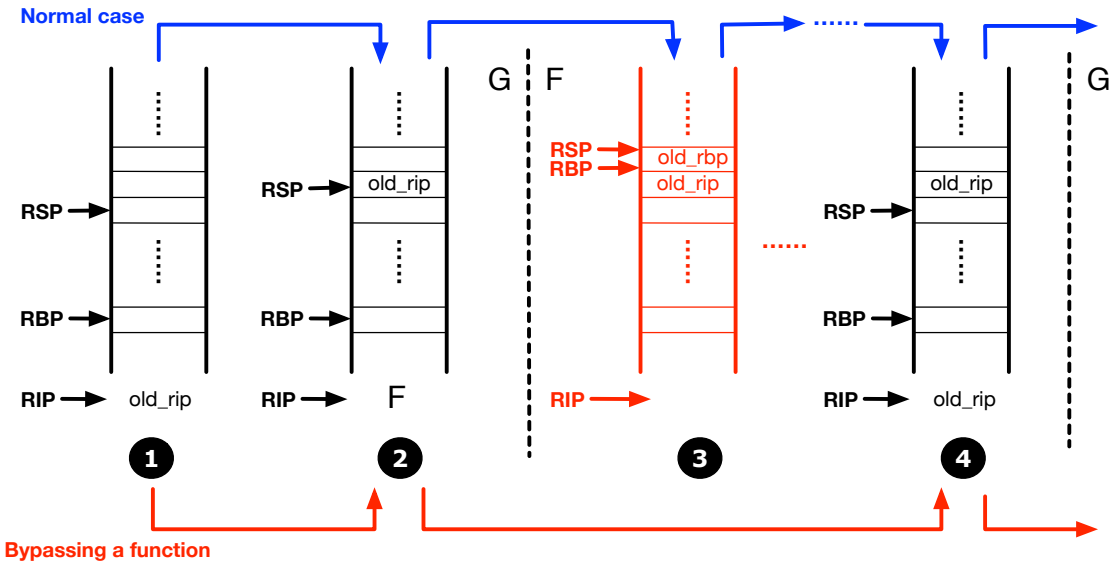


Figure 6.3: Bypassing a function. The red blocks are bypassed

Killing a process. When a malicious process has been launched and running in the guest VM, we aim to kill this active process immediately without interrupting the VM's execution. The idea is to insert the process killing function (e.g., `sys_kill` in Linux) in the VM's code path, and set the function parameter as the malicious process's id. Then the VM will jump to the process killing routine and return to the original code after killing the process.

Figure 6.4 shows the procedure of killing an active process. By convention, the X86 system usually uses the register `RDI` to store the first parameter of a function, and `RSI` to store the second parameter. Specifically, we need to insert the kernel function `sys_kill` in the kernel-space code path. To achieve this, we can choose one of the most frequent kernel-space events (e.g., CPU scheduling function `schedule`), and monitor this event using the mechanism from Section 6.5.2.1. When this event occurs (①), the OS is in the kernel mode. Then we interrupt this VM and save the CPU registers. We push the value of the instruction pointer register `RIP` (denoted as *old_rip*) to the memory stack pointed to by `RSP`, and change `RIP` to the address of `sys_kill`. We also set the first parameter of `sys_kill` (stored in `RDI`) as the process

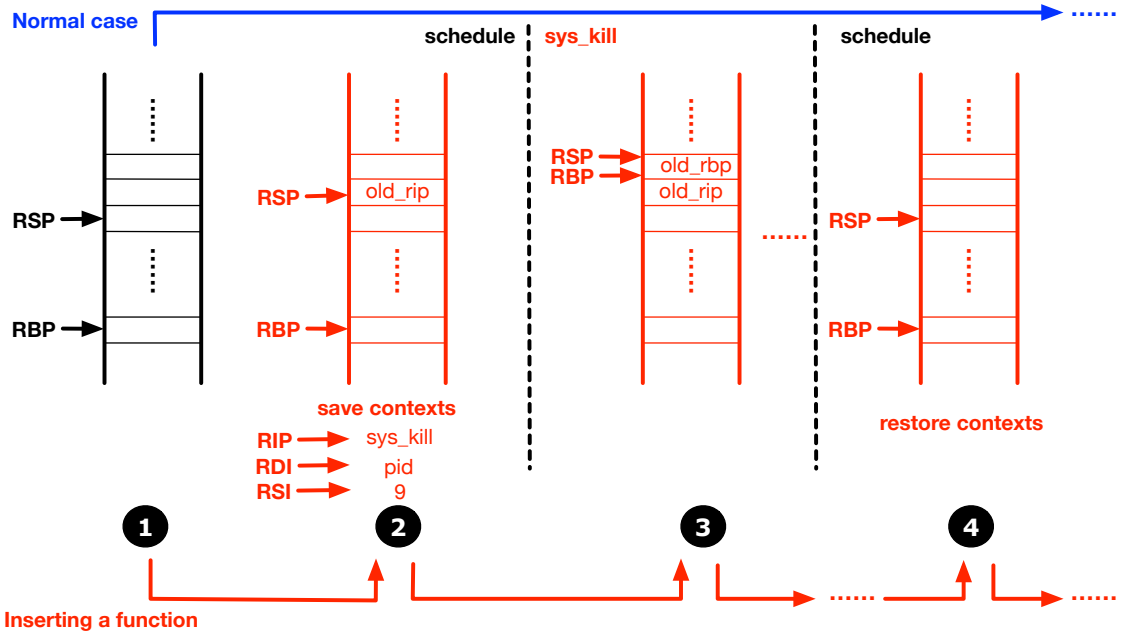


Figure 6.4: Killing a process. The red blocks are inserted routine

id, and the second parameter (stored in `RSI`) as the killing signal 9 (②). After that the VM will jump to the `sys_kill` kernel function (③) and kill the process. We also monitor the event that the `sys_kill` completes and returns. When this function returns, we interrupt the VM, restore the registers and instructions (④). The VM will continue its previous path.

6.5.3 Security Tools

We implemented different security tools in *CloudGuard* using the VM introspection techniques from Section 6.5.2. These tools include a rookits scanner, an anti-malware program and a firewall. We apply them to detect as well as mitigate integrity breaches inside a guest VM. Other security tools can be integrated into *CloudGuard* similarly.

Testbed Configuration: Our case studies are based on a Dell R210II server. The server is equipped with one quad-core, 3.30GHz Intel Xeon E3-1230v2 processor with 8MB LLC, and it supports Intel VT hardware-assisted virtualization technology, which

is necessary for VM introspection. We configure a Xen hypervisor (version 4.7.0) to host fully virtualized guest VMs. Dom0 runs Ubuntu 14.0.4 OS with kernel version 4.2.0-42. The guest VM that we are testing runs an Ubuntu 10.10 server 64-bit OS with kernel version 2.6.35-22.

6.5.3.1 Rootkit Scanner

A rootkit scanner can help check the integrity of a guest OS kernel and detect kernel rootkits. We show how to detect and repair the malicious modifications of kernels caused by rootkits, using the methods from Section 6.3.1

Integrity checking of jump-tables. We propose a security rule for checking jump-table integrity: *the address of each handler function indexed by the jump-tables should match the "good" known one.* So we need to get the address of each handler function and check if it has been changed to an illegal one. In Linux, the base address of the System-call Table is denoted by the symbol `sys_call_table`. The base address of the IDT is stored in the register `IDTR_BASE`, or denoted by the kernel symbol `idt_table`. With these virtual addresses, we can use the introspection method from Section 6.5.2.1 to iterate these jump-tables, get the address of each function handler and compare it with the one from an intact OS kernel of the same version. If one address does not match the corresponding "good" one, we can suspect that the rootkit has changed this handler to its own malicious function.

When one function address is changed, we can get this function index based on its location in the jump-table. We report the name of the hijacked function to customers. By restoring the original function address in the jump-table we can fix this integrity breach.

Integrity checking of kernel codes. We propose a security rule for detecting kernel code integrity: *the hash value of the critical memory region should match the "good" known one.* In Linux, the lower bound address of the kernel memory region that

stores the critical kernel functions is denoted by symbol `_stext` and the upper bound address is denoted by `_etext`. We calculate the MD5 hash value of the data inside this region [`_stext`, `_etext`] and compare it with the one from an intact OS kernel of the same version. Mismatched hash values indicate that certain kernel functions inside this memory region have been compromised.

To mitigate this integrity breach, we can restore the original correct codes within [`_stext`, `_etext`] from an intact kernel. Then the integrity of the kernel codes is maintained.

Cross-view validation of critical objects. We use cross-view validation to detect objects hidden by rootkits. We get an untrusted view from the user-space programs, and trusted view from the kernel space. We implemented the security rule: *the two views should give consistent results.* We consider the detection of hidden processes, and network sockets.

To detect if the VM has hidden processes, the hypervisor needs to get two views of process lists inside this VM. The trusted view shows all the processes while the untrusted view might be tampered with by the rootkits. Trusted view can be obtained from the linked task list maintained by the OS kernel. We first obtain the virtual address of the list head from the kernel symbol `init_task`. Then we iterate this list, and read each process's information from the `task_struct` kernel structure, e.g., `comm` (process name); `tasks` (pointer to the next process); `mm` (memory descriptor); `pid` (process id), etc. By doing so, we can get all the processes running in the OS. To get the untrusted view of the process list, the hypervisor can issue a remote `ps aux` command to the VM via SSH, which is a common way to execute commands on a remote machine. Then the users' view of the process list will be transmitted to the hypervisor. By comparing the two lists, we can identify any hidden process's name and id.

Once detected, the hypervisor can directly kill this hidden process using the method from Section 6.5.2.2, i.e., invoking the `sys_kill` function in the code path with the hidden process's id as the parameter. Then this process will be killed without interrupting the VM's execution.

To detect if the VM has hidden network sockets, the hypervisor also needs to get the trusted view and an untrusted view of active network sockets in this VM. For the trusted view, the Linux kernel uses hashmaps to store the network sockets. The TCP hashmap is denoted by the kernel symbol `tcp_hashinfo` and the UDP hashmap is denoted by the symbol `udp_table`. We can get the virtual addresses of these hashmaps from these symbols, and iterate the table to retrieve the trusted list of active sockets. For the untrusted view, the hypervisor can issue a remote `netstat` command to the VM via SSH, and retrieve the list of sockets from the user's perspective. Through comparing the two lists, we can find the hidden TCP or UDP sockets.

To prevent the connections from these hidden sockets, we can use a firewall in Section 6.5.3.3 to block such connections and kill the malicious processes that establish these sockets.

6.5.3.2 Anti-malware

We can leverage VM introspection to implement an anti-malware tool in the hypervisor layer to monitor the programs' images and execution inside the guest OS. This enables static malware detection as well as dynamic detection.

Static analysis. The static detection method checks the program image before it is launched. Specifically, when a VM attempts to launch a program, the hypervisor checks if the image of this program is problematic by checking the control flow graph, byte patterns or hash sums in the image. If the image is malicious, the hypervisor prevents this program from being launched. There are three steps to conduct static malware detection.

First, we need to capture the program launch event. In Linux, the OS invokes the system call `sys_execve` to launch a program, initializing this program and loading the image to the memory. We use the feature from Section 6.5.2.1 to monitor a key function `do_execve` inside `sys_execve`. When this function is called, the VM is interrupted and control is transferred to the hypervisor.

Second, the hypervisor finds and analyzes the image of the program from the disk. To achieve this, we first need to know the path of the image in the guest VM filesystem. The first parameter of `do_execve` is a struct `filename`, which stores the path of the program executable. This path can be a full path or a relative path to the working directory, depending on how the VM runs the program. If it is a relative path, we get the working directory path from the `task_struct` of the invoking process, and convert the relative path to the full path. Given the full path, we mount the VM filesystem to Dom0 and retrieve the image. Then we can analyze the image and check if it is problematic. In our implementation, we adopt the static signature-based detection, which calculates the MD5 hash value as the signature. Other static detection methods can be implemented in a similar way.

Third, once the program image is problematic, the hypervisor should prevent this program from being launched. The idea is to bypass the `do_execve` function using the method in Section 6.5.2.2. So this function will not be executed to launch this suspicious program.

Dynamic analysis, The detector monitors the dynamic behavior of the inspected program during execution. We can use VM introspection to trace the syscalls invoked by each program inside a VM and check if the syscall trace follows a normal model in anomaly-based detection, or shows malicious features in signature-based detection.

To trace the syscall of the inspected VM, we can insert a breakpoint in the syscall entry routine, the address of which is stored in the register `MSR_LSTAR`. When a process invokes a syscall, the hypervisor will be notified and get the CPU control. It can get

the process id from the register CR3, the syscall index from the register RAX, and the syscall parameters from other general-purpose registers RDI, RSI, etc. Based on the syscall trace of each process, the hypervisor is able to conduct dynamic analysis and judge if these processes are malicious.

When the hypervisor discovers a malicious process, it can simply kill this process using the method from Section 6.5.2.2.

6.5.3.3 Firewall

We can use VM introspection to establish a firewall for the protected VM. The security rule is to prevent the VM from talking to machines with malicious IP addresses and/or port numbers specified in a blacklist, or to limit the VM's connections to some specific IP addresses and/or port numbers in a whitelist. We consider two cases for the protected VM: establishing outbound connections to remote servers, and accepting inbound connections from remote clients.

Outbound protection. To establish an outbound socket connection, the program calls the sys_connect syscall. The key function inside this syscall is inet_stream_connect: its second parameter is a pointer to a sockaddr structure which contains the remote server's IP address and port number. So to detect the connection event, we first set a breakpoint at the function inet_stream_connect. When this function is called, the hypervisor is notified, gets the sockaddr pointer from register RSI, and then gets the remote server's IP address and port number from this structure. With such information the hypervisor can check if the monitored VM is attempting to connect to malicious servers.

mInbound protection. To accept an inbound socket connection, the user-space program calls the sys_accept syscall. This syscall then calls the kernel function inet_csk_accept, which fetches an incoming request from the network queue, and returns its IP address and port number in the structure sock. So we set a breakpoint

at the return instruction of this function. When this breakpoint is reached, the return value (`RAX` register) stores the pointer to the remote client's IP address and port number. We can read such information and check if the VM is attempting to accept connections from a malicious client.

Once the hypervisor discovers that the VM is establishing or accepting illegal connections, it can also get the id of the process that invokes these network socket APIs. To block the connections, the hypervisor can simply kill the process using the method from Section 6.5.2.2.

6.6 Evaluation

We evaluate the performance of VM introspection and its impact on the inspected VMs. We aim to check how long it takes for the hypervisor to check the VM's integrity, and show its performance overhead with respect to the VMs. The main performance cost is caused by the VM interruption for event capturing.

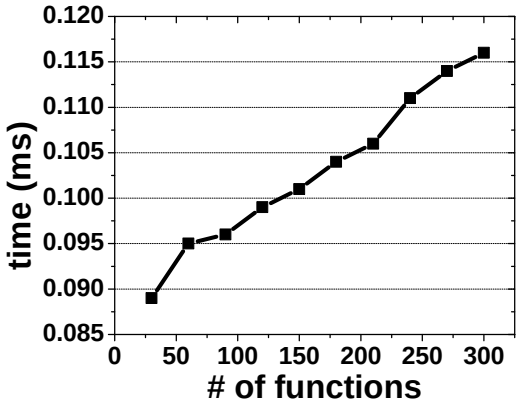
6.6.1 Rootkits Scanner

We measure the time to scan the kernel for rootkits.

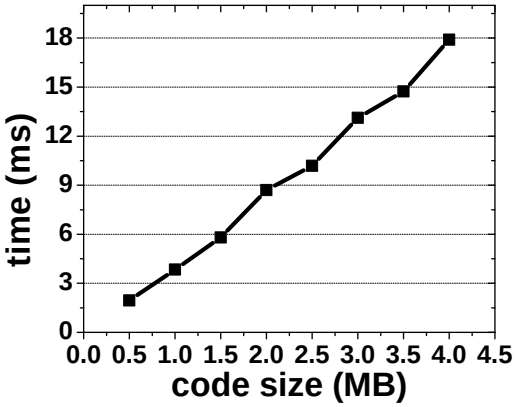
Integrity checking of jump-tables. Figure 6.5a shows the introspection time versus the number of functions in the jump table. We observe that the introspection time is linearly related to the number of functions. When the jump table has 300 functions, the introspection can be done within 0.12ms, which is very fast.

Integrity checking of kernel codes. Figure 6.5b shows the time to calculate hash values versus the critical memory size. When the critical memory is on the order of MB, the whole checking process can be done on the order of milliseconds.

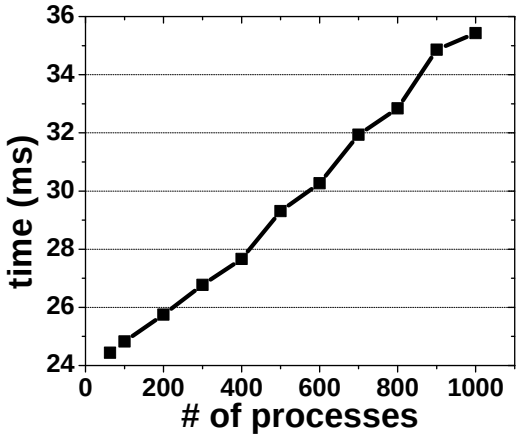
Cross-view validation of critical objects. Figures 6.5c and 6.5d shows the time used to get the trusted view of processes and network sockets in the system. The



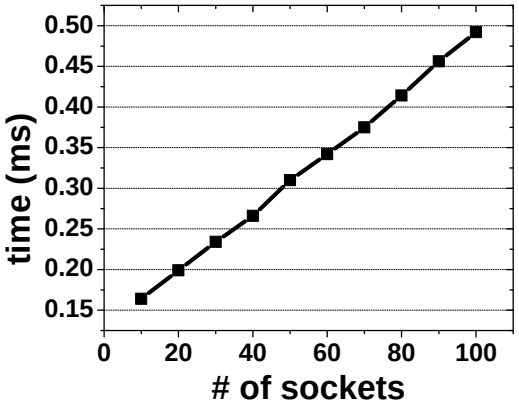
(a) Jump-table checking



(b) Kernel code checking



(c) Process scanning



(d) Socket scanning

Figure 6.5: The performance of rootkits scanner.

time is also linear to the number of processes or sockets in the trusted list. The total checking time is also very short even though the system has a large number of processes or sockets.

6.6.2 Anti-malware

Static analysis. We consider the performance impact when we use VM introspection to conduct static signature-based detection. Every time before the VM launches a program, the hypervisor interrupts the VM, mounts the VM filesystem and calculates its image hash.

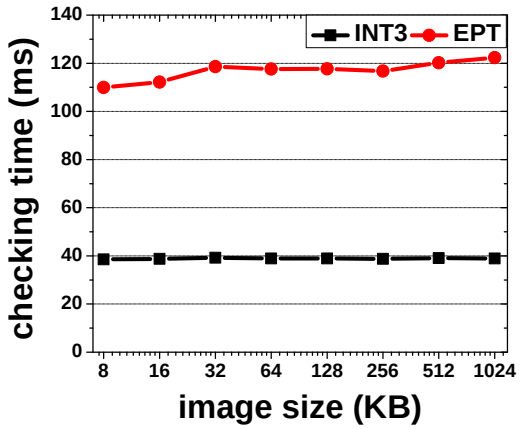
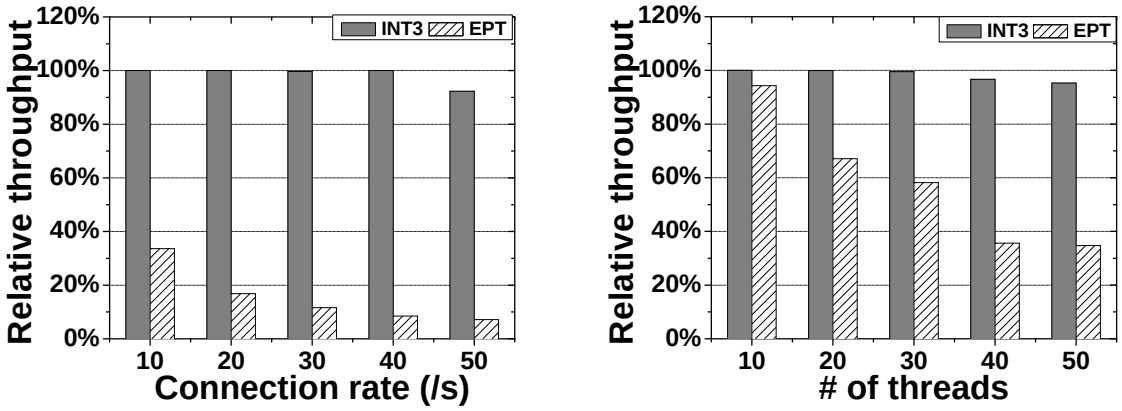


Figure 6.6: Static malware detection

We measure the extra launch time of programs with different image sizes, using either `INT3` breakpoint interrupt or EPT violation interrupt for event capturing. The results are shown in Figure 6.6. We can see the image size does not affect the launch time. The overhead mainly comes from the event interrupt. We can also see the EPT interrupt is worse than the `INT3` interrupt. This is because `INT3` sets an interrupt event in one instruction, while EPT sets an interrupt for the whole physical page. So the EPT mechanism causes more false positive interrupts (other instructions or data in the same memory page cause the interrupts) than `INT3`. In general the performance overhead caused by launch checking is acceptable.

Dynamic analysis. We measure the performance overhead of tracking syscalls inside a guest VM for dynamic detection. We configure the monitored VM to run two cloud applications (web server and storage server) from [243], and set another VM on another host machine as a client to stress this monitored VM. Figure 6.7 shows the relative performance of the monitored VM under the syscall tracking using `INT3` and EPT interrupt. We observe that the EPT violation can reduce much more throughput of the monitored VM than the `INT3` interrupt. When the monitored VM is fully saturated, there are more critical events happening and being interrupted. So



(a) Web server (b) File server

Figure 6.7: The performance overhead of different cloud benchmarks under dynamic malware detection.

the performance degrades. However, if we use the `INT3` mechanism, even if the VM is saturated, the worst performance overhead is around 8% for the web server.

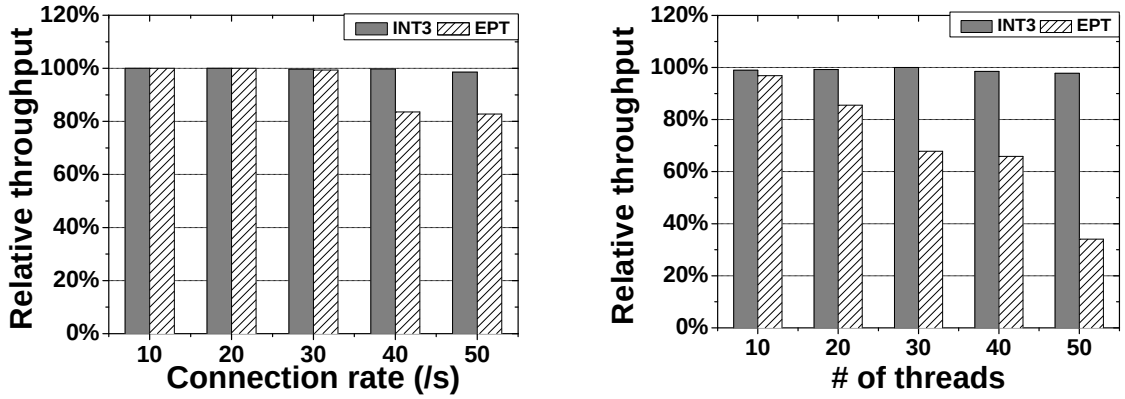
6.6.3 Firewall

Similarly we also monitor the performance overhead of the VM under socket API tracking for the firewall protection, as shown in Figure 6.8. We can see that the `INT3` mechanism has smaller performance overhead than the EPT mechanism. When the inspected VM is fully saturated, the performance overhead is less than 5% for `INT3`, which is acceptable.

6.7 Discussions

There are multiple ways to implement the security protections introduced in Section 6.3. We discuss and compare their advantages and disadvantages.

OS-enabled protection. The most straightforward way to deploy security protection is to install security tools in the OS of the guest VM. Current cloud providers usually adopt this solution to protect customers' VMs (e.g., Amazon Inspector [3], Azure



(a) Web server (b) File server

Figure 6.8: The performance overhead of different cloud benchmarks under firewall protection.

Antimalware [15]). The biggest drawback for this method is that the security tools are also vulnerable to the malware. As we introduced in Section 6.2, once intruding into the system, malware can easily gain root privilege and control any software entities inside the VM. So it is easy for them to identify and disable the security tools, or hide themselves from these tools. This makes the security tools less effective in protecting VMs' integrity.

Hypervisor-enabled protection. The second method is to isolate the security tools from the vulnerable guest VM, e.g., placing the security tools in the hypervisor layer (VM introspection [95]). We adopt this solution in *CloudGuard*. Assuming privilege ring protections by the hardware works, the malware in the user level (ring 3) or the kernel level (ring 0) should not be able to easily compromise the security tools at the hypervisor level (ring -1). The security tools can provide more trustworthy and reliable protections of the guest VMs.

However, this hypervisor-based protection also has some disadvantages. First, this protection usually requires the modifications of the guest OS to insert breakpoints or mitigation functions. Table 6.1 summarizes the modifications of the guest OS needed for the security tools we implemented in Section 6.5.3. For detection, the EPT

Security tools		Detection		Mitigation	
		INT3 interrupt	EPT violation	INT3 interrupt	EPT violation
rootkit scanner	integrity checking of jump-tables	–	–	compromised jump-tables	compromised jump-tables
	integrity checking of kernel codes	–	–	compromised kernel codes	compromised kernel codes
	cross-view validation of critical objects	–	–	`schedule()`	`schedule()`
anti-malware	static analysis	`do_execve()`	–	`do_execve()`	`do_execve()`
	dynamic analysis	`system_call()`	–	`schedule()`	`schedule()`
firewall	outbound protection	`inet_stream_connect()`	–	`schedule()`	`schedule()`
	inbound protection	`inet_csk_accept()`	–	`schedule()`	`schedule()`

Table 6.1: Modifications of the guest OS.

violation scheme does not require VM modification since it only changes the EPT access attributes. The `INT3` interrupt scheme needs to change some kernel functions (e.g., `do_execve()`, `system_call()`, `inet_stream_connect()`, `inet_csk_accept()`) to insert breakpoints. For mitigation, We need to fix the jump-tables or kernel codes if they are compromised. To bypass a malicious function (e.g., `do_execve()`), we need to modify the function routine. To insert the function `sys_kill()` to kill malicious processes, we need to modify the kernel function `schedule()`. Although we modify the VM in a good way to protect the VM, customers who are concerned with the VM integrity may not like such modifications. Second, the hypervisor-based solution introduces performance overhead to the guest VM, as shown in Section 6.6. The performance overhead is huge when using the EPT violation scheme.

Hardware-enabled protection. A third method is to use hardware-based secure enclaves to protect security tools from untrusted software stacks. Past work designed new systems that can shield security-critical applications from the untrusted OSes. For instance, Bastion [61] provides protected software modules with secure fine-grained memory compartments and secure persistent storage areas. Haven [43] leverages Intel SGX [155] to create isolated execution environments for applications. SICE [35] exploits the System Management Mode (SMM) [9] to provide hardware-level isolation and protection for sensitive applications on X86 commodity platforms. We can install security tools in a hardware-based enclave and place them in the untrusted system (i.e., guest VM). So these tools can monitor the untrusted system's integrity. Even if the system is compromised, the attacker cannot subvert the protection enhanced

by the hardware to compromise the security tools. One drawback of this solution is that the customers have the added burden of installing security tools inside the secure enclaves. Another drawback is the need of new hardware support, although SGX is already available. With the growing popularity of such secure processors, this method will be more promising.

6.8 Chapter Summary

This chapter presents *CloudGuard*, a cloud architecture which offers runtime VM protection services to customers. Key contributions in this work include: (1) *CloudGuard* allows customers to select different security protections based on their needs. (2) We show that VM introspection can not only passively retrieve information from the VMs, but can also actively change the VMs' state and data. This can be exploited to mitigate vulnerabilities inside the VMs. (3) We implement different security tools in *CloudGuard*, e.g., rootkit scanner, anti-malware tool, firewall, etc. We show how *CloudGuard* can detect and mitigate various security threats to enhance a VM's security at runtime. Our evaluation indicates these VM introspection services incur low performance overhead to the monitored VMs. We hope *CloudGuard* can attract and satisfy more cloud customers who have security concerns with leasing VMs in the cloud. We also hope this chapter can inspire computer architects to design more efficient hardware mechanisms that can do what is now done by the hypervisor to enhance both the security and the performance of these cloud security services.

Chapter 7

Conclusions

The past decade has witnessed rapid development in cloud computing. Various technologies and systems are proposed to support new functions and services, and attract more corporations and individuals to shift their computation toward cloud computing. At the same time, the cloud computing environment becomes more complicated, thus more vulnerable to known and unknown cyber threats. As Chapter 2 shows, different attack vectors can be exploited by malicious parties to compromise the security of customers' data and computations in different ways. As customers build their applications and services upon the platforms and infrastructure from cloud providers, it is of paramount importance for cloud providers to create a reliable and secure computation environment for customers. This dissertation designs new architecture and methods to protect customers' virtual machines in the Infrastructure-as-a-Service (IaaS) cloud model. Our architecture implements *Security-on-Demand* and *CloudMonatt*, which enable customers to specify their security needs, and the cloud provider to provide security protections during a VM's lifetime.

In Chapter 3 we presented a new architecture, *CloudMonatt*, that provides the functionality of VM security health monitoring and attestation to customers. We realized the *property-based attestation* scheme in the cloud context, by bridging the

semantic gaps between customers' specification and VMs' micro-architecture measurements. We demonstrated the implementation of *CloudMonatt* in the opensource software OpenStack, to show its deployment feasibility in public clouds. We conduct performance evaluation and security verification of *CloudMonatt* to prove it is an effective and trustworthy security service for customers.

In Chapter 4 we focus on the *availability* property of virtual machines. This is motivated by the fact that co-located VMs still share and contend for memory resources even with the strong memory isolation enforced by the hypervisors. This study proposes memory DoS attacks based on such settings, as well as the corresponding defense solution. For attacks, we design a set of techniques to attack different layers of hardware memory resources to degrade the victim VM's performance. The power of these techniques is validated in a public cloud. For defense, we leverage existing hardware features, e.g., Hardware Performance Counters and duty cycle regulation, to detect and mitigate the damage caused by the attacks. This defense can be integrated into the *CloudMonatt* framework, as a security option for customers to choose, who have high demand for resource *availability*.

Chapter 5 studies a *confidentiality* property of virtual machines. Prior work have shown that cache side-channel attacks are a serious confidentiality threat in multi-tenant cloud servers. So we propose novel methods for cloud providers to detect and then mitigate all cache side-channel attacks. In our method, the cloud provider just uses Hardware Performance Counters to monitor the micro-architectural behaviors of the protected VM and the potential adversary VM. Through statistical analysis, the cloud provider is able to identify the threat immediately when the attack begins, and stop the information leakage promptly. Our approach does not require modification to hardware, or privileged software. It can be easily integrated into the *CloudMonatt* framework, as another option for customers to select.

Chapter 6 studies runtime system *integrity* protection. In this chapter, we focus on the threats from inside-VM malware, instead of from co-located VMs as in Chapters 4 and 5. Just like in traditional computing models, customers' VMs can be also compromised by malicious parties. Although it is the customers' responsibility to protect their own VMs from being attacked, we think the cloud provider has the advantage and capability to deploy more reliable and trustworthy protection for VMs, thus making cloud services more attractive. We leverage the Virtual Machine Introspection technique to monitor and mitigate customers' VMs in a comprehensive way. This integrity service can also be integrated into *CloudMonatt*, and exposed to customers.

7.1 System Integration

Figure 7.1 shows that different security mechanisms introduced in this dissertation can be integrated into our *CloudMonatt* architecture. Compared to Figures 3.1 and 3.2, this figure expands the `Monitor Module` in the cloud server, the `Attestation Database` in the Attestation Server and the `Controller Database` in the Cloud Controller, while the rest of the Attestation Server and the Cloud Controller, and the `Trust Module` in the cloud server are the same. For the `Monitor Module`, we integrate the monitoring mechanisms we discussed for providing some aspects of availability (Figure 4.16), confidentiality (Figure 5.4) and integrity (Figure 6.1). For availability, the cloud server uses the `VMM Profiler` to monitor VMs' CPU usage (Section 3.3.3) and `Hardware Performance Counters` via the `Performance Monitor Unit (PMU) Kernel` to monitor VMs' memory availability (Chapter 4). It also uses the `I32_CLOCK_MODULATION` to regulate VMs' execution speed. For confidentiality, the cloud server uses `Hardware Performance Counters` via the `PMU Kernel` to detect covert-channel (Section 3.3.2) and side-channel (Chapter 5) attacks. For integrity,

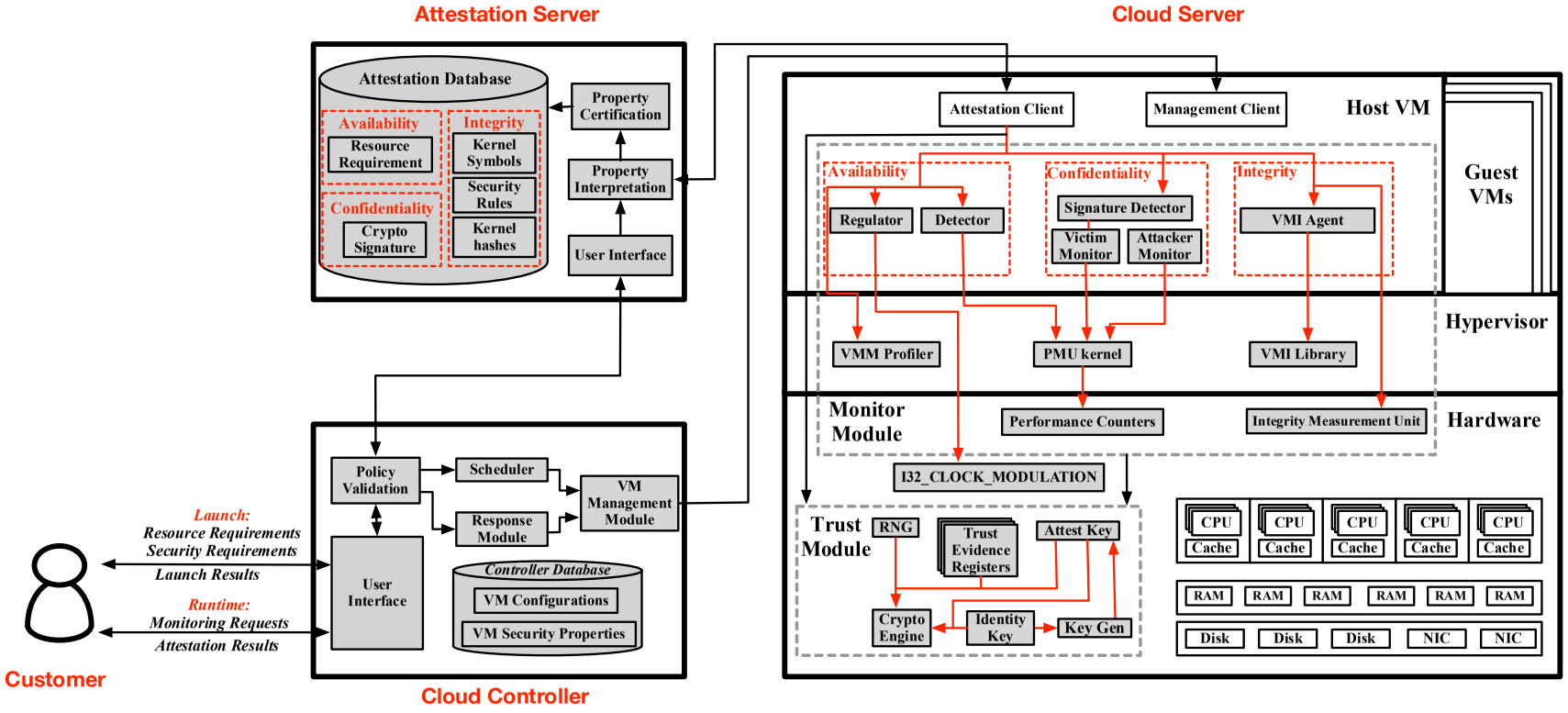


Figure 7.1: *CloudMonatt* can integrate methods from this dissertation to detect and mitigate vulnerabilities in cloud computing.

the cloud server uses the `Integrity Measurement Unit` to measure startup integrity (Section 3.3.1) and the `VMI Library` to monitor runtime system integrity (Chapter 6). These hypervisor and hardware components of the `Monitor Module` are invoked through trusted software interfaces like `Regulator` and `Detector` (Chapter 4), `Signature Detector`, `Victim Monitor` and `Attacker Monitor` (Chapter 5), `VMI Agent` (Chapter 6). A cloud server in *CloudMonatt* does not require new hardware: the `Hardware Performance Counters` and `I32_CLOCK_MODULATION` exist ubiquitously in modern processors; Intel Trusted Execution Technology [10] has Dynamic Root of Trust for Measurement, which can be used as the `Integrity Measurement Unit`; a Trusted Platform Module (TPM) chip can be used as the `Trust Module`.

The `Attestation Database` in the Attestation Server stores information required to interpret the security properties. For availability, the `Attestation Database` stores the resource requirement from the customer. For confidentiality, the `Attestation Database` stores the crypto signatures for crypto detection. For integrity, the `Attestation Database` stores the kernel hashes for startup integrity checking, kernel symbols and their addresses and security rules for VM runtime introspection. The `Controller Database` in the Cloud Controller stores VMs' configurations, and the security properties the customers request for their VMs.

7.2 Future Work

We now suggest some research directions that can extend the work done by this dissertation.

VM Security Health Monitoring and Attestation. In Chapter 3 we presented the *CloudMonatt* architecture. Section 3.5 conducted security verification of this architecture and pointed to several security requirements to protect *CloudMonatt*. As we mentioned in Section 3.5.4, we can exploit Bastion [61] or Intel SGX [155] to protect

the critical modules in the cloud server. Future work could be designing new security mechanisms using these secure architectures to realize the security requirements, and make *CloudMonatt* more secure.

To demonstrate the usage of *CloudMonatt*, we show several case studies in Section 3.3, as well as more sophisticated examples in Chapters 4, 5 and 6. These security threats are caused by the shared infrastructure, or the virtualized system itself. However, there are other types of attack vectors, as demonstrated in Section 2.1. One interesting direction would be proposing new solutions to defeat other types of security threats. For instance, the cloud provider can detect if customers' accounts have been hijacked by monitoring these accounts' behaviors using anomaly-based intrusion detection. This can defeat the service interface attacks. Another example is cloud abuse detection (Section 2.1.6). The cloud provider can monitor VMs' network activities and use signature-based intrusion detection to check if they are conducting some malicious behaviors (e.g., performing DoS, port scan or password guessing attacks). By doing so the cloud provider can prevent these VMs from being hijacked by malicious parties and becoming botnets. These new services can protect the VMs in a more comprehensive way.

Detection and Mitigation of Availability Vulnerabilities. Chapter 4 only considers the memory DoS attacks. However, there are other resources shared by co-tenant VMs, like network or disk. The severity of these resource DoS attacks are unknown. Future work includes investigating the feasibilities of host-based DoS attacks targeting other shared resources. Besides, we can also extend the defense method (Section 4.4) to other types of resource contention. For instance, we can monitor the protected VM's network or disk bandwidth and use the same statistical test to judge if it is under attack. Then we can use I/O bandwidth throttling to mitigate the attack damage. Such extensions will make this method more general and powerful.

Detection and Mitigation of Confidentiality Vulnerabilities. We show how to preserve VM's confidentiality against cache side-channel attacks in Chapter 5. However, there might be other types of side-channel information leakage. For the micro-architectural aspect, CPU pipeline, prefetcher, or DRAM could be potential side-channel media. It would be interesting to extend the detection method to these side channels. In addition, information could be leaked via power or network side channels. Detection of such side-channel attacks would also be challenging and interesting. A general question to solve in the future is whether we can design and validate a method that can identify all kinds of known and unknown side-channel attacks.

Detection and Mitigation of Integrity Vulnerabilities. In Chapter 6 we use VM introspection to monitor activities inside the VM. We considered mainly attacks within the guest VM. However, the hypervisor also has vulnerabilities. An attacker can exploit these vulnerabilities to gain root privilege and take control of the hypervisor. So as future work regarding VM integrity protection, we can consider how the monitoring tools can protect the hypervisor and detect if it is under attack? Also, how to protect the monitoring tools from being compromised by the attacker who gains root privilege on the cloud server?

In summary, this dissertation designs *CloudMonatt*, a general-purpose architectural framework to monitor and protect VMs' security health on behalf of customers. As case studies, we consider three types of security threats and demonstrate the methods to defeat them. First, we consider resource availability threats from co-located VMs. We design and evaluate a set of DoS attacks. We also design a novel method to detect and mitigate these threats using existing hardware features. Second, we consider side-channel confidentiality threats from co-located VMs. We propose a new method to detect the existence of cross-VM side-channel information leakage and use VM migration to mitigate these threats. Third, we consider system integrity vulnerability. We design different types of security tools to monitor and maintain a VM's system

integrity from the hypervisor. *CloudMonatt* is flexible enough to support other security protection mechanisms as well. We hope that new detection and mitigation methods can be proposed and integrated into *CloudMonatt* in the future, to make cloud systems more secure.

Bibliography

[1] Ab — The Apache Software Foundation. http://httpd.apache.org/docs/2.2/programs/ab.html.

[2] Amazon CloudWatch. https://aws.amazon.com/cloudwatch/.

[3] Amazon Inspector. https://aws.amazon.com/inspector/.

[4] Amazon Virtual Private Cloud. https://aws.amazon.com/vpc/.

[5] AMD Architecture Programmer's Manual, Volume 2: System Programming. https://support.amd.com/TechDocs/24593.pdf.

[6] ARM Cortex-A9 Technical Reference Manual, Revision r2p0. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0388e/BEHEDIHI.html.

[7] Common Vulnerabilities and Exposures. https://cve.mitre.org/.

[8] Improving Real-Time Performance by Utilizing Cache Allocation Technology. http://www.intel.com/content/www/us/en/communications/cache-allocation-technology-white-paper.html.

[9] Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3: System Programming Guide. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[10] Intel Trusted Execution Technology. http://http://www.intel.com/content/www/us/en/architecture-and-technology/trusted-execution-technology/malware-reduction-general-technology.html/.

[11] Libbdvmi. https://github.com/razvan-cojocaru/libbdvmi.

[12] LibVMI. http://libvmi.com.

[13] Magento: eCommerce Software and eCommerce Platform. http://www.magento.com/.

[14] Memtier Benchmark. https://github.com/RedisLabs/memtier_benchmark.

[15] Microsoft Antimalware for Azure Cloud Services and Virtual Machines. https://docs.microsoft.com/en-us/azure/security/azure-security-antimalware.

[16] Microsoft Azure Application Insights. https://azure.microsoft.com/en-us/services/application-insights/.

[17] OpenAttestation Project. https://wiki.openstack.org/wiki/OpenAttestation.

[18] Openstack Ceilometer. https://wiki.openstack.org/wiki/Ceilometer.

[19] Openstack Cloud Software. http://www.openstack.org/.

[20] OpenStack Monasca. https://wiki.openstack.org/wiki/Monasca.

[21] Openstack Security Hardening: Trusted Computing Pools. http://docs.openstack.org/admin-guide/compute-security.html.

[22] SPEC CPU 2006. https://www.spec.org/cpu2006/.

[23] Stackdriver Monitoring. https://cloud.google.com/monitoring/.

[24] Sysbench: a System Performance Benchmark. https://launchpad.net/sysbench/.

[25] Welcome to the Httperf Homepage. http://www.hpl.hp.com/research/linux/httperf/.

[26] Xentrace: Capture Xen Trace Buffer Data. https://linux.die.net/man/8/xentrace.

[27] Sherly Abraham and InduShobha Chengalur-Smith. An Overview of Social Engineering Malware: Trends, Tactics, and Implications. *Technology in Society*, 2010.

[28] Irfan Ahmed, Aleksandar Zoranic, Salman Javaid, Golden Richard, and Vassil Roussev. Rule-Based Integrity Checking of Interrupt Descriptor Tables in Cloud Environments. In *IFIP International Conference on Digital Forensics*.

[29] Jeongseob Ahn, Changdae Kim, Jaeung Han, Young-Ri Choi, and Jaehyuk Huh. Dynamic Virtual Machine Scheduling in Clouds for Architectural Shared Resources. In *USENIX Conference on Hot Topics in Cloud Computing*, 2012.

[30] Masoom Alam, Xinwen Zhang, Mohammad Nauman, Tamleek Ali, and Jean-Pierre Seifert. Model-based Behavioral Attestation. In *ACM Symposium on Access Control Models and Technologies*, 2008.

[31] Suaad Alarifi and Stephen D. Wolthusen. Robust Coordination of Cloud-Internal Denial of Service Attacks. In *International Conference on Cloud and Green Computing*, 2013.

[32] Ittai Anati, Shay Gueron, Simon P Johnson, and Vincent R Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *ACM International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[33] Frederik Armknecht, Ludovic Barman, Jens-Matthias Bohli, and Ghassan O. Karame. Mirror: Enabling Proofs of Data Replication and Retrievability in the Cloud. In *USENIX Security Symposium*, 2016.

[34] Ahmed M. Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C. Skalsky. HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity. In *ACM Conference on Computer and Communications Security*, 2010.

[35] Ahmed M. Azab, Peng Ning, and Xiaolan Zhang. SICE: A Hardware-level Strongly Isolated Computing Environment for x86 Multi-core Platforms. In *ACM Conference on Computer and Communications Security*, 2011.

[36] Yossi Azar, Seny Kamara, Ishai Menache, Mariana Raykova, and Bruce Shepard. Co-Location-Resistant Clouds. In *ACM Workshop on Cloud Computing Security*, 2014.

[37] Hyun wook Baek, Abhinav Srivastava, and Jacobus Van der Merwe. CloudVMI: Virtual Machine Introspection As a Cloud Service. In *IEEE International Conference on Cloud Engineering*, 2014.

[38] Mohammad Bagher Bahador, Mahdi Abadi, and Asghar Tajoddin. HPCMal-Hunter: Behavioral Malware Detection Using Hardware Performance Counters and Singular Value Decomposition. In *IEEE International Conference on Computer and Knowledge Engineering*, 2014.

[39] Marco Balduzzi, Jonas Zaddach, Davide Balzarotti, Engin Kirda, and Sergio Loureiro. A Security Analysis of Amazon's Elastic Compute Cloud Service. In *ACM Symposium on Applied Computing*, 2012.

[40] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Efficient Detection of Split Personalities in Malware. In *Network Distributed System Security Symposium*, 2010.

[41] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. *ACM SIGOPS Operating Systems Review*, 2003.

[42] Adam Bates, Benjamin Mood, Joe Pletcher, Hannah Pruse, Masoud Valafar, and Kevin Butler. Detecting Co-residency with Active Traffic Analysis Techniques. In *ACM Workshop on Cloud Computing Security*, 2012.

[43] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *USENIX Conference on Operating Systems Design and Implementation*, 2014.

[44] Harkeerat Singh Bedi and Sajjan Shiva. Securing Cloud Infrastructure Against Co-resident DoS Attacks Using Game Theoretic Defense Mechanisms. In *International Conference on Advances in Computing, Communications and Informatics*, 2012.

[45] Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vTPM: Virtualizing the Trusted Platform Module. In *USENIX Security Symposium*, 2006.

[46] J. Bergeron, M. Debbabi, M. M. Erhioui, and B. Ktari. Static Analysis of Binary Code to Isolate Malicious Behaviors. In *IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, 1999.

[47] Jean Bergeron, Mourad Debbabi, Jules Desharnais, Mourad M Erhioui, Yvan Lavoie, and Nadia Tawbi. Static Detection of Malicious Code in Executable Programs. *International Journal of Requirements Engineering*, 2001.

[48] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

[49] Bitdefender. Hypervisor Introspection. `http://www.bitdefender.com/business/hypervisor-introspection.html`.

[50] Sergey Blagodurov, Sergey Zhuravlev, Alexandra Fedorova, and Ali Kamali. A Case for NUMA-aware Contention Management on Multicore Systems. In *ACM International Conference on Parallel Architectures and Compilation Techniques*.

[51] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier based on Prolog Rules. In *IEEE Computer Security Foundations Workshop*, 2001.

[52] Bruno Blanchet. Security Protocol Verification: Symbolic and Computational Models. In *International Conference on Principles of Security and Trust*, 2012.

[53] Kevin D. Bowers, Ari Juels, and Alina Oprea. HAIL: A High-availability and Integrity Layer for Cloud Storage. In *ACM Conference on Computer and Communications Security*, 2009.

[54] Kevin D. Bowers, Marten van Dijk, Ari Juels, Alina Oprea, and Ronald L. Rivest. How to Tell if Your Cloud Files Are Vulnerable to Drive Crashes. In *ACM Conference on Computer and Communications Security*, 2011.

[55] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. CAn't Touch This: Practical and Generic Software-only Defenses Against Rowhammer Attacks. *arXiv preprint arXiv:1611.08396*, 2016.

[56] Ernie Brickell, Jan Camenisch, and Liqun Chen. Direct Anonymous Attestation. In *ACM Conference on Computer and Communications Security*, 2004.

[57] Sven Bugiel, Stefan Nürnberger, Thomas Pöppelmann, Ahmad-Reza Sadeghi, and Thomas Schneider. AmazonIA: When Elasticity Snaps Back. In *ACM Conference on Computer and Communications Security*, 2011.

[58] Shakeel Butt, H. Andrés Lagar-Cavilla, Abhinav Srivastava, and Vinod Ganapathy. Self-service Cloud Computing. In *ACM Conference on Computer and Communications Security*, 2012.

[59] Martim Carbone, Matthew Conover, Bruce Montague, and Wenke Lee. Secure and Robust Monitoring of Virtual Machines Through Guest-assisted Introspection. In *International Conference on Research in Attacks, Intrusions, and Defenses*, 2012.

[60] David Champagne. *Scalable Security Architecture for Trusted Software*. PhD thesis, Princeton University, 2010.

[61] David Champagne and Ruby.B. Lee. Scalable Architectural Support for Trusted Software. In *International Symposium on High Performance Computer Architecture*, 2010.

[62] Jie Chen and Guru Venkataramani. CC-Hunter: Uncovering Covert Timing Channels on Shared Processor Hardware. In *IEEE International Symposium on Microarchitecture*, 2014.

[63] Liqun Chen, Rainer Landfermann, Hans Löhr, Markus Rohe, Ahmad-Reza Sadeghi, and Christian Stüble. A Protocol for Property-based Attestation. In *ACM Workshop on Scalable Trusted Computing*.

[64] Liqun Chen and Jiangtao Li. Flexible and Scalable Digital Signatures in TPM 2.0. In *ACM Conference on Computer and Communications Security*, 2013.

[65] Liqun Chen, Hans Löhr, Mark Manulis, and Ahmad-Reza Sadeghi. Property-Based Attestation Without a Trusted Third Party. In *International Conference on Information Security*, 2008.

[66] Yu-Yuan Chen, Pramod A. Jamkhedkar, and Ruby B. Lee. A Software-hardware Architecture for Self-protecting Data. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2012.

[67] Ron C. Chiang, Sundaresan Rajasekaran, Nan Zhang, and H.Howie Huang. Swiper: Exploiting Virtual Machine Vulnerability in Third-Party Clouds with Competition for I/O Resources. *IEEE Transactions on Parallel and Distributed Systems*, 2015.

[68] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real Time Detection of Cache-based Side-channel Attacks Using Hardware Performance Counters. *Applied Soft Computing*, 2016.

[69] Mihai Christodorescu and Somesh Jha. Static Analysis of Executables to Detect Malicious Patterns. In *USENIX Security Symposium*, 2003.

[70] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-Aware Malware Detection. In *IEEE Symposium on Security and Privacy*, 2005.

[71] Jeroen V. Cleemput, Bart Coppens, and Bjorn De Sutter. Compiler Mitigations for Time Attacks on Modern x86 Processors. *ACM Transactions on Architecture Code Optimization*, 2012.

[72] George Coker, Joshua Guttman, Peter Loscocco, Amy Herzog, Jonathan Millen, Brian Hanlon, John Ramsdell, Ariel Segall, Justin Sheehy, and Brian Sniffen. Principles of Remote Attestation. *International Journal of Information Security.*, 10(2), June 2011.

[73] Henry Cook, Miquel Moreto, Sarah Bird, Khanh Dao, David A. Patterson, and Krste Asanovic. A Hardware Evaluation of Cache Partitioning to Improve Utilization and Energy-efficiency While Preserving Responsiveness. In *International Symposium on Computer Architecture*, 2013.

[74] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. In *Network Distributed System Security Symposium*, 2015.

[75] Team Cymru. Malware Hash Registry. `https://www.team-cymru.com/malware-data.html`.

[76] Jonathan Dautrich, Emil Stefanov, and Elaine Shi. Burst ORAM: Minimizing ORAM Response Times for Bursty Access Patterns. In *USENIX Security Symposium*, 2014.

[77] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[78] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the Feasibility of Online Malware Detection with Performance Counters. In *ACM International Symposium on Computer Architecture*, 2013.

[79] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *ACM Conference on Computer and Communications Security*, 2008.

[80] 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.

[81] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In *IEEE Symposium on Security and Privacy*, 2011.

[82] Danny Dolev and Andrew C. Yao. On the Security of Public Key Protocols. Technical report, Stanford University, 1981.

[83] 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*, 2012.

[84] Maximillian Dornseif, Thorsten Holz, and Christian N. Klein. NoSEBrEaK - Attacking Honeynets. In *IEEE Information Assurance Workshop*, 2004.

[85] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience, 2000.

[86] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. Fairness via Source Throttling: A Configurable and High-performance Fairness Substrate for Multi-core Memory Systems. In *Architectural Support for Programming Languages and Operating Systems*, 2010.

[87] Daniel R. Ellis, John G. Aiken, Kira S. Attwood, and Scott D. Tenaglia. A Behavioral Approach to Worm Detection. In *ACM Workshop on Rapid Malcode*, 2004.

[88] Paul England and Jork Loeser. Para-Virtualized TPM Sharing. In *Trusted Computing — Challenges and Applications*, 2008.

[89] EPFL. CloudSuite. `http://parsa.epfl.ch/cloudsuite/cloudsuite.html`.

[90] F-Secure. Backdoor:W32/Agobot. `https://www.f-secure.com/v-descs/agobot.shtml`.

[91] Zhenqian Feng, Bing Bai, Baokang Zhao, and Jinshu Su. Shrew Attack in Cloud Data Center Networks. In *International Conference on Mobile Ad-hoc and Sensor Networks*, 2011.

[92] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, 2000.

[93] Yangchun Fu and Zhiqiang Lin. Space Traveling Across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. In *IEEE Symposium on Security and Privacy*, 2012.

[94] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A Virtual Machine-based Platform for Trusted Computing. In *ACM Symposium on Operating Systems Principles*, 2003.

[95] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Network and Distribution System Security Symposium*, 2003.

[96] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. Sprobes: Enforcing Kernel Code Integrity on the Trustzone Architecture. In *IEEE Mobile Security Technologies Workshop*, 2014.

[97] Martin Georgiev and Vitaly Shmatikov. Gone in Six Characters: Short URLs Considered Harmful for Cloud Services. *arXiv preprint arXiv:1604.02734*, 2016.

[98] Yossi Gilad, Amir Herzberg, Michael Sudkovitch, and Michael Goberman. CDN-on-Demand: An Affordable DDoS Defense via Untrusted Clouds. In *Network Distributed System Security Symposium*, 2016.

[99] T. C. Group. TCG Software Stack Specification. `http://trustedcomputinggroup.org`, August 2003.

[100] T. C. Group. Design, Implementation, and Usage Principles for TPM-Based Platforms, May 2005.

[101] T. C. Group. TPM Library Specification. `http://www.trustedcomputinggroup.org/tpm-library-specification/`, October 2014.

[102] Top Threats Working Group. The Treacherous 12 Cloud Computing Top Threats in 2016. In *Cloud Security Alliance*, 2016.

[103] Dirk Grunwald and Soraya Ghiasi. Microarchitectural Denial of Service: Insuring Microarchitectural Fairness. In *ACM/IEEE International Symposium on Microarchitecture*, 2002.

[104] Nils Gruschka and Luigi Lo Iacono. Vulnerable Cloud: SOAP Message Security Validation Revisited. In *IEEE International Conference on Web Services*, 2009.

[105] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *Conference on Detection of Intrusions and Malware and Vulnerability Assessment*. 2016.

[106] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *Conference on Detection of Intrusions and Malware and Vulnerability Assessment*, 2016.

[107] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-level Caches. In *USENIX Security Symposium*, 2015.

[108] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *USENIX Conference on Operating Systems Design and Implementation*, 2016.

[109] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games — Bringing Access-Based Cache Attacks on AES to Practice. In *IEEE Symposium on Security and Privacy*, 2011.

[110] Arpan Gupta, Jack Sampson, and Michael Bedford Taylor. Quality Time: A Simple Online Technique for Quantifying Multicore Execution Efficiency. In *IEEE International Symposium on Performance Analysis of Systems and Software*, 2014.

[111] Vivek Haldar, Deepak Chandra, and Michael Franz. Semantic Remote Attestation: A Virtual Machine Directed Approach to Trusted Computing. In *Conference on Virtual Machine Research And Technology Symposium*, 2004.

[112] Halflife. Bypassing Integrity Checking Systems. *Phrack Magazine*, 7(51), 1997.

[113] Yi Han, Tansu Alpcan, Jeffrey Chan, and Christopher Leckie. Security Games for Virtual Machine Allocation in Cloud Computing. In *Decision and Game Theory for Security*. 2013.

[114] Nishad Herath and Anders Fogh. These Are Not Your Grand Daddy's CPU Performance Counters: CPU Hardware Performance Counters for Security. In *Black Hat USA*, 2015.

[115] Amir Herzberg, Haya Shulman, Johanna Ullrich, and Edgar Weippl. Cloudoscopy: Services Discovery and Topology Mapping. In *ACM Workshop on Cloud Computing Security*, 2013.

[116] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion Detection Using Sequences of System Calls. *Journal of Computer Security*, 1998.

[117] Qun Huang and Patrick P.C. Lee. An Experimental Study of Cascading Performance Interference in a Virtualized Environment. *ACM SIGMETRICS Performance Evaluation Review*, 2013.

[118] Joseph Idziorek, Mark Tannian, and Doug Jacobson. Detecting Fraudulent Use of Cloud Resources. In *ACM Workshop on Cloud Computing Security Workshop*, 2011.

[119] Kenneth Ingham and Stephanie Forrest. A History and Survey of Network Firewalls. Technical report, University of New Mexico, 2002.

[120] 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 *IEEE Symposium on Security and Privacy*, 2015.

[121] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. MASCAT: Stopping Microarchitectural Attacks Before Execution. Cryptology ePrint Archive, Report 2016/1196, 2016.

[122] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, Cross-VM attack on AES. In *Research in Attacks, Intrusions and Defenses*. Springer, 2014.

[123] Trent Jaeger, Reiner Sailer, and Umesh Shankar. PRIMA: Policy-reduced Integrity Measurement Architecture. In *ACM Symposium on Access Control Models and Technologies*, 2006.

[124] Pramod Jamkhedkar, Jakub Szefer, Diego Perez-Botero, Tianwei Zhang, Gina Triolo, and Ruby B. Lee. A Framework for Realizing Security on Demand in Cloud Computing. In *IEEE Conference on Cloud Computing Technology and Science*, 2013.

[125] Quan Jia, Huangxin Wang, Dan Fleck, Fei Li, Angelos Stavrou, and Walter Powell. Catch Me If You Can: A Cloud-Enabled DDoS Defense. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014.

[126] Xuxian Jiang and Xinyuan Wang. "Out-of-the-Box" Monitoring of VM-based High-interaction Honeypots. In *International Conference on Recent Advances in Intrusion Detection*, 2007.

[127] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy Malware Detection Through Vmm-based "Out-of-the-box" Semantic View Reconstruction. In *ACM Conference on Computer and Communications Security*, 2007.

[128] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. VMM-based Hidden Process Detection and Identification Using Lycosid. In *ACM International Conference on Virtual Execution Environments*, 2008.

[129] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B. Lee. NoHype: Virtualized Cloud Infrastructure Without the Virtualization. In *ACM International Symposium on Computer Architecture*, 2010.

[130] Daehoon Kim, Hwanju Kim, and Jaehyuk Huh. vCache: Providing a Transparent View of the LLC in Virtualized Environments. *Computer Architecture Letters*, 2014.

[131] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-level Protection Against Cache-based Side Channel Attacks in the Cloud. In *USENIX Security Symposium*, 2012.

[132] Yoongu Kim. Rowhammer Memtest. `https://github.com/CMU-SAFARI/rowhammer`.

[133] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *International Symposium on Computer Architecuture*, 2014.

[134] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *ACM SIGOPS Symposium on Operating Systems Principles*, 2009.

[135] Wenke Lee and Salvatore J. Stolfo. Data Mining Approaches for Intrusion Detection. In *USENIX Security Symposium*, 1998.

[136] Tamas K. Lengyel, Steve Maresca, Bryan D. Payne, George D. Webster, Sebastian Vogl, and Aggelos Kiayias. Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System. In *Annual Computer Security Applications Conference*, 2014.

[137] Chao Li, Zhenhua Wang, Xiaofeng Hou, Haopeng Chen, Xiaoyao Liang, and Minyi Guo. Power Attack Defense: Securing Battery-Backed Data Centers. In *ACM/IEEE International Symposium on Computer Architecture*, 2016.

[138] Min Li, Yulong Zhang, Kun Bai, Wanyu Zang, Meng Yu, and Xubin He. Improving Cloud Survivability through Dependency based Virtual Machine Placement. In *International Conference on Security and Cryptography*, 2012.

[139] Peng Li, Debin Gao, and Michael K. Reiter. StopWatch: A Cloud Architecture for Timing Channel Mitigation. *ACM Transactions on Information and System Security*, 2014.

[140] Wei-Jen Li, Ke Wang, S. J. Stolfo, and B. Herzog. Fileprints: Identifying File Types by n-gram Analysis. In *IEEE Information Assurance Workshop*, 2005.

[141] Fangfei Liu, Qian 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*, 2016.

[142] Fangfei Liu and Ruby B. Lee. Random Fill Cache Architecture. In *IEEE/ACM International Symposium on Microarchitecture*, 2014.

[143] Fangfei Liu, Hao Wu, Ken Mai, and Ruby B. Lee. Newcache: Secure Cache Architecture Thwarting Cache Side-Channel Attacks. *IEEE Micro*, 36(5), 2016.

[144] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE Symposium on Security and Privacy*, 2015.

[145] Huan Liu. A New Form of DoS Attack in a Cloud and Its Avoidance Mechanism. In *ACM Workshop on Cloud Computing Security Workshop*, 2010.

[146] Huan Liu. A Measurement Study of Server Utilization in Public Clouds. In *IEEE International Conference on Dependable, Autonomic and Secure Computing*, 2011.

[147] Lei Liu, Zehan Cui, Mingjie Xing, Yungang Bao, Mingyu Chen, and Chengyong Wu. A Software Memory Partition Approach for Eliminating Bank-level Interference in Multicore Systems. In *International Conference on Parallel Architectures and Compilation Techniques*, 2012.

[148] Peter A. Loscocco, Perry W. Wilson, J. Aaron Pendergrass, and C. Durward McDonell. Linux Kernel Integrity Measurement Using Contextual Inspection. In *ACM Workshop on Scalable Trusted Computing*, 2007.

[149] Matteo Maffei, Giulio Malavolta, Manuel Reinert, and Dominique Schroder. Privacy and Access Control for Outsourced Personal Records. In *IEEE Symposium on Security and Privacy*, 2015.

[150] Corey Malone, Mohamed Zahran, and Ramesh Karri. Are Hardware Performance Counters a Cost Effective Way for Integrity Checking of Programs. In *ACM Workshop on Scalable Trusted Computing*, 2011.

[151] Heiko Mantel and Artem Starostin. Transforming Out Timing Leaks, More or Less. In *European Symposium on Computer Security*, 2015.

[152] Frank J Massey Jr. The Kolmogorov-Smirnov Test for Goodness of Fit. *Journal of the American Statistical Association*, 1951.

[153] John D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. `http://www.cs.virginia.edu/stream/`.

[154] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Symposium on Security and Privacy*, 2010.

[155] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *ACM International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[156] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. In *NIST Special Publication*, 2011.

[157] Rui Miao, Rahul Potharaju, Minlan Yu, and Navendu Jain. The Dark Menace: Characterizing Network-based Attacks in the Cloud. In *ACM Conference on Internet Measurement Conference*, 2015.

[158] Zhen Mo, Qingjun Xiao, Yian Zhou, and Shigang Chen. On Deletion of Outsourced Data in Cloud Computing. In *IEEE International Conference on Cloud Computing*, 2014.

[159] Soo-Jin Moon, Vyas Sekar, and Michael K. Reiter. Nomad: Mitigating Arbitrary Cloud Side Channels via Provider-Assisted Migration. In *ACM Conference on Computer and Communications Security*, 2015.

[160] Thomas Moscibroda and Onur Mutlu. Memory Performance Attacks: Denial of Memory Service in Multi-core Systems. In *USENIX Security Symposium*, 2007.

[161] Martin Mulazzani, Sebastian Schrittwieser, Manuel Leithner, Markus Huber, and Edgar Weippl. Dark Clouds on the Horizon: Using Cloud Storage As Attack Vector and Online Slack Space. In *USENIX Security Symposium*, 2011.

[162] Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda. Reducing Memory Interference in Multicore Systems via Application-aware Memory Channel Partitioning. In *ACM/IEEE International Symposium on Microarchitecture*, 2011.

[163] Aarthi Nagarajan, Vijay Varadharajan, Michael Hitchens, and Eimear Gallery. Property Based Attestation and Trusted Computing: Analysis and Challenges. In *International Conference on Network and System Security*, 2009.

[164] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *USENIX Conference on Annual Technical Conference*, 2013.

[165] Keisuke Okamura and Yoshihiro Oyama. Load-based Covert Channels Between Xen Virtual Machines. In *ACM Symposium on Applied Computing*, 2010.

[166] Yoshinori Okazaki, Izuru Sato, and Shigeki Goto. A New Intrusion Detection Method Based on Process Profiling. In *Symposium on Applications and the Internet*, 2002.

[167] Rolf Oppliger. Internet Security: Firewalls and Beyond. *Communications of the ACM*, 1997.

[168] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In *RSA Conference on Topics in Cryptology*, pages 1–20, 2006.

[169] Bryan D. Payne, Martim Carbone, and Wenke Lee. Secure and Flexible Monitoring of Virtual Machines. In *Annual Computer Security Applications Conference*, 2007.

[170] Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *IEEE Symposium on Security and Privacy*, 2008.

[171] Gábor Pék, Andrea Lanzi, Abhinav Srivastava, Davide Balzarotti, Aurélien Francillon, and Christoph Neumann. On the Feasibility of Software Attacks on Commodity Virtual Machine Monitors via Direct Device Assignment. In *ACM Symposium on Information, Computer and Communications Security*, 2014.

[172] Colin Percival. Cache Missing for Fun and Profit. In *BSDCan*, 2005.

[173] Diego Perez-Botero, Jakub Szefer, and Ruby B. Lee. Characterizing Hypervisor Vulnerabilities in Cloud Computing Servers. In *International Workshop on Security in Cloud Computing*, 2013.

[174] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium*, 2016.

[175] Nick L. Petroni, Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot — a Coprocessor-based Kernel Runtime Integrity Monitor. In *USENIX Security Symposium*, 2004.

[176] Nicolas Poggi, David Carrera, Ricard Gavalda, and Eduard Ayguade. Non-intrusive Estimation of QoS Degradation Impact on E-Commerce User Satisfaction. In *IEEE Internatiopnal Symposium on Network Computing and Applications*, 2011.

[177] Jonathan Poritz, Matthias Schunter, Els Van Herreweghen, and Michael Waidner. Property Attestation — Scalable and Privacy-friendly Security Assessment of Peer Computers. Technical report, IBM Research, 2004.

[178] Nguyen Anh Quynh and Yoshiyasu Takefuji. Towards a Tamper-resistant Kernel Rootkit Detector. In *ACM Symposium on Applied Computing*, 2007.

[179] Arthur Rahumed, Henry C. H. Chen, Yang Tang, Patrick P. C. Lee, and John C. S. Lui. A Secure Cloud Backup System with Assured Deletion and Version Control. In *International Conference on Parallel Processing Workshops*, 2011.

[180] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. Resource Management for Isolation Enhanced Cloud Services. In *ACM Workshop on Cloud Computing Security*, 2009.

[181] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing Digital Side-channels Through Obfuscated Execution. In *USENIX Security Symposium*, 2015.

[182] 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 *ACM Conference on Computer and Communications Security*, 2009.

[183] Francisco Rocha and Miguel Correia. Lucy in the Sky Without Diamonds: Stealing Confidential Data in the Cloud. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*, 2011.

[184] Ahmad-Reza Sadeghi and Christian Stüble. Property-based Attestation for Computing Platforms: Caring About Properties, Not Mechanisms. In *Workshop on New Security Paradigms*, 2004.

[185] Ahmad-Reza Sadeghi, Christian Stüble, and Marcel Winandy. Property-Based TPM Virtualization. In *International Conference on Information Security*, 2008.

[186] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *USENIX Security Symposium*, 2004.

[187] Hiroaki Sakoe and Seibi Chiba. Dynamic Programming Algorithm Optimization for Spoken Word Recognition. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 1978.

[188] Daniel Sanchez and Christos Kozyrakis. Vantage: Scalable and Efficient Fine-grain Cache Partitioning. In *AMC International Symposium on Computer Architecture*, 2011.

[189] Nuno Santos, Rodrigo Rodrigues, Krishna P. Gummadi, and Stefan Saroiu. Policy-sealed Data: A New Abstraction for Building Trusted Cloud Services. In *USENIX Security Symposium*, 2012.

[190] Vincent Scarlata, Carlos Rozas, Monty Wiseman, David Grawrock, and Claire Vishik. TPM Virtualization: Building a General Framework. In *Trusted Computing*. 2008.

[191] Joshua Schiffman, Thomas Moyer, Hayawardh Vijayakumar, Trent Jaeger, and Patrick McDaniel. Seeding Clouds with Trust Anchors. In *ACM Workshop on Cloud Computing Security*, 2010.

[192] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *IEEE Symposium on Security and Privacy*, 2015.

[193] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In *IEEE Symposium on Security and Privacy*, 2001.

[194] R. Sekar, A. Gupta, J. Frullo, T. Shanbhag, A. Tiwari, H. Yang, and S. Zhou. Specification-based Anomaly Detection: A New Approach for Detecting Network Intrusions. In *ACM Conference on Computer and Communications Security*, 2002.

[195] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. In *ACM Symposium on Operating Systems Principles*, 2005.

[196] Ryan Shea and Jiangchuan Liu. Understanding the Impact of Denial of Service Attacks on Virtual Machines. In *IEEE International Workshop on Quality of Service*, 2012.

[197] Ryan Shea and Jiangchuan Liu. Performance of Virtual Machines Under Networked Denial of Service Attacks: Experiments and Analysis. *IEEE Systems Journal*, 2013.

[198] Timothy Sherwood, Erez Perelman, Greg Hamerly, Suleyman Sair, and Brad Calder. Discovering and Exploiting Program Phases. *IEEE Micro*, 2003.

[199] Elaine Shi, Adrian Perrig, and Leendert van Doorn. BIND: a Fine-grained Attestation Service for Secure Distributed Systems. In *IEEE Symposium on Security and Privacy*, 2005.

[200] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. Limiting Cache-based Side-channel in Multi-tenant Cloud using Dynamic Page Coloring. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*, 2011.

[201] Emin Gün Sirer, Willem de Bruijn, Patrick Reynolds, Alan Shieh, Kevin Walsh, Dan Williams, and Fred B. Schneider. Logical Attestation: An Authorization Architecture for Trustworthy Computing. In *ACM Symposium on Operating Systems Principles*, 2011.

[202] Gaurav Somani, Manoj Singh Gaur, and Dheeraj Sanghi. DDoS/EDoS Attack in Cloud: Affecting Everyone out There! In *International Conference on Security of Information and Networks*, 2015.

[203] Juraj Somorovsky, Mario Heiderich, Meiko Jensen, Jörg Schwenk, Nils Gruschka, and Luigi Lo Iacono. All Your Clouds Are Belong to Us: Security Analysis of Cloud Management Interfaces. In *ACM Workshop on Cloud Computing Security*, 2011.

[204] Deepa Srinivasan, Zhi Wang, Xuxian Jiang, and Dongyan Xu. Process Outgrafting: An Efficient "out-of-VM" Approach for Fine-grained Process Execution Monitoring. In *ACM Conference on Computer and Communications Security*, 2011.

[205] Emil Stefanov and Elaine Shi. Multi-cloud Oblivious Storage. In *ACM Conference on Computer Communications Security*, 2013.

[206] Emil Stefanov and Elaine Shi. ObliviStore: High Performance Oblivious Cloud Storage. In *IEEE Symposium on Security and Privacy*, 2013.

[207] Emil Stefanov, Elaine Shi, and Dawn Song. Towards Practical Oblivious RAM. In *Network Distributed System Security Symposium*, 2012.

[208] Emil Stefanov, Marten van Dijk, Ari Juels, and Alina Oprea. Iris: A Scalable Cloud File System with Efficient Integrity Checks. In *ACM Annual Computer Security Applications Conference*, 2012.

[209] Mario Strasser and Heiko Stamer. A Software-based Trusted Platform Module Emulator. In *Trusted Computing — Challenges and Applications*. Springer, 2008.

[210] Frederic Stumpf, Omid Tafreschi, Patrick Röder, and Claudia Eckert. A Robust Integrity Reporting Protocol for Remote Attestation. In *Workshop on Advances in Trusted Computing*, 2006.

[211] Y. Sun, G. Petracca, T. Jaeger, H. Vijayakumar, and J. Schiffman. Cloud Armor: Protecting Cloud Commands from Compromised Cloud Services. In *IEEE International Conference on Cloud Computing*, 2015.

[212] Yuqiong Sun, Giuseppe Petracca, Xinyang Ge, and Trent Jaeger. Pileus: Protecting User Resources From Vulnerable Cloud Services. In *Annual Conference on Computer Security Applications*, 2016.

[213] Yuqiong Sun, Giuseppe Petracca, and Trent Jaeger. Inevitable Failure: The Flawed Trust Assumption in the Cloud. In *ACM Workshop on Cloud Computing Security*, 2014.

[214] A. H. Sung, J. Xu, P. Chavez, and S. Mukkamala. Static Analyzer of Vicious Executables (SAVE). In *Annual Computer Security Applications Conference*, 2004.

[215] Wai Kit Sze, Abhinav Srivastava, and R. Sekar. Hardening OpenStack Cloud Platforms Against Compute Node Compromises. In *ACM Asia Conference on Computer and Communications Security*, 2016.

[216] Jakub Szefer. *Architectures for Secure Cloud Computing Servers*. PhD thesis, Princeton University, 2013.

[217] Jakub Szefer, Eric Keller, Ruby B. Lee, and Jennifer Rexford. Eliminating the Hypervisor Attack Surface for a More Secure Cloud. In *ACM Conference on Computer and Communications Security*, 2011.

[218] Jakub Szefer and Ruby B. Lee. BitDeposit: Deterring Attacks and Abuses of Cloud Computing Services through Economic Measures. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2013.

[219] Peter Szor. *The Art of Computer Virus Research and Defense*. Pearson Education, 2005.

[220] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. Unsupervised Anomaly-based Malware Detection Using Hardware Features. In *Research in Attacks, Intrusions and Defenses*. 2014.

[221] Yang Tang, Patrick P. C. Lee, John C. S. Lui, and Radia Perlman. FADE: Secure Overlay Cloud Storage with File Assured Deletion. In *International Conference on Security and Privacy in Communication Networks*, 2010.

[222] Carol Taylor and Jim Alves-Foss. NATE: Network Analysis of Anomalous Traffic Events, a Low-cost Approach. In *Workshop on New Security Paradigms*, 2001.

[223] Marten van Dijk, Ari Juels, Alina Oprea, Ronald L. Rivest, Emil Stefanov, and Nikos Triandopoulos. Hourglass Schemes: How to Prove That Cloud Files Are Encrypted. In *ACM Conference on Computer and Communications Security*, 2012.

[224] Venkatanathan Varadarajan, Thawan Kooburat, Benjamin Farley, Thomas Ristenpart, and Michael M. Swift. Resource-freeing Attacks: Improve Your Cloud Performance (at Your Neighbor's Expense). In *ACM Conference on Computer and Communications Security*, 2012.

[225] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. Scheduler-based Defenses Against Cross-VM Side-channels. In *Usenix Security Symposium*, 2014.

[226] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. A Placement Vulnerability Study in Multi-Tenant Public Clouds. In *USENIX Security Symposium*, 2015.

[227] Amit Vasudevan, Sagar Chaki, Limin Jia, Jonathan McCune, James Newsome, and Anupam Datta. Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework. In *IEEE Symposium on Security and Privacy*, 2013.

[228] Amit Vasudevan, Sagar Chaki, Petros Maniatis, Limin Jia, and Anupam Datta. überSpark: Enforcing Verifiable Object Abstractions for Automated Compositional Security Analysis of a Hypervisor. In *USENIX Security Symposium*, 2016.

[229] Bhanu C. Vattikonda, Sambit Das, and Hovav Shacham. Eliminating Fine Grained Timers in Xen. In *ACM Workshop on Cloud Computing Security*, 2011.

[230] Michael Velten and Frederic Stumpf. Secure and Privacy-aware Multiplexing of Hardware-protected TPM Integrity Measurements Among Virtual Machines. In *International Conference on Information Security and Cryptology*, 2013.

[231] VMWare. Understanding Full Virtualization, Paravirtualization, and Hardware Assist. `http://linuxcourse.rutgers.edu/documents/VMware_paravirtualization.pdf`.

[232] Hui Wang, Canturk Isci, Lavanya Subramanian, Jongmoo Choi, Depei Qian, and Onur Mutlu. A-DRM: Architecture-aware Distributed Resource Management of Virtualized Clusters. In *ACM International Conference on Virtual Execution Environments*, 2015.

[233] Ke Wang and Salvatore J Stolfo. Anomalous Payload-based Network Intrusion Detection. In *Recent Advances in Intrusion Detection*, 2004.

[234] Xueyang Wang and R. Karri. NumChecker: Detecting Kernel Control-Flow Modifying Rootkits by Using Hardware Performance Counters. In *ACM/EDAC/IEEE Design Automation Conference*, 2013.

[235] Xueyang Wang, Charalambos Konstantinou, Michail Maniatakos, and Ramesh Karri. ConFirm: Detecting Firmware Modifications in Embedded Systems Using Hardware Performance Counters. In *IEEE/ACM International Conference on Computer-Aided Design*, 2015.

[236] Yao Wang, Andrew Ferraiuolo, and G. Edward Suh. Timing Channel Protection for a Shared Memory Controller. In *IEEE International Symposium on High Performance Computer Architecture*, 2014.

[237] Yi-Min Wang, Doug Beck, Binh Vo, Roussi Roussev, and Chad Verbowski. Detecting Stealth Software with Strider GhostBuster. In *International Conference on Dependable Systems and Networks*, 2005.

[238] Zhenghong Wang and Ruby B. Lee. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *ACM International Symposium on Computer Architecture*, 2007.

[239] Zhenghong Wang and Ruby.B. Lee. A Novel Cache Architecture with Enhanced Performance and Security. In *IEEE/ACM International Symposium on Microarchitecture*, 2008.

[240] Zhi Wang and Xuxian Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *IEEE Symposium on Security and Privacy*, 2010.

[241] Jinpeng Wei, Xiaolan Zhang, Glenn Ammons, Vasanth Bala, and Peng Ning. Managing Security of Virtual Machine Images in a Cloud Environment. In *ACM Workshop on Cloud Computing Security*, 2009.

[242] Dong Hyuk Woo and Hsien-Hsin S Lee. Analyzing Performance Vulnerability due to Resource Denial-of-Service Attack on Chip Multiprocessors. In *Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2007.

[243] Hao Wu, Fangfei Liu, and Ruby B. Lee. PALMScloud: Cloud Server Benchmark Suite for Evaluating New Hardware Architectures. *IEEE Computer Architecture Letters*, PP(99), 2016.

[244] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *USENIX Security Symposium*, 2012.

[245] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. CFIMon: Detecting Violation of Control Flow Integrity Using Performance Counters. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2012.

[246] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In *USENIX Security Symposium*, 2016.

[247] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. An Exploration of L2 Cache Covert Channels in Virtualized Environments. In *ACM Workshop on Cloud Computing Security*, 2011.

[248] Zhang Xu, Haining Wang, and Zhenyu Wu. A Measurement Study on Co-residence Threat inside the Cloud. In *USENIX Security Symposium*, 2015.

[249] Zhang Xu, Haining Wang, Zichen Xu, and Xiaorui Wang. Power Attack: An Increasing Threat to Data Centers. In *Network Distributed System Security Symposium*, 2014.

[250] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. Secure Hierarchy-Aware Cache Replacement Policy (SHARP): Defending Against Cache-Based Side Channel Atacks. In *ACM International Symposium on Computer Architecture*, 2017.

[251] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *ACM International Symposium on Computer Architecture*, 2013.

[252] Kan Yang, Xiaohua Jia, and Kui Ren. Attribute-based Fine-grained Access Control with Efficient Revocation in Cloud Storage Systems. In *ACM Symposium on Information Computer and Communications Security*, 2013.

[253] Ziye Yang, Haifeng Fang, Yingjun Wu, Chungi Li, Bin Zhao, and H.H. Huang. Understanding the Effects of Hypervisor I/O Scheduling for Virtual Machine Performance Interference. In *IEEE International Conference on Cloud Computing Technology and Science*, 2012.

[254] Fangzhou Yao, Read Sprabery, and Roy H. Campbell. CryptVMI: A Flexible and Encrypted Virtual Machine Introspection System in the Cloud. In *International Workshop on Security in Cloud Computing*, 2014.

[255] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *USENIX Security Symposium*, 2014.

[256] Ben-Ami Yassour, Muli Ben-Yehuda, and Orit Wasserman. Direct Device Assignment for Untrusted Fully-virtualized Virtual Machines. Technical report, IBM Research, 2008.

[257] Liwei Yuan, Weichao Xing, Haibo Chen, and Binyu Zang. Security Breaches As PMU Deviation: Detecting and Identifying Security Attacks Using Performance Counters. In *Asia-Pacific Workshop on Systems*, 2011.

[258] Zentific. LibVMI-based GDB Server for Virtual Machines. `https://github.com/Zentific/vmidbg`.

[259] Yan Zhai, Lichao Yin, Jeffrey Chase, Thomas Ristenpart, and Michael Swift. CQSTR: Securing Cross-Tenant Applications with Cloud Containers. In *ACM Symposium on Cloud Computing*, 2016.

[260] Kehuan Zhang, Xiaoyong Zhou, Yangyi Chen, XiaoFeng Wang, and Yaoping Ruan. Sedic: Privacy-aware Data Intensive Computing on Hybrid Clouds. In *ACM Conference on Computer and Communications Security*, 2011.

[261] Su Zhang, Xinwen Zhang, and Xinming Ou. After We Knew It: Empirical Study and Modeling of Cost-effectiveness of Exploiting Prevalent Known Vulnerabilities Across IaaS Cloud. In *ACM Symposium on Information, Computer and Communications Security*, 2014.

[262] Tianwei Zhang and Ruby B. Lee. CloudMonatt: An Architecture for Security Health Monitoring and Attestation of Virtual Machines in Cloud Computing. In *ACM International Symposium on Computer Architecture*, 2015.

[263] Tianwei Zhang and Ruby B. Lee. Monitoring and Attestation of Virtual Machine Security Health in Cloud Computing. *IEEE Micro*, 36(5), 2016.

[264] 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*. 2016.

[265] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. DoS Attacks on Your Memory in Cloud. In *ACM Asia Conference on Computer and Communications Security*, 2017.

[266] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Hardware Execution Throttling for Multi-core Resource Management. In *USENIX Annual Technical Conference*, 2009.

[267] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI2: CPU Performance Isolation for Shared Compute Clusters. In *ACM European Conference on Computer Systems*, 2013.

[268] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In *IEEE Symposium on Security and Privacy*, 2011.

[269] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *ACM Conference on Computer and Communications Security*, 2012.

[270] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *ACM Conference on Computer and Communications Security*, 2014.

[271] Yinqian Zhang and Michael K. Reiter. DüPpel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud. In *ACM Conference on Computer and Communications Security*, 2013.

[272] Yulong Zhang, Min Li, Kun Bai, Meng Yu, and Wanyu Zang. Incentive Compatible Moving Target Defense against VM-colocation Attacks in Clouds. In *Information Security and Privacy Research*. 2012.

[273] Yunqi Zhang, Michael A. Laurenzano, Jason Mars, and Lingjia Tang. SMiTe: Precise QoS Prediction on Real-System SMT Processors to Improve Utilization in Warehouse Scale Computers. In *IEEE/ACM International Symposium on Microarchitecture*, 2014.

[274] Fangfei Zhou, Manish Goel, Peter. Desnoyers, and Ravi Sundaram. Scheduler Vulnerabilities and Coordinated Attacks in Cloud Computing. In *EEE International Symposium on Network Computing and Applications*, 2011.

[275] Ziqiao Zhou, Michael K Reiter, and Yinqian Zhang. A Software Approach to Defeating Side Channels in Last-level Caches. In *ACM Conference on Computer and Communications Security*, 2016.

[276] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.