EXPLORING SECURITY SUPPORT FOR CLOUD-BASED APPLICATIONS

by

HAI NGUYEN

A dissertation submitted to the

School of Graduate Studies

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

For the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Vinod Ganapathy

And approved by

_____

_____

_____

_____

New Brunswick, New Jersey

October, 2018

ABSTRACT OF THE DISSERTATION

Exploring Security Support for Cloud-based Applications

By Hai Nguyen

Dissertation Director:

Vinod Ganapathy

Users are increasingly adopting cloud services for various purposes such as storing and processing data or using cloud-based software. However, this computing model poses cloud-specific security challenges to these cloud-based applications.

This dissertation describes novel solutions to three security problems of cloud-based applications. *First*, the introduction of hardware-based implementations of isolated execution such as Intel SGX makes it challenging to enforce security compliance of cloud applications. It is desirable to have a mechanism that allows cloud providers to inspect the code and data of cloud applications while still preserves the integrity and confidentiality offered by Intel SGX. *Second*, cloud services have increasingly become the target of ransomware attacks. However, current ransomware detection techniques are prone to false positives and some of them are unable to distinguish ransomware from benign programs that exhibit ransomware-like behaviors. *Third*, in today's cloud platforms, clients do not have much power and flexibility to deploy security services. Clients often rely heavily on cloud providers for deployment of security measures such as intrusion detection systems (IDSs) or have to manually install and configure software stack with security tools.

This dissertation makes the following contributions. *First*, it implements EnGarde, an enclave inspection library that preserves the security and privacy benefits offered by Intel SGX

and allows the cloud provider to verify the clients SGX-based enclave against predefined policies mutually agreed by the cloud provider and the client. *Second*, it builds HRD, a system that can detect ransomware in cloud-based environments with low false positives. HRD uses Hardware Performance Counters (HPCs) and machine learning to build classifiers that effectively detect ransomware with high accuracy. *Third*, it demonstrates the utility of a new cloud computing model where the client can make use of cloud apps, implemented as virtual machines (VMs), to implement security measures.

Acknowledgements

I would like to thank my advisor, Professor Vinod Ganapathy, for his tremendous guidance and support. I am really grateful to him for his insights and feedbacks that have helped shape many ideas in this dissertation. I am honored to work under Vinod's supervision.

I would like to express my gratitude to my committee members, Professor Thu Nguyen, Professor Abhishek Bhattacharjee, and Professor Trent Jaeger for providing me valuable feedbacks to improve this dissertation. I would also like to thank my collaborators, Dr. Pratyusa Manadhata, Dr. Abhinav Srivastava, and Shivaramakrishnan Vaidyanathan.

I have had the fortune of sharing my time at Rutgers with past and present members of DiscoLab. I especially thank Daeyoung Kim, Amruta Gokhale, Shakeel Butt, Rezwana Karim, Nader Boushehrinejadmoradi, Daehan Kwak, Ruilin Liu and Shivaramakrishnan Vaidyanathan. I would also like to thank my friends at Rutgers, especially Tam Vu, Binh Pham, Long Le, Minh Vu, Hang Dam, Tuyen Tran, Viet Nguyen, Trung Duong, Hai Pham,and Cuong Tran.

Last but not least, I am thankful to my parents, Thanh Tran and Quan Nguyen, my wife Thao Nguyen, my son William Nguyen and my daughter Anh Nguyen for their support and patience while I was working on this dissertation. I dedicate this dissertation to them.

Dedication

To my family

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation proposes novel solutions to three security and privacy problems of cloud-based applications. The *first* problem is finding a mechanism that allows the cloud provider to inspect the client's code within a hardware-based protected execution environment without relying on a trusted third party. The *second* problem is developing a new method that is able to detect ransomware in cloud computing environments with low false positives. The *third* problem is exploring a new cloud computing model where clients have more power to deploy security services.

1.1 Motivation

Users are increasingly adopting cloud services for various purposes such as storing and processing data or using cloud-based software. However, this computing model poses cloud-specific security challenges to these cloud-based applications [1, 2]. For applications that process sensitive content which should not be accessed by any other party including the cloud provider, the major concern is privacy as well as the integrity and confidentiality of their code and data. This stems from the fact that the clients using these applications do not have physical access to cloud infrastructures and therefore their data could be accessed by cloud providers who own both hardware and software stack. In fact, there have been many privacy incidents in cloud computing systems such as Google Docs' inadvertent sharing users documents to other users [3]. There has been numerous efforts by both the research community and the industry to address the security concern. The first solution is to use hybrid cloud computing [4] and split applications' computation into computation on sensitive data and computation on non-sensitive data. The computation on non-sensitive data is then outsourced to public clouds while the computation

on sensitive data stays on private clouds. Amazon Web Service (AWS) offers hardware security modules (HSMs) [5] where users' code and data are protected by tamper-proof hardware. There are also techniques that enable applications to perform query processing on encrypted data [6, 7]. Recent years have seen hardware-based implementations of isolated execution that protects specific code from the rest of the system, including operating system, hypervisor and firmware. For example, Intel SGX [8–10] and AMD Memory Encryption [11] are the two most popular commodity processors that enable creating an isolated execution environment for clients' code. However, the introduction of SGX consequently limits the capability of cloud providers on contemporary cloud platforms. A benign cloud provider now can no longer inspect client code and data for security compliance. Cloud providers often wish to ensure that their users follow regulatory compliance including security compliance. For example, AWS [12] requires its users to not host malicious software on their platforms. This requirement aims to prevent such scenario as the Zeus botnet [13] which was created by using Amazon's Elastic Cloud Computing (EC2). Similarly, Google's AppEngine [14], a cloud platform for developing and hosting web applications in Google-managed data centers, also has a similar security policy.

As cloud services are increasingly utilized by end users, they have become the target of malware attacks in general and ransomware attacks in particular. For example, users of Microsoft's cloud-based Office 365 were attacked by the Cerber ransomware [15] that encrypts a variety of file types. Recent ransomware detection methods [16–18] often focus on monitoring filesystem activities to track the real-time changes of user data such as the entropy of files to detect ransomware. Because these methods rely on file encryption as the signature behavior of ransomware, they are prone to false positives. For example, CryptoDrop [16] flags legitimate programs such as GPG or 7-zip as ransomware.

In today's cloud platforms, clients can secure their applications by either relying on security services provided by cloud providers or customizing and configuring the software stack with standard security tools. For example, cloud providers can deploy intrusion detection systems (IDS) [19] that help filter malicious traffic to clients' machines. Clients' code and data could be protected against malicious code by relying on trusted components such as a hypervisor [20–24]. In commercial cloud platforms such as Amazon EC2, clients can install firewalls on

their virtual machines (VMs). However, in this cloud computing model, clients do not have much power and flexibility to deploy security services. The large majority of clients may lack the resources and expertise to set up and configure security tools.

This dissertation proposes three solutions to address the above security and privacy problems of cloud-based applications: 1) a mechanism allowing cloud providers to verify clients' code within a hardware-based protected execution environment, 2) effectively detecting ransomware in the cloud, and 3) enriching security related services for clients in the cloud.

1.2 Inspecting Application Code Within a Hardware-based Protected Execution Environment

The introduction of Intel SGX [8–10] has demonstrated a widespread adoption of using trusted hardware in building secure systems [25, 26]. Intel SGX is a set of x86-64 ISA extensions which provide any application with the ability to set up protected execution environments called enclaves which contain application code and data and are guarded against unauthorized access by privileged software and hardware attacks. SGX helps reduce the trusted computing base (TCB), which includes only the code users place inside enclaves and the processor.

Enclaves reside in a protected physical memory region called the enclave page cache (EPC) and are protected by processor access controls. In particular, the processor prevents code running outside an enclave from reading or writing to the EPC pages belonging to that enclave. Also, the cache lines belonging to enclave memory are encrypted by an on-chip memory encryption engine(MEE) before being written to DRAM. SGX also features measurement and remote attestation primitives that allow a remote system to verify cryptographically that an enclave has been created properly. During an enclave creation, the processor computes a digest of the enclave including enclave content and attributes. Any attempt by untrusted software such as the OS to interfere with this process will result in a different digest for the enclave. To safely store enclave content onto untrusted persistent storage, SGX offers hardware-generated sealing primitives.

However, the introduction of SGX consequently limits the capability of cloud providers on contemporary cloud platforms. A benign cloud provider now can no longer inspect client code

and data for service-level agreement (SLA) compliance. Cloud platforms such as Google Cloud Platform [27] often specify customer responsibility to adhere to security requirements such as regularly updating anti-virus software. In fact, some researchers have raised concerns about malicious clients using SGX to conceal malware [28–30]. On a non-SGX cloud platform where cloud providers can access client code and data, benign clients can also benefit from services such as malware detection, vulnerability scanning, and memory deduplication.

A cloud provider who wishes to inspect the client's code within an SGX enclave has only a few approaches. The *first* approach is to indirectly infer the presence of malicious activities by monitoring the system call interface exposed to the application. For example, minibox [31] verifies that an application behaves properly by checking system call parameters of the application for malicious activities such as accessing sensitive files that do not belong to the application. However, this approach is limited to system call related activities and therefore is not applicable to verifying the enclave's code for a broad range of regulatory compliance. The *second* solution is to rely on a trusted-third party (TTP) mutually trusted by both the cloud provider and client. Both the cloud provider and client would agree upon a certain set of compliance encoded in SLAs that the enclave code must satisfy. For example, the cloud provider could specify that the enclave code must be certified as clean by a certain anti-malware tool. Given the compliance and the sensitive content disclosed by the client, the TTP then checks the content for policy compliance. Only if the content passes the check, will it be provisioned to the enclave. However, the main drawback of this solution is the need for a TTP. TTPs could themselves be subject to government subpoenas that force them to hand over the client's sensitive content. From the client's perspective, this solution provides no more security than handing over sensitive content to the cloud provider.

This dissertation presents EnGarde, an enclave inspection library that achieves the above goal without TTPs. Cloud providers and clients agree upon the policies that enclave code must satisfy and encode it in EnGarde. Thus, cloud providers and clients mutually trust EnGarde with policy enforcement. The cloud provider creates a fresh enclave provisioned with EnGarde, and proves to the client, using SGX's hardware attestation, that the enclave was created securely. The client then hands its sensitive content to EnGarde over an encrypted channel. It provisions the enclave with the client's content only if the content is policy-compliant. If not, it informs

the cloud provider, who can prevent the client from creating the enclave.

EnGarde's approach combines the security benefits of non-SGX and SGX platforms. From the cloud provider's perspective, it is able to check client computations for policy-compliance, as in non-SGX platforms. From the client's perspective, its sensitive content is not revealed to the cloud provider, preserving the security guarantee as offered by SGX platforms. Moreover, EnGarde statically inspects the client's enclave content only once—when the enclave is first provisioned with that content. One can also imagine an extension of EnGarde that instruments client code to enforce policies at runtime, but our current implementation only implements support for static code inspection. Thus, except for a small increase in enclave-provisioning time, EnGarde does not impose any runtime performance penalty on the client's enclave computations. We have implemented a prototype of EnGarde and have used it to check a variety of security policies on a number of popular open-source programs running within enclaves.

1.3 Ransomware Detection in The Cloud

Ransomware has become a major security threat to individuals and businesses and has caused millions of dollars in damages [32, 33]. For example, the WannaCry ransomware and its variants affected hundreds of thousands individuals and organizations in 150 countries, causing a disruption of critical services [34–37]. Ransomware functions by encrypting a victim's files and asking for a ransom to release the decryption key. The ransom payment is often transferred to the attackers using an anonymous payment mechanism such as Bitcoin. As cloud services are increasingly utilized by end users, they have become the target of ransomware attacks. For example, the Cerber ransomware [15] targeted users of Microsoft's cloud-based Office 365. This ransomware is able to encrypt 442 file types and perform many harmful activities such as modifying Internet Explorer (IE) zone settings, deleting shadow copies and disabling Windows Startup Repair.

State-of-the-art ransomware detection techniques rely on the filesystem layer to track the ransomware activity. For example, CryptoDrop [16] detects ransomware by monitoring real-time changes in user data. It uses three indicators to detect the existence of ransomware in the system: the entropy of the content of user files, file type changes where the type of a file

is often the same before and after it is written, and the similarity between two versions of the same file where the file before being infected by ransomware is often completely dissimilar to ransomware-encrypted content. UnVeil [17] is another work that uses the entropy of the I/O data buffer to detect ransomware, where a significant increase in the entropy between read and write data buffers at a given file offset is a sign of ransomware running on the system. ShieldFS [18] is a detection system that focuses on building detection models to distinguish ransomware from benign processes. The models are based on calculating the entropy of write operations, frequency of read and write, folder-listing operations, fractions of files renamed, and the file-type usage statistics. ShieldFS also scans the memory of any suspected processes to search for the typical block cipher key schedules.

While the above detection systems are effective at detecting ransomware, they unfortunately are prone to false positives. In fact, the authors of CryptoDrop acknowledge that legitimate programs such as GPG or 7-zip will be flagged as ransomware by their system. The reason is that these techniques often rely on file encryption as the signature behavior of ransomware [16, 17] regardless whether the encryption is legitimate or malicious. Specifically, these techniques monitor data changes and I/O request sequences to detect encryption operations performed on the data. In this work, we take another approach that not only efficiently detects ransomware-like behaviors but also reliably distinguishes whether the behaviors are exposed by ransomware or benign programs that perform encryption and compression. Our experimental results show that although both ransomware and encryption programs such as 7-zip encrypt users' files, they expose substantially different low-level event measurements.

This dissertation proposes a solution that addresses the above limitations by combining HPCs and machine learning techniques. Specifically, we propose HRD [1], a system that uses HPCs to gather measurements of all hardware events in the entire system for different runs of benign workloads and ransomware samples. After having the event traces, machine learning is applied to select the most discriminating events that clearly distinguish ransomware from benign workloads. HRD is then used to measure the previously selected events for a large number of ransomware samples and benign programs from a training set.

[1] Hardware-based ransomware detector.

It has demonstrated that the system is able to detect unseen ransomware including variants of the WannaCry ransomware with high accuracy and is also able to distinguish benign programs that perform encryption and compression such as 7-zip and IE with 100% precision.

Unlike previous approaches that use HPCs to detect malware, HRD does not require changing the existing processor architectures. As will be shown in the next sections, commodity processor architectures are sufficient to detect ransomware. The evaluation showed that HRD consumes small amounts of memory and CPU time and incurs no additional overhead to filesystem operations because it does not interfere with the filesystem.

1.4 Enriching Security Related Services for Clients in the Cloud

In Infrastructure-as-a-Service (IaaS) cloud platforms such as Amazon EC2 and Windows Azure, clients can implement security measures either by relying on IDS deployed by providers [19] or customize and configure the software stack of VMs with standard security tools such as network intrusion detection systems (NIDS) or firewalls. However, a large majority of clients often wish to install the same standard security tools and may lack the resources and expertise to set up and configure VMs from scratch.

Currently some cloud providers provide clients with customized services via cloud markets [38] where clients can choose VMs pre-installed with software stacks that address their needs. However, these services are often standard applications such as a web server or database server and are restricted to run within a VM. Motivated by these cloud markets, this dissertation envisions a cloud app market where cloud apps, implemented as VMs, offer standard utilities such as firewalls, NIDS, storage encryption, and VMI-based security tools. Cloud apps can also implement a host of other non-security-related utilities, such as memory and disk deduplication, and network middleboxes such as packet shapers and QoS tools. Clients can leverage these utilities by simply downloading the appropriate cloud apps, and *linking them suitably with their work VMs*. The key challenge in realizing this vision on current cloud computing environments is that such interaction between VMs is disallowed. Each virtualized platform has one privileged VM (also called the *management VM*), controlled by the cloud provider, that supervises the execution of client VMs. The management VM oversees all I/O from client

VMs, and completely isolates VMs from each other. While such isolation is desirable across VMs of *different clients*, it also prevents VMs belonging to the same client from interacting in useful ways.

This dissertation presents a taxonomy of cloud apps, ranging from standalone apps to ones that involve complex system- and network-level interactions with other VMs. It uses this taxonomy to motivate the key requirements of an ecosystem that supports such cloud apps.

It describes an end-to-end overview of the various components of the ecosystem from a client's perspective. In particular, it develops the notion of *cloud app permissions* to allow clients to reason about and control the behavior of third-party apps that they may download from a cloud app market. It also presents techniques for a client to compose the functionality of multiple cloud apps within a single app.

It explores the design options, such as hypervisor modifications, nested virtualization and network-level support, to implement various classes of cloud apps, such as those that offer system-level introspection, those that act as network middleboxes and those that offer storage-level services. It also explores the benefits and tradeoffs of each design option.

It presents an implementation of our design atop the KVM hypervisor, and quantify the performance of various design options.

Finally, it demonstrates the utility of our model by building and evaluating a number of security-related cloud apps, and showing that clients can use these apps to realize the vision of security-as-a-service.

1.5 Summary of Contributions

The thesis that this dissertation support is the following:

It is possible to enhance the security and privacy of cloud-based applications by using recent hardware advances and by rethinking the security model in today's cloud platforms

This dissertation supports the above thesis statement and makes the following contributions:

- We have proposed novel approaches to support different security and privacy requirements of cloud applications. EnGarde (Chapter 2) is an SGX enclave inspection library that allows the provider of an SGX-based cloud platform to examine the client's code for security compliance while ensures that the client's sensitive content is not revealed to the cloud provider. HRD (Chapter 3) is a system that leverages HPCs and machine learning to effectively detect ransomware in the cloud. We explore the benefits and feasibility of a new cloud computing model (Chapter 4) where clients can flexibly deploy security services by simply downloading apps (implemented as VMs) and linking them with their work VMs.

- We have designed and implemented EnGarde, an enclave inspection library that preserves the security and privacy benefits offered by Intel SGX and allows the cloud provider to verify the client's SGX-based enclave against predefined policies mutually agreed by the cloud provider and the client. EnGarde achieves its goal by using SGX's hardware attestation and having an encrypted channel set up between the cloud provider and the client. EnGarde only allows the client code to execute if it follows mutually agreed security policies. We have evaluated the effectiveness of EnGarde by using it to enforce three popular security policies for numerous real world applications.

- We have built HRD, a system that can detect ransomware in cloud-based environments with low false positives. Unlike recent ransomware detection systems that rely on tracking the filesystem layer, HRD uses HPCs to collect measurements of low-level events of benign programs and ransomware samples independently. We then leverage machine learning to select the most discriminating events that clearly distinguish ransomware from benign programs. We have evaluated the accuracy of various machine learning models on event traces gathered from a training set consisting of a large number of ransomware samples and benign programs. We have demonstrated that HRD is able to detect unseen ransomware including variants of the WannaCry ransomware with high accuracy. The evaluation have shown that HRD is able to distinguish benign programs that perform encryption such as 7-zip, IE, and AESCrypt from ransomware.

- We have demonstrated the utility of a new cloud computing model where the client can

make use of cloud apps, implemented as VMs, to implement security measures. We have explored various design options to realize this cloud computing model and have implemented our design atop the KVM hypervisor, and evaluated the performance overhead of each design option. We have built and evaluated a number of security-related cloud apps and have shown that clients can use these apps to realize the vision of security-as-a-service.

Chapter 2

Enforcing Security Compliance of Cloud-based Applications Within a Hardware-based Protected Execution Environment

Intel's SGX architecture allows cloud clients to create enclaves, whose contents are cryptographically protected by the hardware even from the cloud provider. While this feature protects the confidentiality and integrity of the client's enclave content, it also means that enclave content is completely opaque to the cloud provider. Thus, the cloud provider is unable to enforce policy compliance on enclaves.

In this chapter, we introduce EnGarde, a system that allows cloud providers to ensure SLA compliance on enclave content. In EnGarde, cloud providers and clients mutually agree upon a set of policies that the client's enclave content must satisfy. EnGarde executes when the client provisions the enclave, ensuring that only policy-compliant content is loaded into the enclave. EnGarde is able to achieve its goals without compromising the security guarantees offered by the SGX, and imposes no runtime overhead on the execution of enclave code. We have demonstrate the utility of EnGarde by using it to enforce a variety of security policies on enclave content.

2.1 Introduction

In recent years, the research community has devoted much attention to security and privacy threats that arise in public cloud computing platforms, such as Amazon EC2 and Microsoft Azure. From the perspective of cloud clients, one of the chief security concerns is that the computing infrastructure is not under the client's control. While relinquishing control frees the client from having to procure and manage computing infrastructure, it also exposes the client's code and data to cloud providers and administrators. Malicious cloud administrators can compromise client confidentiality by reading sensitive code and data directly from memory images

of the client's virtual machines (VM). They could also inject malicious code into client VMs, *e.g.,* to insert backdoors or log keystrokes, thereby compromising integrity. Even otherwise honest cloud providers could be forced to violate client trust because of subpoenas.

Intel's SGX architecture [8–10, 39, 40] offers hardware support to alleviate such client security and privacy concerns. SGX allows client processes and VMs to create *enclaves*, within which they can store and compute on sensitive data. Enclaves are encrypted at the hardware level using hardware-managed keys. SGX guarantees that enclave content that includes enclave code and data is not visible in the clear outside the enclave, even to the most privileged software layer running on the system, *i.e.,* the operating system (OS) or the hypervisor. SGX also offers support for enclave attestation, thereby providing assurances rooted in hardware that an enclave was created and bootstrapped securely, without interference from the cloud provider. With SGX, clients can therefore protect the confidentiality and integrity of their code and data even from a malicious cloud provider or administrator, so long as they are willing to trust the hardware.

Despite these benefits, SGX has the unfortunate consequence of flipping the trust model that is prevalent on contemporary cloud platforms. On non-SGX platforms, a benign cloud provider benefits from the ability to inspect client code and data. The cloud provider can provide clients with services such as malware detection, vulnerability scanning, and memory deduplication. Such services are also beneficial to benign clients. The cloud provider can check client VMs for service-level agreement (SLA) compliance, thereby catching malicious clients who may misuse the cloud platform in various ways, *e.g.,* by using it to host a botnet command and control server. In contrast, on an SGX platform, the cloud provider can no longer inspect the content of a client's enclaves. This affects benign clients, who can no longer avail of cloud-based services for their enclaves. It also benefits malicious clients by giving them free reign to perform a variety of SLA-violating activities within enclaves. Researchers have discussed the possibility of such "detection-proof" SGX malware [28–30]. Without the ability to inspect the client's code, the cloud provider is left to using other, indirect means to infer the presence of such malicious activities. For example, minibox [31] verifies that an application behaves properly by checking system call parameters of the application for malicious activities such as accessing sensitive files that do not belong to the application.

Can a benign client benefit from the security offered by the SGX while still allowing the cloud provider to exert some control over the content of the client's enclaves? One strawman solution to achieve this goal is to use a trusted-third party (TTP). Both the cloud provider and client would agree upon a certain set of policies/constraints that the enclave content must satisfy (as is done in SLAs). For example, the cloud provider could specify that the enclave code must be certified as clean by a certain anti-malware tool, or that the enclave code be produced by a compiler that inserts security checks, *e.g.,* to enforce control-flow integrity or check for other memory access violations. They inform the TTP about these policies, following which the client discloses its sensitive content to the TTP, which checks for policy compliance. The cloud provider then allows the client to provision the enclave with this content.

However, the main drawback of this strawman solution is the need for a TTP. Finding such a TTP that is acceptable to both the cloud provider and the client is challenging in real-world settings, thereby hampering deployability. TTPs could themselves be subject to government subpoenas that force them to hand over the client's sensitive content. From the client's perspective, this solution provides no more security than handing over sensitive content to the cloud provider.

Contributions. We present EnGarde, an enclave inspection library that achieves the above goal without TTPs. Cloud providers and clients agree upon the policies that enclave code must satisfy and encode it in EnGarde. Thus, cloud providers and clients mutually trust EnGarde with policy enforcement. The cloud provider creates a fresh enclave provisioned with EnGarde, and proves to the client, using SGX's hardware attestation, that the enclave was created securely. The client then hands its sensitive content to EnGarde over an encrypted channel. It provisions the enclave with the client's content only if the content is policy-compliant. If not, it informs the cloud provider, who can prevent the client from creating the enclave.

EnGarde's approach combines the security benefits of non-SGX and SGX platforms. From the cloud provider's perspective, it is able to check client computations for policy-compliance, as in non-SGX platforms. From the client's perspective, its sensitive content is not revealed to the cloud provider, preserving the security guarantee as offered by SGX platforms. Moreover, EnGarde statically inspects the client's enclave content only once—when the enclave is first

provisioned with that content. One can also imagine an extension of EnGarde that instruments client code to enforce policies at runtime, but our current implementation only implements support for static code inspection. Thus, except for a small increase in enclave-provisioning time, EnGarde does not impose any runtime performance penalty on the client's enclave computations. We have implemented a prototype of EnGarde and have used it to check a variety of security policies on a number of popular open-source programs running within enclaves.

2.2 SGX Background

Enclaves. The main feature of SGX is its support for enclaves. An enclave is a linear span of a process's virtual address space whose physical pages are drawn from a region of physical memory called the encrypted page cache (EPC). The contents of EPC pages are protected cryptographically by the hardware, which does not reveal the encryption key even to the most privileged software layer on the system (*e.g.,* the OS or the hypervisor). A process can have multiple enclaves in its address space.

A process enters an enclave via an instruction (`EENTER`). Within an enclave, the process can have multiple threads of execution. Each such thread can freely access the memory contents of both the enclave as well as the rest of the process address space. If an enclave thread references an address within the enclave, the hardware fetches corresponding memory page from the EPC and decrypts it within the hardware cache hierarchy, thereby offering the process a view of the plaintext content of the page. An adversary outside the enclave (*e.g.,* observing the memory bus) will only see encrypted traffic to the EPC page, thereby preserving the confidentiality and integrity of the EPC page. SGX imposes a few restrictions on the code that can execute within an enclave. An enclave can only execute user-mode code and cannot invoke any OS services, *e.g.,* via system calls. If the enclave code needs to avail of such services, it must save the enclave state, exit the enclave (via an instruction called `EEXIT`), and have the non-enclave code of the process access such services on its behalf. SGX offers various data structures to save enclave state in an encrypted fashion, thereby protecting it from adversaries outside the enclave. SGX hardware ensures that code executing outside the enclave, whether in user-mode context in the process address space or in kernel-mode context within the OS (or hypervisor), cannot access the plaintext enclave content.

Although an OS (or hypervisor) cannot view the plaintext contents of a process's enclaves, it is still responsible for various aspects of enclave management. The OS creates enclaves for processes (using `ECREATE`), adds or removes pages from a process's enclaves (using `EADD` and `EREMOVE`, respectively), and manages the process's page tables. Page table entries corresponding to the virtual address range of an enclave will be mapped to the EPC. Although we have only introduced a handful of instructions, the SGX supports a total of 24 new enclave management instructions [39, 40].

Attesting and Provisioning Enclaves. When an enclave is newly created within a process's address space, it is initialized with some generic bootstrap code. The exact nature of this bootstrap code differs based on the software vendor who offers this code. However, at the very minimum, this bootstrap code implements basic cryptographic functionality (*e.g.,* for SSL/TLS), wrappers for system calls and other popular libraries that the client's enclave code may wish to use. SGX offers support for *attestation* [8], which allows remote clients of an SGX-based cloud platform to ensure that enclaves are initialized securely.

Remote attestation on SGX platforms follows a standard challenge/response scheme as in TPM-based attestation protocols [41]. The client sends a challenge to the SGX-based machine on the cloud platform. Each SGX-based machine is endowed with a dedicated, Intel-provided *quoting enclave*. The quoting enclave obtains a measurement (a SHA-256 digest of a log of all activities during enclave initialization [8], obtained via the `EREPORT` instruction) of each newly-created enclave, and signs the measurement using a device-specific private key, called the Intel EPID key. The SGX hardware ensures that only the quoting enclave has access to the EPID key. The client can then verify the signed measurements, thereby obtaining a guarantee, rooted in SGX hardware, that the enclave was initialized correctly.

Following attestation, the client *provisions* the enclave with sensitive content. Thus, the client needs an encrypted, authenticated channel between its server and the newly-created enclave on the cloud platform. On SGX systems, this problem is addressed by generating an ephemeral public/private key pair during enclave creation and initialization. The value of this ephemeral public key is included in the attestation quote that is signed by the quoting enclave,

thereby providing the client a hardware-rooted guarantee that binds the public-key to the enclave. The client can then use this public-key to bootstrap an SSL/TLS handshake, thereby establishing a secure channel to the enclave. The client then transmits all sensitive content to the enclave over this encrypted channel.

2.3 Design of EnGarde

Problem Definition. Given the features of the SGX, a client's enclave is opaque to the cloud provider. This benefits clients because it protects the confidentiality and integrity of their sensitive content. However, the cloud provider can no longer inspect or enforce any policies on enclave content.

In this work, we remedy the situation by introducing EnGarde, which statically checks the policy compliance of the code that the client proposes to execute in its enclaves. The client and cloud provider agree upon a set of policies that the client's code must satisfy. For instance, the cloud provider may require the client to instrument its code with certain security checks or link its code against certain versions of libraries. EnGarde's architecture supports plugging in *policy modules*, which check compliance based upon the policies that the cloud provider and client mutually agree upon. EnGarde executes during enclave provisioning, and checks that the client's enclave code is policy-compliant. If the client's code is not policy compliant, EnGarde informs the cloud provider, who can prevent code from executing.

Threat Model. We assume that the cloud provider and client are mutually distrusting. Before allowing the client to create and provision enclaves, the cloud provider and client mutually agree upon the set of policies that the client's code must satisfy. We assume that the code of EnGarde and its policy modules is available to both the cloud provider and client for inspection.

From the cloud provider's perspective, the client will attempt to violate the mutually agreed-upon policies. It therefore verifies that EnGarde and its policy modules indeed enforce these policies. From the client's perspective, the cloud provider will attempt to learn the contents of the enclave. Thus, the client verifies that EnGarde and its policy modules leak no additional information about its code to the cloud provider, *i.e.,* the only explicit communication between

EnGarde and the cloud provider must be to inform the cloud provider about policy compliance and to identify the virtual addresses of the pages that contain the client's code. For this work, we do not consider implicit and covert communication channels via which EnGarde can communicate information about the client's code to the cloud provider; techniques to analyze EnGarde's code for such covert channels can be the subject of future research. The client can also use EnGarde to independently verify policy compliance of the enclave code that it wants to provision.

Both the cloud provider and the client trust the SGX hardware platform. EnGarde and its policy modules are loaded into a freshly-created enclave (as part of the bootstrap code). Both the provider and the client use SGX's attestation features to ensure that EnGarde was correctly loaded into the enclave. EnGarde receives the client code over a SSL/TLS channel, checks policy compliance, and informs the cloud provider. Any attempts by the cloud provider to cheat, *e.g.,* by falsely claiming that the code is not policy-compliant or failing to allow policy-compliant code to execute, can easily be detected by the client.

Overall Design. EnGarde primarily consists of in-enclave components that are loaded when an enclave is created (see Figure 2.1). As is standard on all SGX systems, the client first ensures (using SGX's attestation protocols [8]) that the enclave was initialized securely.

Following this step, the client sets up an end-to-end encrypted channel with the enclave. To do this, the bootstrap code loaded into a freshly-created enclave first generates a 2048-bit RSA key pair and then establishes a socket connection to the client machine. As a next step, the enclave sends its newly-generated public key to the client machine so that it can encrypt its 256-bit AES key and sends the encrypted AES key back to the loader. This key is used to establish an end-to-end encrypted channel with the client.

EnGarde checks the client's enclave contents for policy compliance after the client sends it the contents, but before the content is provisioned within the enclave for execution. The client sends the content in encrypted blocks, which EnGarde's crypto library decrypts to form an in-memory executable representation.

EnGarde operates at the granularity of memory pages, and therefore splits the content into page-level chunks. We assume that the client sends x86 binary code and identifies pages which

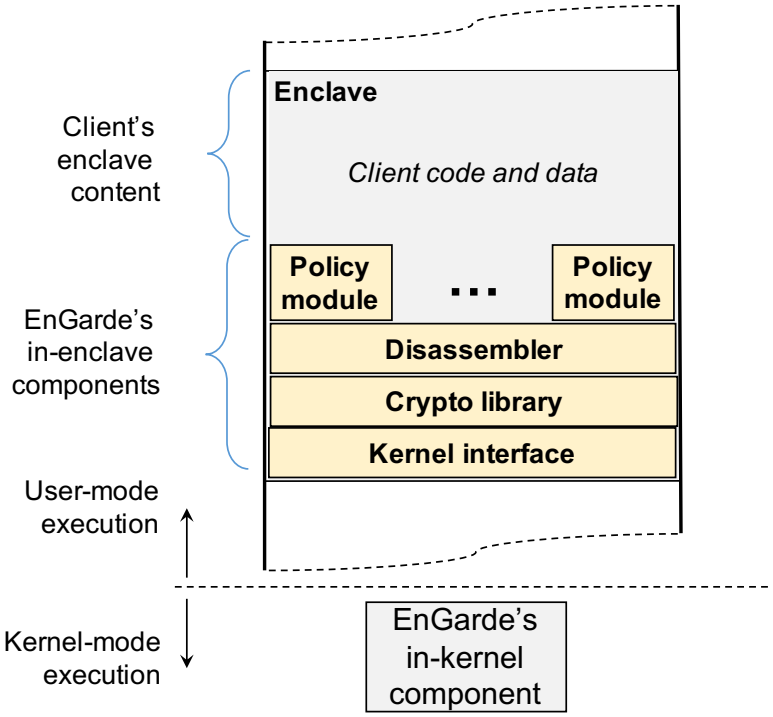


Figure 2.1: Design of EnGarde.

contain code. The remaining pages are assumed to contain data. EnGarde rejects pages that contain mixed code and data. We assume that clients suitably compile their code so as to satisfy this assumption. We also assume that the enclave code is not obfuscated to hinder analysis.

Once EnGarde has received all the code, it proceeds to disassemble the client's code. To do this, EnGarde relies on the disassembler provided by Google's Native Client (NaCl) [42]. NaCl makes a number of assumptions to ensure clean, unambiguous disassembly. For example, it requires no instructions to overlap a 32-byte boundary, that all control-transfers target valid instructions, and that all valid instructions are reachable from the start address. EnGarde requires the client's enclave to satisfy the same constraints.

After disassembling the code, EnGarde checks the code for policy compliance. Recall that the specific policies that EnGarde checks depend on those negotiated with the client. In general, the policies can check structural properties of the code, *e.g.,* that certain instrumentation has been added to the code. EnGarde checks policies using pluggable *policy modules*. Each policy module checks compliance for a specific property, and specific policy modules that are loaded during enclave creation depend upon the policies that the client and cloud provider have agreed

upon. In Section 4.6, we discuss three examples of policy modules that we have implemented in our current EnGarde prototype. While EnGarde's disassembler works even on stripped binary code (*i.e.,* code without symbol-table information), specific policy modules may require symbol table information to check compliance. If EnGarde's policy modules require such information, then it requires the client to produce code using symbol tables.

The policy modules determine whether the client's code is policy compliant. If not, the code is rejected, and the enclave is not provisioned. If EnGarde determines that the code is policy-compliant, it then informs the host. EnGarde also contains a host-level component, either running within the host's OS kernel or the hypervisor (if the host is virtualized). EnGarde's in-enclave components provide the in-kernel component with a list of executable code pages. The underlying OS component marks these pages as executable, but not writable. The remaining pages are given write permissions, but are not given execute permissions. The host OS component of EnGarde also prevents the enclave from being extended after it has been provisioned. This ensures that the client cannot inject any further code into the enclave after it has been checked for compliance. EnGarde's in-kernel component enforces execute and write permissions by setting page-table permission bits in the underlying host OS. While the current version of SGX hardware allows for page permissions to be set/cleared by the host OS, it does not yet offer support for page permissions at the hardware level (*i.e.,* page permissions for EPC pages). This feature has been proposed in version 2 of the SGX instruction set [43]. Although EnGarde can be implemented readily even on SGX version 1 processors, the permission check can only be enforced in software within the host OS, and this has been shown to be open to attack [44]. Thus, EnGarde requires the features of SGX version 2 for security.

Following this, the enclave can be accessed and executed as on traditional SGX platforms. Note that EnGarde only operates during enclave provisioning. Thus, EnGarde only imposes a performance penalty during enclave provisioning. Enclave execution incurs no additional runtime overhead.

2.4 Implementation

Features of the SGX are now commercially deployed in Intel's Skylake series of processors. Despite the availability of commodity hardware, for this work, we chose to develop EnGarde

atop OpenSGX [45], a QEMU-based SGX emulation infrastructure. Two factors governed our choice.

First, open-source software support for SGX enclave development is still rudimentary. To create a system that fully realizes the power of enclaves, we need support for in-enclave bootstrap code and supporting libraries, drivers within the OS, and compiler support to emit SGX code. Although Intel provides SDKs for Windows 10, these SDKs are closed-source, which complicates extension and modification [46]. An open-source Linux SDK [47], which we could have extended, was released only in June 2016 when we were already underway with our EnGarde prototype. While Intel's programming references [39, 40] specify the semantics of instructions, they offer considerable freedom to end-developers to choose their enclave programming model. Community consensus has yet to emerge on these programming models, and rather than define one of our own, we chose to use the programming model defined by OpenSGX. Moreover, OpenSGX incorporates driver support for SGX, and has ported various utilities and libraries to work in enclave mode, which we could readily utilize and extend for EnGarde.

Second, even the SGX architecture itself is evolving. Skylake processors currently implement version 1 of the instruction set. This instruction set poses a number of restrictions [48, 49], the chief of which is that it does not permit page protections to be changed at the hardware level for pages in the EPC. Page protections can still be changed at the level of page tables, and SGX performs a two-level page protection check prior to writing or modifying a page: at the page-table level and at the hardware level. However, recent work has shown that lack of support for page protection modifications at the EPC level can be exploited [44]. As already discussed, EnGarde relies on the ability to change EPC page protections. In addition, SGX hardware currently requires all enclave memory to be committed at enclave build time (therefore requiring the developer to predict and use the maximum stack/heap sizes during enclave build) and does not allow additional code modules to be dynamically loaded into the enclave after it has been built. While these changes have been proposed in version 2 of the instruction set [48, 49], it is not yet commercially available [43]. In contrast, it is easy to explore such changes within the context of a software-based SGX emulator such as OpenSGX.

Our EnGarde prototype supports x86-64 executables that use ELF format [50, 51], are

compiled as position independent executables and are statically linked. In this section, we first describe our modifications to OpenSGX. We then discuss the components of EnGarde.

Modifications to OpenSGX. The client enclave holds the client executable as well as its decoded instructions. As a result, the number of EPC pages should be large enough to meet the memory requirements of the client enclave. OpenSGX restricts the number of EPC pages to 2000. We modified OpenSGX to increase the default number of EPC pages to 32000 which translates to 128 MB for the physical protected memory region. On OpenSGX, this size can be extended to meet further memory requirements. We also change the number of initial page frames for the heap region from the default value of 300 to 5000.

Binary Disassembly. The executable file provided by the client is in 64-bit ELF format. An ELF binary comprises of several segments and each segment has one or more sections. Each section contains information of similar type, for instance the `.text` section contains the executable code, all writable data is stored in the `.data` section and uninitilized data is kept in the `.bss` section. The ELF format also features an ELF header located at the beginning of the file and is used to recognize other parts of the file.

One common challenge in disassembling a binary is mixing of code and data within the code section. Our EnGarde prototype assumes that the client's executable is compiled with separated code and data sections. Before disassembling the code sections of the executable, the loader checks its header to verify that the executable is correctly formatted. The checks include checking the signature as well as the ELF class of the executable. The loader next reads the program header of the executable to extract all text sections. We implement the disassembler based on the 64-bit binary disassembler of NaCl, an open source sandbox for native code. Using prefix and opcode tables for x86-64 bit instruction set, the disassembler parses the byte sequence of the text sections into instructions and associated metadata information, *e.g.,* the number of prefix bytes, number of opcode bytes and number of displacement bytes [52].

NaCl's disassembler does not track all disassembled instructions. Instead, during the disassembly it uses a buffer that stores the most recently disassembled instructions. This stems from the fact that the NaCl validates each instruction right after it is disassembled. We instead use a dynamically allocated buffer that can hold all the instructions and use that buffer as the

input to policy checks. Since dynamic memory allocation involves exiting the enclave mode and invoking a trampoline, we reduce the involved overhead by restricting the calls to `malloc` by allocating a memory page at a time instead of just a memory region for an instruction.

Along with disassembling the executable, the loader also reads the symbol tables to keep track of the address and name of all the functions in the executable. It constructs a symbol hash table whose key is the address of a function and value is the name of the function. This symbol hash table could be used by the policy checking component when it perform policy checks.

Loading. After the executable has been checked and confirmed to follow certain policies the loader takes over. The loader maps the `text`, `data` and `bss` segments to the enclave memory, making the `text` segment be executable but read-only, the `data` segment and `bss` segment be writable but non-executable. It then locates the sections that require relocations and the locations at which the relocations should be applied. The loader acquires all the information that it needs for relocations from the `.dynamic` section of the executable. In particular, the loader determines the address and the size of relocation tables which contain detailed information for relocations by reading appropriated entries of the `.dynamic` section. Upon completing relocation, the loader sets up a call stack and transfers control to the executable.

2.5 Evaluation

In evaluating EnGarde, our main goals were to demonstrate the flexibility of EnGarde by showing that it can check compliance against a variety of policies, and understand the performance costs of various components of EnGarde.

Our setup consisted of running OpenSGX atop of Ubuntu 14.04 on a physical machine equipped with an Intel Core i5 CPU and 16GB of memory. We use clang and llvm version-3.6 to compile and instrument many real world applications to run within enclaves: Nginx (an HTTP server), Memcached (a popular key-value store), Netperf (a networking benchmark), otp-gen (a password generator), graph-500 (a graph data benchmark) and two SPEC benchmarks (401.bzip2 and 429.mcf). In all experiments, all the applications are compiled as position independent executables and are statically linked. To keep the size of the executables small all applications are linked against musl-libc [53] instead of GNU libc [54]. Figure 2.2

Components	LOC
Code Provisioning	270
Loading and Relocating	188
Checking Executables linked against musl-libc	1,949
Checking Executables Compiled with Stack Protection	109
Checking Executables Containing Indirect Function-Call Checks	129
Client's side program	349
Musl-libc	90,728
Lib crypto (openssl)	287,985
Lib ssl (openssl)	63,566
Total	453,349

Figure 2.2: Sizes of various components of EnGarde. Some of these components (*e.g.,* the cryptographic libraries) are part of the default loader in all enclave implementations.

shows the lines of code of all the components of EnGarde's implementation. In the following sections, we describe the performance costs of three policy modules that we implemented in EnGarde. For each benchmark, we assume that the benchmark executes within the enclave, and we evaluate the cost of EnGarde as it loads the benchmark within the enclave for execution, and checks for policy compliance.

Compliance for Library Linking. When a cloud provider allows a client to run code on its platform, it often expects the client to run a particular version of the code. For example, the cloud provider may require that the client execute a version of the code that has been patched with the latest security updates. As a special case of this, the cloud provider may wish to check whether the client's code has been linked against specific versions of certain libraries. For example, the cloud provider may wish to ensure that if the client's code uses OpenSSL, then the version of OpenSSL that is used is free of the vulnerability that caused the HeartBleed exploit. As another example, consider `/CONFIDENTIAL` [55], a library that ensures that enclave code satisfies certain information-flow confinement properties, *i.e.,* enclave code that is linked against this library will not accidentally leak sensitive information. To prevent liability issues arising from any accidental data leaks in the client's code, the cloud provider may wish to ensure that the client's code is linked with the `/CONFIDENTIAL` library.

To illustrate the power of EnGarde at enforcing such library-linking policies, we implemented a policy module that verifies whether an executable is linked against musl-libc [53]

Benchmark	#Inst.	Disassembly	Policy Checking	Loading and Relocation
Nginx	262,228	694,405,019	1,307,411,662	128,696
401.bzip2	24,112	34,071,240	148,922,245	4,239
Graph-500	100,411	140,307,017	246,669,796	4,582
429.mcf	12,903	18,242,127	123,895,553	4,363
Memcached	71,437	137,372,517	489,914,732	8,115
Netperf	51,403	90,616,563	367,356,878	18,090
Otp-gen	28,125	42,823,024	198,587,525	5,388

Figure 2.3: Performance of EnGarde to check the *Library-linking* policy. Here EnGarde checks whether each benchmark has been linked against musl-libc. The figure shows the size of each benchmark, measured as the number of instructions in the code to be loaded in the enclave, and the time taken to execute each step of EnGarde, reported as CPU cycles. Wall-clock time can be obtained by multiplying CPU clock cycles with the clock cycle time. A CPU with a clock rate of 3.5GHz as used in our experiments has 1/3.5 nanoseconds cycle time. Therefore, the 694,405,019 cycles it takes to disassemble Nginx, for example, consumes 198.4 milliseconds.

version 1.0.5. To perform this check, we first generate the SHA-256 hashes of all the functions of musl-libc v1.0.5. For enforcement, the policy module iterates through the instruction buffer of the code to be loaded in the enclave, and looks for all direct function calls. For each direct function call, the policy check computes the target of the call and then looks up the symbol hash table to get the function name of the target. If the target does not exist in the symbol hash table the check will mark the function call as invalid; otherwise, it will compute the SHA-256 hash of all the instructions of the function. Specifically, the policy module sequentially reads instructions starting from the computed target address and stops when it comes across an instruction that is at the beginning of another function. The policy module relies on the symbol hash table to identify whether an instruction address is at the beginning of a function. The policy check next compares the hash of the function in the executable with its hash in musl-libc. If the two hashes do not match, the client has not provided the required musl-libc; otherwise, the policy check continues with the next iteration until it reaches the end of the instruction buffer.

To compute the performance cost, we adopt the approach suggested in the OpenSGX paper [45] and assume that each SGX instruction takes 10K CPU cycles and non-SGX instructions run at native speed within the enclave. We leverage OpenSGX's performance counter and QEMU's instruction count [56] to count SGX and non-SGX instructions. We calculate the CPU cycles of non-SGX instructions by measuring the instructions per cycle by executing the loader natively without OpenSGX. Figure 2.3 presents the results of our experiments when running this policy check against different benchmarks.

Compliance for Stack Protection. Given the prevalence of buffer overflow vulnerabilities in low-level code, a number of modern compilers now give the option of emitting extra code to protect loads and stores to memory locations. Clang's `-fstack-protector` flag lets the LLVM compiler add a guard variable when a function starts and checks the variable when a function exits. For instance, when compiled with the flag, the following extra code is emitted:

```
19311:   mov %fs:0x28, %rax
1931a:   mov %rax, (%rsp)
193fe:   mov %fs:0x28, %rax
19407:   cmp (%rsp), %rax
1940b:   jne 1941f
1941f:   callq 8d5bf <__stack_chk_fail>
```

The two instructions at addresses `193fe` and `19407` check if the variable at the top of the stack is the same as the variable at `%fs:0x28`. If the values do not match, control will be transfered to the `__stack_chk_fail` function.

Clang also provides the `-fstack-protector-all` option which is similar to `-fstack-protector` except that *all* functions are protected. To check whether an executable is compiled with this flag, the policy module iterates through the instruction buffer and identifies the start of a function using the symbol hash table. Within each function, the policy check looks for instructions that affect the stack's variables (*e.g.,* `mov %rax,(%rsp)` in the above example). It then identifies the source operand of the instruction (`%rax`) and figures out the value of the source operand (`mov %fs:0x28,%rax`). As a next step, it checks if the function contains a `cmp` instruction with the source and destination are the stack's variable and the previous source operand, respectively. It also has to check that just preceding the `cmp` instruction, there is an instruction that computes the original value of the source operand (`mov %fs:0x28,%rax`). Finally, the policy looks for the `jne` and `callq` instructions. It computes the target of the `callq` instruction and checks the symbol hash table to verify that the target corresponds to the the `__stack_chk_fail` function.

Of course, our implementation of EnGarde's policy module is customized for Clang's stack protection instrumentation as emitted by the `-fstack-protector` flag. It can easily be customized to check stack protection instrumentation inserted by other tools, such as Google's AddressSanitizer [57], LLVM SoftBound [58], etc. Figure 2.4 presents the results of our experiments when running this policy check against different benchmarks executing in enclaves.

Benchmark	#Inst.	Disassembly	Policy Checking	Loading and Relocation
Nginx	271,106	719,360,640	713,772,098	128,662
401.bzip2	24,226	34,292,136	862,023,613	4,206
Graph-500	100,488	140,588,361	195,218,892	4,548
429.mcf	12,985	18,288,921	31,459,881	4,330
Memcached	71,677	137,877,497	325,442,403	8,081
Netperf	51,868	91,577,335	183,274,713	18,057
Otp-gen	28,217	43,053,386	217,302,816	5,355

Figure 2.4: Performance of EnGarde to check the *Stack protection* policy. As before, for performance numbers, we report the CPU cycles.

Restricting Indirect Function Calls. Protecting applications against control-flow hijacking attacks is one of the emerging concern due to the fact that attackers have recently focused on taking advantage of heap-based corruptions to overwrite function pointers to change the flow of a program. Control-flow Integrity (CFI)is a measure that guards against these attacks by restricting the targets of indirect control transfers to a set of precomputed locations.

We implemented a policy check to verify that executables are compiled with indirect function-call checks as proposed in recent work by Google (IFCC) [59]. IFCC protects indirect calls by generating instrumentation for the targets of indirect calls. It adds code at indirect call sites to ensure that function pointers point to a jump table entry. For example, the LLVM implementation of IFCC emits the following code for an indirect function call:

```
1b459:   lea 0x85c70(%rip), %rax
         #<__llvm_#jump_instr_table_0_1>
1b460:   sub %eax, %ecx
1b462:   and $0x1ff8, %rcx
1b469:   add %rax, %rcx
1b475:   callq *%rcx
```

To instrument executables with these checks, we use the LLVM/clang toolchain enhanced with the IFCC patch [60]. To check whether an executable is compiled with IFCC checks, EnGarde's policy module first figures out the range of the jump table by relying on the fact that all jump table entries have the following format:

```
a19d0 <__llvm_jump_instr_table_0_289>:
a19d0:       jmpq 41090 <ngx_execute_proc>
a19d5:       nopl (%rax)
```

EnGarde's policy module for this check iterates through the instruction buffer and looking for indirect function calls. It then verifies that before the indirect function calls, there is a sequence of instructions `lea`, `sub`, `and` and `add`, with data dependence between registers as shown in the code snippet above. It then computes the target of the indirect call and verifies that the

Benchmark	#Inst.	Disassembly	Policy Checking	Loading and Relocation
Nginx	267,669	821,734,999	20,843,253	128,668
401.bzip2	24,201	34,235,817	1,751,276	4,206
Graph-500	100,424	140,429,738	7,014,913	4,548
429.mcf	12,903	18,242,127	1,177,429	4,330
Memcached	71,508	138,231,446	5,301,168	8,081
Netperf	51,431	91,161,601	3,775,318	18,057
Otp-gen	28,132	42,829,680	2,334,847	5,355

Figure 2.5: Performance of EnGarde to check the *Indirect Function-Call* policy. As before, for performance numbers, we report the CPU cycles.

target is within the range of the jump table. Figure 2.5 presents the results of our experiments when running this policy check against different benchmarks.

2.6 Summary

In this chapter, we present the design and implementation of EnGarde, an enclave inspection library that preserves the security benefits offered by the SGX and allows the cloud provider to verify the client's SGX-based enclave against a set of policies mutually agreed by the cloud provider and the client. In EnGarde, the cloud provider and the client mutually trust the inspection library with policy enforcement. EnGarde achieves its goal by using SGX's hardware attestation and having an encrypted channel set up between the cloud provider and the client. EnGarde only allows the client content to execute if the content follows mutually agreed policies. We have evaluated the effectiveness of EnGarde by using it to enforce three popular security policies for various real world applications.

Chapter 3

Detecting Ransomware Attacks on Cloud-based Applications

Ransomware is an emerging threat and has caused substantial damage to individuals and businesses. There has been recent work by the research community to combat this type of malware. However, they all rely on monitoring filesystem activities and therefore contain complex code bases. Moreover, they fail to distinguish ransomware from benign programs that use encryption and compression.

This chapter introduces HRD, a system that addresses these shortcomings. HRD leverages hardware performance counters (HPCs) and machine learning to fulfill its goals. Specifically, HRD uses HPCs to measure low-level events in the entire system when running a training set with benign workloads and ransomware samples.

Our evaluation indicated that various machine learning models show high accuracy when running on the training set using cross-validation. We use the trained model that has the best performance to classify ransomware and benign programs in deployment. The evaluation indicates that our method is able to detect new ransomware with high accuracy and achieves 100% precision in distinguishing benign programs such as 7-zip and Internet Explorer (IE) from ransomware. We also demonstrated that HRD adds no additional latency to filesystem operations and consumes small amounts of memory and CPU time.

3.1 Introduction

Ransomware is a form of malware that encrypts a victim's files and asks for a ransom to release the decryption key. The ransom payment is often transferred to the attackers using an anonymous payment mechanism such as Bitcoin. Ransomware is becoming an increasing threat, using more sophisticated techniques and now causes millions of dollars in damages [32, 33].

For example, the WannaCry ransomware and its variants affected hundreds of thousands of individuals and organizations in 150 countries, causing a disruption of critical services [34–37]. It encrypts hundred different file types and appends .WCRY to the end of the file names and then asks the victims to pay a $300 ransom in bitcoins and doubles the amount after three days [34]. As another example, the Cyber Threat Alliance estimated $325M in damages caused by CryptoWall version 3 between January 2015 and October 2015 [61].

Existing malware detection systems have limitations in detecting ransomware because of two main reasons. *First*, systems that focus on detecting stealthy behaviors such as abnormal network activity or suspicious system calls are not effective against ransomware whose operations are similar to those of benign programs that use encryption or compression. *Second*, ransomware is easy to obtain and obfuscate which renders signature-based detection techniques ineffective [62]. As a result, the emergence of ransomware has attracted the attention of many security researchers.

Essentially, recent methods to detect ransomware rely on the filesystem layer to monitor the ransomware activity. For example, CryptoDrop [16] detects ransomware by monitoring real-time changes in user data. Specifically, they track filesystem related indicators to detect the existence of ransomware in the system. The *first* indicator is the entropy of the content of user files, content written by ransomware often has high entropy. The *second* indicator is file type changes where files infected by ransomware are often changed to different types. The *third* indicator is the similarity between two versions of the same file where the file before being infected by ransomware is often completely dissimilar to ransomware-encrypted content. Other indicators include the deletion of many files from a user's documents and the number of file types read and written by each process. Another effort to detect ransomware is UnVeil [17] that works by having full visibility into all filesystem modifications. Specifically, the system monitors system-wide filesystem accesses of user-mode processes formalized as access patterns that include the entropy of the I/O data buffer where a significant increase in the entropy between read and write data buffers at a given file offset is a sign of ransomware running on the system. UnVeil also makes further attempts to detect screen locker ransomware in which the attacker displays a ransom note to the victim that covers a significant portion of the display. UnVeil achieves this goal by taking series of screenshots of the target system and then analyzing

them using image analysis methods to determine if a large part of the screen has changed between captures. ShieldFS [18] is another work that focuses on building detection models to distinguish ransomware from benign processes. The models are based on calculating the entropy of write operations, frequency of read and write, folder-listing operations, fractions of files renamed, and the file-type usage statistics. ShieldFS also scans the memory of any suspected processes to search for the typical block cipher key schedules. ShieldFS uses a copy-on-write strategy that shadows the write operations with a virtual filesystem; If a file is affected by ransomware, the filesystem presents the original, mirrored copy to the user applications.

However, the above detection systems are prone to false positives. In fact, the authors of CryptoDrop acknowledge that legitimate programs such as GPG or 7-zip will be flagged as ransomware by their system. The reason is that these techniques often rely on file encryption as the signature behavior of ransomware [16, 17] regardless whether the encryption is legitimate or malicious. Specifically, these techniques monitor data changes and I/O request sequences to detect encryption operations performed on the data. In this work, we take another approach that not only efficiently detects ransomware-like behaviors but also reliably distinguishes whether the behaviors are exposed by ransomware or benign programs that perform encryption and compression. Our experimental results show that although both ransomware and encryption programs such as 7-zip encrypt users' files, they expose substantially different low-level event measurements.

Contribution In this work, we present an alternate solution for detecting ransomware, which does not require tracking the filesystem layer. Specifically, we propose HRD [1], a system that uses hardware performance counters (HPCs) to gather measurements of all hardware events in the entire system for different runs of benign workloads and ransomware samples. After having the event traces, we leverage machine learning to select the most discriminating events that clearly distinguish ransomware from benign workloads. We use HRD to measure the previously selected events for a large number of ransomware samples and benign programs from a training set. By evaluating the accuracy of various machine learning models on the training set, we found that the random forest model performs the best which consequently

[1] Hardware-based ransomware detector.

prompted us to apply the trained random forest model during deployment.

We have demonstrated that HRD is able to detect unseen ransomware including variants of the WannaCry ransomware with high accuracy. Our experiments showed that there exists discriminating low-level events that clearly distinguish activities performed by benign programs that do encryption and compression from those performed by ransomware. Our evaluation showed that HRD is able to distinguish benign programs that perform encryption and compression such as 7-zip and Internet Explorer (IE) with 100% precision.

HRD has a smaller code base than the filesystem-based systems. This is because filesystem-based systems require complex components for I/O monitoring while HRD simply collects performance measurements.

Unlike previous approaches that use HPCs to detect malware, our approach does not require changing the existing processor architectures. As will be shown in the next sections, commodity processor architectures are sufficient to detect ransomware. Our evaluation showed that HRD consumes small amounts of memory and CPU time and incurs no additional overhead to filesystem operations because it does not interfere with the filesystem.

3.2 Motivation

In finding an approach that has low-overhead but still can effectively detect ransomware, we pay attention to using hardware performance counters (HPCs) because of three main reasons: 1. HPCs incur low-overhead to the system whether they are used to profile performance characteristics of the system or are used to detect malware [63]. This is because HPCs can be sampled periodically and do not interfere with the operations of the system such as I/O operations. Therefore, HPCs can potentially address the high overhead incurred by previous approaches in detecting ransomware [16–18] 2. HPC-based methods raise the bar for malware writers to evade detection because malware writers have to take into account a broad range of microarchitectural events when producing malware variants to evade detection. The intuition behind this is that malware variants that are functionally equivalent tend to exhibit similar activities in the form of event measurements [64]. 3. There has been a trend by the research community to show the effectiveness of HPCs in detecting other forms of malware [64–69].

It is worth noting that previous work on detecting malware by leveraging HPCs has not

focused on detecting ransomware. Previous efforts either target Linux rootkits or Android malware while ransomware mostly runs on Windows platforms. Also, all malware samples used in previous works do not expose ransomware behaviors which involve encrypting victims' files. In this work, we attempt to detect behaviors of malware with a focus on ransomware behaviors on Windows platforms. As will be discussed in the rest of the chapter, ransomware does have different discriminating performance events compared to general malware [64–66].

Previous malware detection systems based on HPCs require a modification to processor architecture to be able to collect performance data of multiple events simultaneously. This is because commodity processors only allow measuring a limited number of events at any time. In this chapter, we will propose techniques that address this limitation and demonstrate that the current capability of commodity processors is sufficient for detecting ransomware.

3.3 Background on HPCs

Hardware performance counters (HPCs) were originally introduced as a method for system administrators to profile the performance characteristics of a computer system with little overhead or optimize system performance by counting low-level events [70, 71]. HPCs are supported by all modern processor platforms and HPC-based profilers are provided on every popular operating system (OS) [72, 73]. HPCs are capable of collecting a wide range of hardware related events such as cache hits/misses, retired instructions and branch mispredictions. Essentially, HPCs contain a set of control registers and counting registers that are memory-mapped at fixed memory addresses by the OS and can only be accessed in kernel mode. The control registers serve as an interface to specify low-level events that need to be monitored and the counting registers are the output that contains event counts. Each time a specified event occurs, its corresponding counting register will be incremented. The number of available HPCs and the events that can be monitored differ from one processor to another: Intel Pentium III supports a hundred events and provides two HPCs [63], the 3rd generation Intel Core processor family has hundreds of events and four HPCs. Although commodity processors provide hundreds of performance events, only four events can be monitored at any given time due to the limited number of HPCs. One of the key contributions of our approach is identifying the four events that clearly represent the behaviors of ransomware and using their measurements to detect ransomware.

3.4 HRD Overview

In this section, we first discuss the threat model and assumptions, the overview of our approach and methodology behind the system and the infrastructure for collecting performance events. We then describe the methods we use to select most important events on top of which we build machine learning classifiers used to classify ransomware from benign programs.

3.4.1 Threat Model and Assumptions

HRD targets only user space ransomware. To the best of our knowledge, there is no detection technique that can detect all malware. This assumption is similar to previous work on detecting ransomware [16, 17] and is reasonable given that the majority of ransomware perform malicious activities on user space programs and therefore leaves the OS kernel space untampered. We assume that ransomware can have arbitrary actions in the user space such as encrypting documents or erasing OS' files.

We also assume that ransomware does not attempt to manipulate HPCs to affect the measurements produced by HPCs. To the best of our knowledge, there are no real world ransomware samples that try to evade detection using this technique. Because HPCs can only be accessed in kernel mode, a user space ransomware will not be able the change the HPCs' value.

3.4.2 Methodology

The baseline workload. In all experiments, we use an Nginx web server [74] benchmarked with `ab` (the Apache HTTP server benchmarking tool) [75] as the baseline workload. The tool sends 100 requests to the web server at a time for a 612 byte web page from the beginning to the end of each experiment. Event measurements of this workload indicate the expected performance characteristics of the system when it is not infected by ransomware. In our experiments, each ransomware sample or benign program runs along with this baseline workload for a duration of 10 minutes. We choose that duration because most real-world ransomware, upon being triggered, starts showing their malicious behavior including encrypting files and showing popup windows in about 10 minutes.

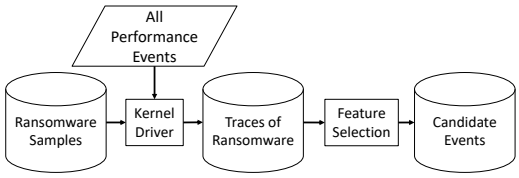
User Files. We create a collection of files with a total size of 2GB to serve as the target

of ransomware. The collection contains miscellaneous file types that are often tampered by real-world ransomware such as .doc, .pdf, .txt, .xlsx.

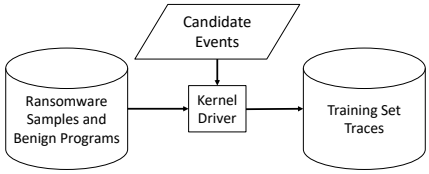
Figure 3.1 shows the overview of our approach which involves four primary phases. In the *first* phase, we perform feature selection to select a short list of candidate events from hundreds of available events which best represent ransomware execution. Specifically, we measure all performance events of the system when running 76 ransomware samples one after another. We also run the baseline workload multiple times to get 35 traces. The next section will describe the kernel driver that we use to collect event measurements. Note that we do not run any benign programs in this phase because the goal of this phase is to identify performance events whose values are clearly affected before and after the system is infected with ransomware. As will be shown later in the chapter, this phase effectively selects a small set of candidate events from hundreds of events.

The candidate events selected from the *first* phase is then used as the inputs of the *second* phase. We obtain 750 ransomware samples from VirusTotal and use 80% of them for this phase and use the rest to evaluate the performance of our system in deployment. Table 3.1 shows the family distribution of the ransomware samples. Note that the ransomware samples used this phase do not overlap with the samples used in the previous phase. In this phase, we measure the candidate events of the system when running the ransomware samples and three benign programs: 7-zip, Internet Explorer (IE), and AESCrypt. We run each of these benign programs 50 times to obtain 50 traces for each of them. We also measure the candidate events of multiple runs of the baseline workload in this phase. Table 3.2 breaks down the number of event traces corresponding to each program. All the traces collected in this step serve as the training set for the training phase.

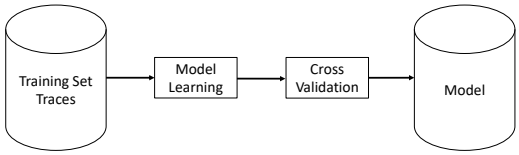
In the *training phase*, we evaluate the performance of various machine learning models using cross-validation and then select the model that has the best performance to use in deployment. For *deployment* phase, HRD uses the machine learning model obtained from the previous phase to classify measurements of the candidate events provided by the kernel driver. HRD analyzes event measurements in a sliding window of 10 minutes.



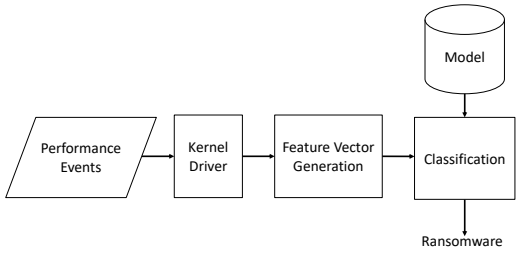
(a) Event selection.



(b) Collecting traces of candidate events for the training phase.



(c) Training phase.



(d) Deployment phase.

Figure 3.1: Overall workflow of HRD.

Ransomware Family	Samples
Crowti	18
Cryptodefense	15
Cryptolocker	31
CryptoWall	59
CTB-Locker	76
Filecoder	61
Locky	93
Reveton	129
TeslaCrypt	136
Tobfy	23
Virlock	65
Urausy	44

Table 3.1: Family distribution of ransomware samples used to collect training set traces.

Workloads	Number of Event Traces
Baseline	300
Ransomware	750
IE	50
7-zip	50
AESCrypt	50

Table 3.2: Breakdown of number of collected event traces.

3.4.3 HPC Trace Collection

The event collection infrastructure must meet several requirements:

- It should not interfere with the functionality of the OS and its applications. In other words, the system should be able to function normally even when the infrastructure crashes.
- Measurements are integrity protected, or in other words, are not tampered by ransomware.
- Measurements must incur low performance overhead.

The first requirement is to guarantee that the event traces we receive from the target OS are not intentionally polluted by ransomware as an attempt to avoid detection. Ransomware that has access to the traces might change them to make event counts similar to those retrieved from benign programs. The second requirement indicates that the infrastructure should not use up system resources so that it can be efficiently deployed in a real-world scenario.

While previous work on leveraging performance counters to detect malware focuses on

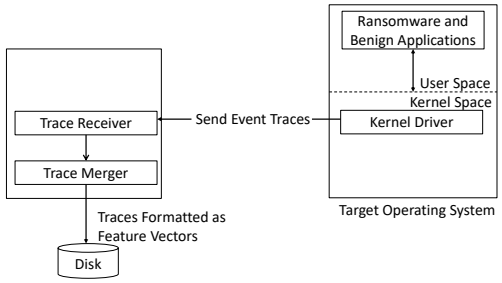


Figure 3.2: The design of the event collection infrastructure. Low-level events are measured by a kernel driver and are sent by the kernel driver to a receiver running on another host.

counting a small set of events [64–66], we are interested in counting all available events[2] when ransomware samples run on the system. The main reason is the previous work targets general malware most of which does not show ransomware behavior and therefore we might omit some key events that are good indicators to detect ransomware if we only measure a few events. It is worth noting that most architectures only have a limited number of HPCs which means only a few events can be counted at any moment. In order to count all available events, it is necessary to run the same ransomware instance multiple times and measure a different set of events in each experiment. To address this issue, we use virtual machines (VMs) to keep the system state to be the same between experiments. In particular, we create a VM snapshot and restore the system state from the snapshot after each experiment. Instead of using Vtune to measure performance events as done in previous work [67], we choose to implement a kernel driver to gather performance events because of several reasons. *First*, HRD is designed to deploy on a live system. As will be shown in the next sections, the kernel driver of HRD measures performance events in each sliding windows of 10 minutes and uses machine learning models to classify each window. Vtune is appropriate for systems that classify event traces in an offline manner [67]. *Second*, given the arms race of malware detection, ransomware might adapt to fingerprint the detection system to evade detection. The user space program used by Vtune to program and collect performance events from its kernel component could be fingerprinted by ransomware. Upon detecting that Vtune is running and gathering performance events, ransomware might stop or delay its harmful activities to avoid detection. HRD, on the

[2]Most architectures typically have hundreds of events.

other hand, functions mostly at the kernel space and does not contain a userspace component with GUI. *Third*, we want to keep the code base of the driver as small as possible. Since most processor architecture requires ring 0 privilege to access HPCs, the code that needs to interact with HPCs must run in the kernel space. Hence, we develop a kernel driver that programs HPCs to measure event counts and is automatically loaded when the OS boots up. The driver collects event measurements in each unit of time[3] to form time series of event counts that are stored in a kernel buffer. In each experiment, we have the driver construct a time series of 10 minutes[4]. The driver configures performance counters to count events taking place in both user space and kernel space and it resets counting registers before counting events. Once the driver finishes an experiment it sends the event trace stored in the buffer to a remote host for analysis via a socket connection. Since event traces are stored in a kernel buffer and all operations taken to send the traces to the remote host take place in the kernel space, ransomware is unable to tamper with the traces by our threat model. Also, by measuring events in each small unit of time and putting all related operations in the kernel space, the overhead incurred by the infrastructure is negligible as will be shown by its low memory footprint and computation in section 3.5. It is worth mentioning that unlike other systems [16, 17] that detect ransomware by interfering with the filesystem operations and therefore incur signification latency to read/write operations, our infrastructure does not add additional overhead to read/write operations. Figure 3.2 depicts the overall design of the infrastructure. At the beginning of each experiment, the analysis host sends a command to a daemon running on the target OS specifying the full path to a ransomware sample or benign workload which is then executed by the daemon.

Once the analysis host receives the traces of all events for each instance of ransomware or benign program, it merges them into a feature vector and appends it to a text file to be used for analytic tasks.

We use Windows OS in our infrastructure since it is the main target of most real-world ransomware. We use Windows 10 running inside a VMWare Workstation VM [76] with virtualized CPU performance counters and create a snapshot of the VM to restore system states at the beginning of each experiment. The VM has an Intel i5 3.50GHz Haswell processor and

[3]Default is every 1 second.

[4]The duration is configurable.

4GB of memory. Internally, the driver that collects event measurements contains a table that maps each event with the relevant content that should be written to a control register. The content includes the event number, set bits to measure both user mode and kernel mode events and many other fields. When the driver is first loaded, it establishes a socket connection to the analysis host to receive commands consisting of events that need to be measured. It next closes the connection to avoid the overhead and then starts measuring events. Upon finishing the work, the driver sets up another connection to the analysis host to transfer event traces to the host for analysis.

3.4.4 Most Significant HPCs

Feature Selection

Each event trace is represented as a feature vector consisting of time series of 205 events of the Haswell microarchitecture [77]. Each time series contains 600 data points[5] which translates to having $\sim 120,000$ features in each feature vector. We also add a binary label at the end of each feature vector to mark whether the feature vector belongs to a benign program or a ransomware sample.

There are two primary reasons that require us to select and use a subset of features in a feature vector. First, there are only four events that can be measured simultaneously due to the limited number of performance counters. Second, the number of observations or feature vectors is relatively small compared to $\sim 120,000$ features. This can potentially lead to problems caused by the curse of dimensionality [78]. Hence, it is necessary to do feature selection to select only the most relevant features that can be used to build optimal classifiers. Therefore, the ultimate goal of feature selection is to extract the most discriminating features that correspond to a group of four events.

For feature selection, we collect event traces of two types of programs: the benign workload and ransomware samples. Specifically, we gather 35 traces of the benign workload and traces of 76 ransomware samples. Table 3.3 depicts the families of the ransomware samples used in this phase.

The first step we take in feature selection is normalizing the values of the feature vectors

[5]Events are counted every one second for a duration of 10 minutes.

Ransomware Family	Samples
CryptoWall	9
CTB-Locker	14
Filecoder	11
Locky	15
Reveton	10
TeslaCrypt	17

Table 3.3: Family distribution of ransomware samples used in feature selection.

that have some features with small values and the others with large values. Each normalized value for a feature i, $normalized_i$, is derived from the current value, $current_i$, the minimum value of the feature, the maximum value of the feature and the mean value of the feature as below:

$$normalized_i = \frac{current_i - mean}{max - min} \tag{3.1}$$

After having normalized feature vectors, we leverage *regularization* [79] to perform feature selection. *Regularization* is a method that helps prevents overfitting a machine learning model by adding a penalty to that model. The penalty is typically represented as $\lambda\ \vec{v}$ where λ is a tunable parameter indicating the amount of regularization and $\vec{v}$ is a vector of coefficients of the model whose form depends on the specific regularization type. LASSO (least absolute shrinkage and selection operator) [80] is a popular regularization method because its L1 penalty creates a sparse solution by forcing weak feature to have zero coefficients. In particular, the objective of LASSO is to minimize the value of the following equation:

$$min\left\{\frac{1}{N}\sum_{i=1}^{N}(y_i - \sum_{j=1}^{p}\beta_j x_{ij})^2 + \lambda\sum_{i=1}^{p}\beta_i\right\}$$

where N is the number of observations, p is the number of coefficients, y_i and (x_{i1},...,x_{ip}) are the response variable and the predictor variables of the ith observation respectively, β_i are the coefficients and λ is a tunable parameter. LASSO uses the value of coefficients in its vector to add a penalty $\lambda \sum_{i=1}^{n}\beta_i$ to a model. Since LASSO is a natural solution for feature selection, we apply LASSO to select the most relevant performance events related to ransomware behaviors. Specifically, we select a sequence of 100 values for λ, use the same weight for each observation and use 20-fold cross-validation to find the optimal value of λ. Table 3.4 demonstrates the

Value of λ	Number of Selected Features	Number of Selected Events
lambda.min	110	60
lambda.1se	15	10

Table 3.4: Number of selected features and events by running LASSO with lambda.min and lambda.1se.

Events	Timestamps	Coefficients
Number of multiply packed/scalar single precision uops allocated	1	0.015461187
Cycles the RS is empty for the thread	64	-0.284798598
Qualify conditional near branch instructions executed, but not necessarily retired	60 62	0.274112265 0.094262836
Number of far branches retired	59 62	0.005279482 0.040508429
Number of near branch instructions retired that were taken but mispredicted	6	-0.036647073
Number of X87 FP assists due to input values	96	-0.047242007
Cycles with any input/output SSE* or FP assists	74	-0.027642710
Retired load uops with locked access	59	0.095108911
Retired load uops that split across a cacheline boundary	62 72	-0.044511884 -0.033722482
Retired store uops that split across a cacheline boundary	35 60 67	-0.074874024 -0.047379502 -0.016483707

Table 3.5: Events selected by running LASSO with lambda.1se.

number of features extracted by running LASSO with lambda.min, which gives minimum mean cross-validated error, and lambda.1se, and lambda.1se, which gives the most regularized model such that error is within one standard error of the minimum. In the former case, we have selected 15 features belonging to 10 low-level events and there are 110 features associated with 60 events in the latter case. Hence, we choose lambda.1se since it gives us a smaller set of events. Table 3.5 reports the selected features with their timestamps and coefficients using lambda.1se where some features belong to the same event; for example, the number of retired far branches has two features at timestamp 59 and 62, which means the frequency of this event at these two timestamps are good indicators to detect ransomware.

Schemes For Selecting Four Events

Most current processors provide only four HPCs allowing up to four events that can be monitored simultaneously. As a result, previous approaches [64, 81] proposed to modify existing processor architectures to support collecting more events simultaneously. We argue that our approach can work with current processors and that four events are sufficient to detect ransomware. Therefore, we need to further filter four events from the 10 events selected in the previous step. In order to avoid bias in selecting events, we come up with three schemes to rank and select events. In the first scheme, we calculate the sum of all non-zero coefficients for each event. We get the average of non-zero coefficients in the second scheme and count the number of non-zero coefficients of each event in the third scheme. We then rank the events by decreasing order of their values in each scheme and select the top four events. Table 3.6, Table 3.7 and Table 3.8 depict the ranked events obtained in the first, second and third scheme respectively.

Events	**Sum of Coefficients**
Qualify conditional near branch instructions executed, but not necessarily retired	0.368375
Cycles the RS is empty for the thread	0.284799
Retired store uops that split across a cacheline boundary	0.138737
Retired load uops with locked access	0.0951089
Retired load uops that split across a cacheline boundary	0.0782344
Number of X87 FP assists due to input values	0.047242
Number of far branches retired	0.0457879
Number of near branch instructions retired that were taken but mispredicted	0.0366471
Cycles with any input/output SSE* or FP assists	0.0276427
Number of multiply packed/scalar single precision uops allocated	0.0154612

Table 3.6: Sum of coefficients of each event.

To evaluate the power of each scheme, we run five machine learning classifiers using 10-fold cross validation on each subset of traces selected by each scheme. Table 3.9 demonstrates the area under the ROC curve of each classifier in each scheme. We also report the performance in the ideal case when all features of 10 events are used. Overall, the performance in all case is promising with the ideal case gives the best performance across all classifiers. The second and the third scheme also have equally good performance and are more practical to deploy since they require only four events. As a result, we choose the second and the third scheme to select

Events	Average of Coefficients
Cycles the RS is empty for the thread	0.284799
Qualify conditional near branch instructions executed, but not necessarily retired	0.184188
Retired load μops with locked access	0.0951089
Number of X87 FP assists due to input values	0.047242
Retired store μops that split across a cacheline boundary	0.0462457
Retired load μops that split across a cacheline boundary	0.0391172
Number of near branch instructions retired that were taken but mispredicted	0.0366471
Cycles with any input/output SSE* or FP assists	0.0276427
Number of far branches retired	0.022894
Number of multiply packed/scalar single precision μops allocated	0.0154612

Table 3.7: Average of coefficients of each event.

Events	Number of Coefficients
Retired store μops that split across a cacheline boundary	3
Qualify conditional near branch instructions executed, but not necessarily retired	2
Number of far branches retired	2
Retired load μops that split across a cacheline boundary	2
Number of multiply packed/scalar single precision μops allocated	1
Cycles the RS is empty for the thread	1
Number of near branch instructions retired that were taken but mispredicted	1
Number of X87 FP assists due to input values	1
Cycles with any input/output SSE* or FP assists	1
Retired load μops with locked access	1

Table 3.8: Number of coefficients of each event.

events to be measured.

Correlation Between Ransomware Activities and the Frequency of the Selected Events

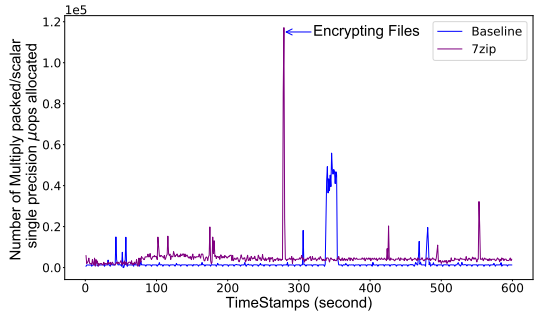
In this section, we describe our key findings why the above selected events serve as the best indicators of ransomware behaviors. We first notice that programs that use cryptographic operations tremendously increase the number of multiply packed/scalar single precision micro-operations, as shown in Figure 3.3a, Figure 3.3b and Figure 3.3c. This is the event that counts the number of micro-operations allocated to perform floating point operations. The Intel architecture offers floating point assists that address the concern caused by denormal and underflow numbers in floating point arithmetic [82, 83]. In the figures, we demonstrate the event counts

Schemes	Naive Bayes	Logistic Regression	SVM	Random Forest	Neural Networks
1	0.978	0.935	0.882	0.981	0.984
2	0.993	0.971	0.928	0.985	0.984
3	0.990	0.978	0.890	0.995	0.995
Ideal	1	0.992	0.958	0.997	0.999

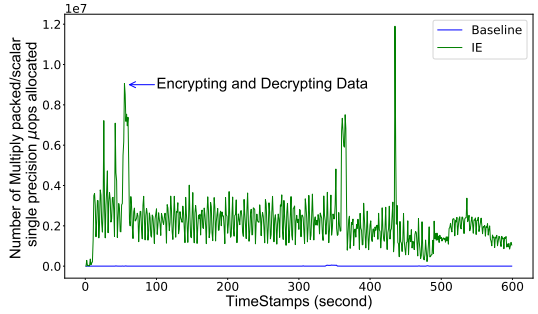
Table 3.9: Area under the ROC curve of five classifiers under different schemes.

caused by 7-zip, IE, and a ransomware sample along with measurements of the event for the baseline workload. 7-zip spends 275 seconds to compress all the files to a .zip file which is then encrypted using AES. This complies with the event measurements for 7-zip where the timestamps around 275 seconds have high event counts and the other has relatively low event counts. On the other hand, IE shows considerably high event counts during its runtime. This is because we have IE automatically browse multiple HTTPS websites including some HTTPS video streaming websites such as Youtube. Most of the cryptographic operations in this case are performed by IE to decrypt the encrypted video streams. During the runtime of the ransomware sample, we also notice a significant increase in the frequency of the event. Interestingly, this ransomware sample starts encrypting files shortly after it infects the system; it then stays dormant for a short period of time before continuing its action.

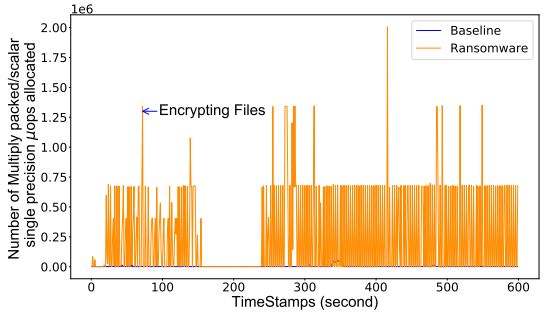
We also notice that the execution of ransomware drastically change the number of CPU cycles when the reservation station (RS) is empty. Reservation stations are structures that support for out-of-order execution [84] by waiting for and storing information needed to execute instructions such as instruction operands. The RS improves performance because the processor can process instructions that can run immediately instead of wasting cycles for waiting for a data dependent instruction to retrieve data. A decrease in the number of cycles when the RS is empty implies that the processor issues more instructions to the RS. 7-zip slightly changes the counts of this event during its execution which lasts 275 seconds. After that, the event measurement closely matches that of the baseline workload. IE, on the other hand, fairly reduces the counts of this event during its runtime. This is because many IE's tabs keep executing instructions from beginning to the end to browse multiple web pages. Among the three programs, the ransomware sample affects this event the most. It greatly reduces the counts of this event



(a)



(b)



(c)

Figure 3.3: Number of multiply packed/scalar single precision μops allocated.

for the entire duration. This could be explained by the fact that ransomware makes the processor execute more instructions including encrypting files, scanning directories, communicating to the command and control server(C&C).

It is not trivial to analyze why a particular event is significant to malware [67]. One method to solve this problem is instrumenting the OS kernel to collect performance traces at various points along the control flow and then comparing the control flow traces of the system before and after it is infected with ransomware. This comparison is used to identify the code that causes changes in the performance measurements [67].

3.4.5 Using HPCs and Machine Learning Methods to Detect Ransomware

Classifiers

Given the candidate features, we next gather numerous event traces of a collection of programs and then use the traces to train multiple machine learning classifiers. Specifically, we collect event measurements of 80% of 750 ransomware samples we get from VirusTotal and three benign programs: 7-zip, IE, and AESCrypt. We use IE to browse 23 web pages of various types such as news, entertainment, sports. It is worth noting that we intentionally browse some web pages that use the HTTPS protocol to evaluate whether our method could distinguish the cryptographic operations used by HTTPS from the ones used by ransomware. We also want to evaluate the effectiveness of our method by demonstrating that it could distinguish a benign program like 7-zip from ransomware. For this, we use 7-zip to compress and encrypt the same files used in experiments with ransomware which consequently makes 7-zip expose similar behaviors as ransomware. Similarly, we use AESCrypt to encrypt the same set of files and delete each original file after each file encryption. In these experiments, we also measure the candidate performance events of the baseline workload. We then use the scikit-learn [85] Python library to run five classifiers using 10-fold cross validation on the traces: Naive Bayes, Support Vector Machine (SVM), Logistic Regression, Random Forest, and Neural Network.

Figure 3.4 and Figure 3.5 demonstrate the area under the ROC curve of the classifiers in the second and third scheme respectively. All classifiers show good performance with the random forest model performing the best in both schemes. Therefore, we save the random forest model and use it to predict class labels of the testing set in the next section.

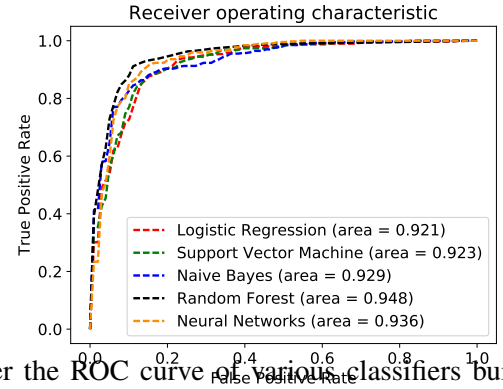


Figure 3.4: Area under the ROC curve of various classifiers built using the second feature selection scheme.

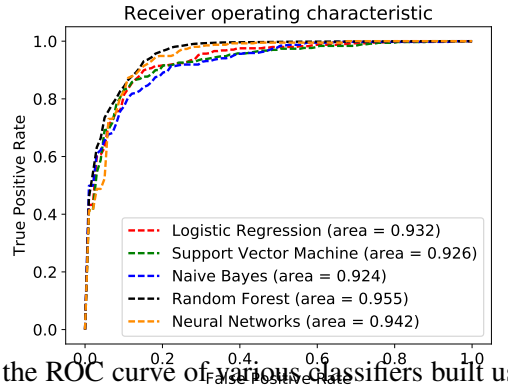


Figure 3.5: Area under the ROC curve of various classifiers built using the third feature selection scheme.

Effectiveness Evaluation

In this section, we demonstrate that the random forest classifiers selected in the previous section can effectively detect new ransomware in the testing set and can also distinguish benign programs from ransomware. We use the classifier to predict the class labels of ransomware traces as well as IE, 7-zip, and AESCrypt traces. Using the default threshold 0.5 for probability, we report the accuracy of the classifier in predicting the class labels for each type of program in Table 3.10. The classifier can detect new ransomware with high accuracy and also can distinguish IE and 7-zip from ransomware with 100% precision. Prediction accuracy of AESCrypt is slightly lower with 84.7% and 87.3% for the second and the third scheme respectively. To the best of our knowledge, 7-zip has a higher prediction accuracy than AESCrypt because it compresses files in addition to encrypting them. These results prove that our approach does

Schemes	Ransomware	IE	7-zip	AESCrypt
2	0.820	1	1	0.847
3	0.820	1	1	0.873

Table 3.10: Accuracy of the random forest model in predicting class labels of different programs.

better than previous systems such as CryptoDrop [16] in distinguishing benign programs from ransomware. CryptoDrop tends to flag compression programs like 7-zip or any other programs exposing ransomware behavior as ransomware. Figure 3.6 and Figure 3.7 depict the area under the ROC curve of the random forest classifiers in classifying the testing data for the two schemes.

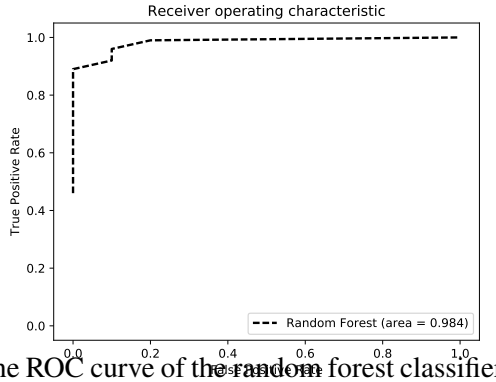


Figure 3.6: Area under the ROC curve of the random forest classifier in classifying ransomware in the testing set selected using the second scheme.

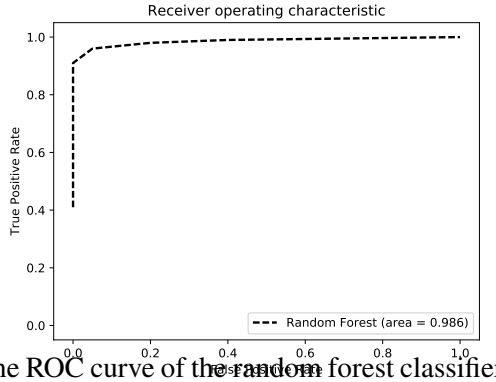


Figure 3.7: Area under the ROC curve of the random forest classifier in classifying ransomware in the testing set selected using the third scheme.

Schemes	Detection Accuracy
2	0.86
3	0.94

Table 3.11: Accuracy of the random forest model in detecting the WannaCry ransomware.

3.4.6 Case Study: the Wannacry Ransomware

In this section, we describe the effectiveness of HRD in detecting the Wannacry ransomware family, a ransomware family that has made the headlines recently. Note that, while there are samples of this ransomware family executing in the kernel space, we only demonstrate the capability of HRD in detecting the samples running only in the user space. In our experiments with this type of ransomware, we notice that it starts its malicious activities a few seconds after it infects the system. It communicates with its C&C server and shortly after that encrypts the files in the system. It takes 16 minutes for this ransomware to encrypt 2GB of documents in the system. This ransomware then shows a pop-up window asking its victim to pay $300 worth of bitcoin to a bitcoin's address within three days. If the victim fails to pay the demanded amount, they will lose their files. To evaluate the effectiveness of our approach in detecting variants of the Wannacry ransomware, we select only the samples that are flagged as belonging to the Wannacry family by VirusTotal for the testing data and use all other ransomware samples for the training data. Table 3.11 shows the accuracy of the random forest classifiers in predicting the class labels for the testing data. Overall, our approach achieves higher accuracy in detecting the Wannacry ransomware than other types of ransomware. This could be explained by the aggressive nature of the Wannacry family which keeps searching and encrypting files of users. Figure 3.8 and Figure 3.9 depict the area under the ROC curve of the random forest classifiers in classifying the testing data for the two schemes.

3.5 Discussion

HRD in Deployment

HRD's deployment model is quite similar to other machine learning based malware detection systems. When we deploy HRD on a device such as a desktop or a server, the kernel driver continuously measures the device's 4 performance counters every second. HRD then uses a

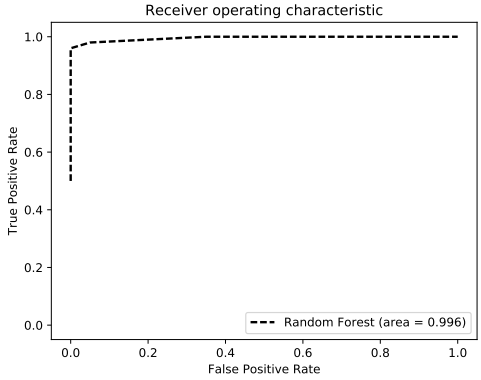


Figure 3.8: Area Under the ROC Curve of the Random Forest Classifier in Classifying the Wannacry Ransomware in the Testing Set Selected Using the Second Scheme

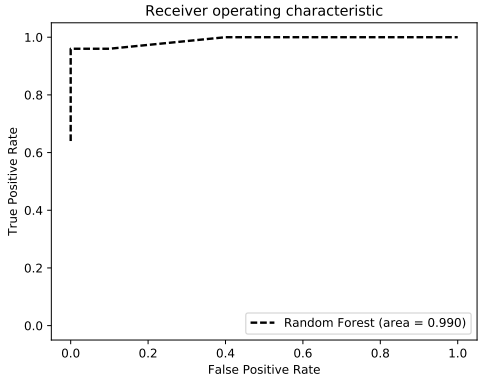


Figure 3.9: Area Under the ROC Curve of the Random Forest Classifier in Classifying the Wannacry Ransomware in the Testing Set Selected Using the Third Scheme

sliding window approach to convert the measurements into feature vectors. Recall that a feature vector is generated from measurements taken over a 10 minute window. Hence, HRD uses a sliding window of size 10 minutes and generates a feature vector for every window of measurements. The window may slide at the granularity of 1 second, or more, in order to reduce performance overhead.

HRD then applies the model learned in the training phase to the feature vectors generated from the sliding windows and classifies the vectors as ransomware or benign. If a window is classified as ransomware, then HRD alerts the device's user or administrator.

Similar to other machine learning based malware detection systems, new classification

Ransomware	#Low Similarity	#Medium Similarity	#High Similarity
CryptoWall	13	38	36
TeslaCrypt	17	42	28
TCB-Locker	22	40	25
Virlock	19	37	31
Cryptolocker	24	36	27

Table 3.12: Similarity of time series of events on Haswell and Nehalem microarchitecture.

models are generated periodically for a number of reasons. For example, availability of new training data such as traces of new benign programs and ransomware, and availability of new classification algorithms may necessitate retraining and model regeneration. When a new model is generated, HRD replaces its older classification model with the new model.

The Effectiveness of the Method on Different Processors Although we have demonstrated the effectiveness of our approach on a specific processor microarchitecture, we have proved that the approach works on other platforms as well. Specifically, we show that the time series of events measured on two microarchitectures: *Haswell* and *Nehalem* [86] show a strong correlation. For this, we step by step run five ransomware samples and measure 87 performance events that are supported on both processors. We then use the Pearson correlation coefficient method (PCC) [87] to measure the correlation between two time series of an event on the two processors, PCC outputs a value within the range [-1,1] with -1 indicating negative linear correlation, 0 implying no linear correlation and 1 indicating positive linear correlation. We divide the similarity between two time series into three groups based on their correlation scores: low similarity with the correlation between -1 and 0.3, medium similarity with the correlation between 0.3 and 0.8 and high similarity with the correlation between 0.8 and 1. Table 3.12 shows the distribution of number of events within each group on both microarchitectures. Most events have medium and high correlation of time series on both processors which implies that the values of performance events can also be used to train classifiers to detect ransomware on the Nehalem microarchitecture.

Detection of Other Benign Applications with Ransomware-like Behavior In this work, we trained HRD to distinguish three benign programs IE, 7-zip, and AESCrypt from ransomware. To distinguish other benign programs with ransomware-like behavior such as BitLocker from

Workload	CPU Cycles	Size of the Trace Buffer (KB)
Baseline	33,186,396	20.1
Ransomware	35,841,621	20.8

Table 3.13: CPU cycles and memory consumed by the infrastructure to measure four events for 10 minutes.

ransomware, HRD needs to be trained against those programs. However, we could also choose to whitelist those programs. In other words, HRD will not be used to classify the whitelisted programs; these programs will be considered benign.

Performance Evaluation In this section, we evaluate the performance of the infrastructure by measuring CPU utilization and memory consumption needed to count events. In particular, we report the CPU cycles and the size of the trace buffer needed by the infrastructure to count four events for the baseline workload and for a representing ransomware sample for a duration of 10 minutes. We use the rdtsc instruction to calculate CPU cycles. Table 3.13 depicts the results. The infrastructure incurs a negligible amount of CPU cycles and memory which makes it practical for real world deployment.

Timely Detection Table 3.14 and Table 3.15 show the timestamp of each event when ransomware behavior is best detected. In the second scheme, ransomware can be detected after 96 seconds since it first runs. At that timestamp, the values of all four performance events are available for detecting ransomware behavior. The third scheme even requires only 72 seconds to recognize ransomware execution on the system.

Events	Timestamps
Cycles the RS is empty for the thread	64
Qualify conditional near branch instructions executed, but not necessarily retired	[60,62]
Retired load uops with locked access	59
Number of X87 FP assists due to input values	96

Table 3.14: Scheme 2: timestamps of events when ransomware are best detected.

Resilience to Rootkits Our method is not resilient to rootkits manipulation since rootkits could potentially avoid detection by controlling performance counters. However, since most popular ransomware run in the user mode and do not exploit the kernel space, our method is capable to

Events	Timestamps
Retired store uops that split across a cacheline boundary	[35,60,67]
Qualify conditional near branch instructions executed, but not necessarily retired	[60,62]
Number of far branches retired	[59,62]
Retired load uops that split across a cacheline boundary	[62,72]

Table 3.15: Scheme 3: timestamps of events when ransomware are best detected.

detect ransomware without being subverted.

3.6 Summary

In this chapter, we presented HRD, a system that takes an exclusive approach in detecting ransomware. HRD does not monitor the filesystem layer like previous approaches but instead relying on HPCs to collect hardware events and machine learning to select representing features and build optimal classifiers that can be used to effectively detect ransomware. The evaluation of HRD shows that our method consumes small system resources and more importantly introduces no latency overhead to filesystem operations. Also, HRD was able to detect 750 ransomware samples including variants of the WannaCry ransomware with high accuracy and was also able to benign programs from ransomware with 100% precision. We have also demonstrated that our approach does not require hardware changes and works with different processor families.

Chapter 4

Exploring Infrastructure Support for App-based Services on Cloud platforms

Major cloud providers have recently been building cloud markets, which serve as a hosting platform for "cloud apps": VMs pre-installed with a variety of software stacks. Clients of cloud computing leverage such markets by downloading and instantiating the apps that best suit their computing needs, thereby saving the effort needed to configure and build VMs from scratch.

This chapter argues for a richer ecosystem of cloud apps. We envision a market-place of apps that can interact with client VMs in a rich set of ways to provide a number of services that are currently supported only by cloud providers. For example, clients can use VM apps to deploy virtual machine introspection-based security tools and various network middleboxes on their work VMs without requiring the cloud provider to deploy these services on their behalf. We present a taxonomy of cloud apps, investigate the design space of building such an app ecosystem, and present a security architecture to contain untrusted third-party cloud apps. We demonstrate the utility of our model by demonstrating and evaluating a number of security tools built as cloud apps, thereby enabling the vision of security-as-a-service.

4.1 Introduction

Infrastructure-as-a-Service (IaaS) cloud platforms such as Amazon EC2 and Windows Azure offer customers full access to virtual machines (VMs) whose software stacks they can customize and configure according to their needs. Enterprise-level clients with complex computing needs benefit because they can flexibly configure and execute their computations within these VMs. However, a large majority of clients often have standard computing needs (*e.g.,* they may simply want to host a Web or database server on the cloud), and may lack the resources and expertise

to set up and configure VMs from scratch. This problem has motivated major cloud players to build *cloud markets* [38] to distribute VMs pre-installed with software stacks that address the needs of such clients. Vendors have also been developing the supporting infrastructure necessary to support such *cloud apps* (*e.g.,* [88–90]).

In a cloud market, cloud providers or third-party developers build VMs customized for a variety of standard workflows, and publish images of these VMs in the market as cloud apps. Clients simply choose these apps to get started with their computing needs, thereby providing a more agile and hassle-free cloud computing experience. For example, Amazon allows publishers to create and publicly offer VM images, known as *Amazon Machine Images* (AMIs) [91], that execute on EC2. AMIs that offer a variety of standard software stacks (e.g. LAMP or SQL database software) are now available, which customers can directly instantiate to their specific domains. Publishers who create AMIs can also decide whether the AMIs must be paid or free.

We show that there are a number of benefits to enriching this nascent notion of cloud markets and cloud apps. In particular, on current cloud markets, the notion of a cloud app is largely restricted to VMs with different operating system versions, distributions, and other system or application software. In contrast, the model developed in this chapter allows apps to cooperate and interact with each other to support complex operations.

To illustrate the benefits of a richer model of cloud apps, consider that we want to offer security tools as a cloud-based service. That is, we would like to set up a VM equipped with standard security tools such as network intrusion detection systems (NIDS), firewalls, and sophisticated malware detectors, such as those based on virtual machine introspection (VMI) [92–94]. To benefit from the tools offered by this VM on current cloud markets, clients are required to install their software stacks within this VM. Instead, we aim for a model where this *security app* directly interacts with the client's work VMs (perhaps themselves downloaded from the cloud app market) to provide its services. For example, the security app must be able to monitor all incoming and outgoing network traffic from the client VMs, filtering this traffic using its NIDS and firewall.

We envision a cloud app market where cloud apps, implemented as VMs, offer standard utilities such as firewalls, NIDS, storage encryption, and VMI-based security tools. Cloud apps

can also implement a host of other non-security-related utilities, such as memory and disk deduplication, and network middleboxes such as packet shapers and QoS tools. Clients can leverage these utilities by simply downloading the appropriate cloud apps, and *linking them suitably with their work VMs*. The key challenge in realizing this vision on current cloud computing environments is that such interaction between VMs is disallowed. Each virtualized platform has one privileged VM (also called the *management VM*), controlled by the cloud provider, that supervises the execution of client VMs. The management VM oversees all I/O from client VMs, and completely isolates VMs from each other. While such isolation is desirable across VMs of *different clients*, it also prevents VMs belonging to the same client from interacting in useful ways.

In this chapter, we present the design and implementation of a complete ecosystem to support rich cloud apps. In particular, our contributions are as follows:

- We present a taxonomy of cloud apps, ranging from standalone apps to ones that involve complex system- and network-level interactions with other VMs. We use this taxonomy to motivate the key requirements of an ecosystem that supports such cloud apps.
- We present an end-to-end overview of the various components of the ecosystem from a client's perspective. In particular, we develop the notion of *cloud app permissions* to allow clients to reason about and control the behavior of third-party apps that they may download from a cloud app market. We also present techniques for a client to compose the functionality of multiple cloud apps within a single app.
- We explore the design options, such as hypervisor modifications, nested virtualization and network-level support, to implement various classes of cloud apps, such as those that offer system-level introspection, those that act as network middleboxes and those that offer storage-level services. We also explore the benefits and tradeoffs of each design option.
- We present an implementation of our design atop the KVM hypervisor, and quantify the performance of various design options.
- Finally, we demonstrate the utility of our model by building and evaluating a number of security-related cloud apps, and showing that clients can use these apps to realize the vision of security-as-a-service.

4.2 Threat Model

We assume a standard IaaS cloud computing model, where a cloud provider, *i.e.,* an entity such as Amazon or Microsoft, provides computing infrastructure. Clients rent resources from the cloud provider, and run their virtual machines on the cloud provider's hardware. We assume that cloud apps are hosted on a cloud app market, which is supported either by the cloud provider or a third-party.

In our threat model, the cloud provider is assumed to be trusted, and does not intentionally violate the security and privacy of clients. Thus the computing platform, consisting of the physical hardware and the hypervisor, is trusted. To a certain extent, clients can verify their trust in the cloud provider using trusted hardware, *e.g.,* using attestation based on the TPM). We also assume that the cloud provider's infrastructure is equipped with IOMMU units to enable I/O virtualization. The implication of placing trust in the cloud provider is that we cannot protect client VMs from attacks that involve government subpoenas or even malicious cloud administrators. This setting may therefore be unsuitable for cloud clients with stringent security and privacy requirements, *e.g.,* finanial organizations and health providers. However, we believe that trusting the cloud is a reasonable assumption for a large number of cloud clients. Our goal is to explore a number of useful applications that are enabled by placing such trust in the cloud provider.

We assume that cloud apps are not trusted, and may be written by third-parties, and are not trusted. However, as we will describe, in our app model each cloud app explicitly states the permissions that it requires to perform its operations. Our model also gives clients control over the I/O of cloud apps. We assume that clients examine permissions prior to installing apps and will devise suitable security policies to govern their I/O. In many ways, this situation resembles the one on mobile app markets, where clients download and execute mobile apps written by third-parties. On such markets, it is well understood that malicious apps can make their way into app markets, and that permissions are inadequate to prevent malicious apps from leaking sensitive information that belongs to clients. It is quite likely that the same issues (and much the same kind of research) applies even to the setting of cloud apps. We do not attempt to address those issues in this chapter, but rather focus on the infrastructure support necessary to enable

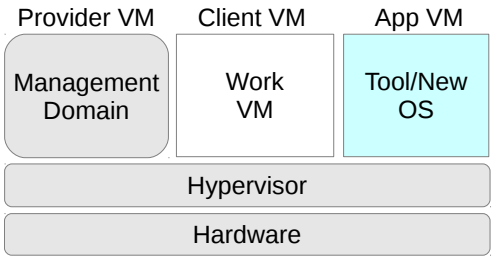


Figure 4.1: A standalone cloud app.

cloud apps.

Finally, we do not also explicitly aim to protect against side-channel attacks, in which one client attempts to infer sensitive information about other clients or the cloud provider itself [95, 96]. We believe that such threats are orthogonal to our work, and that defenses developed for such threats can be incorporated into a cloud platform that supports the app model that we develop.

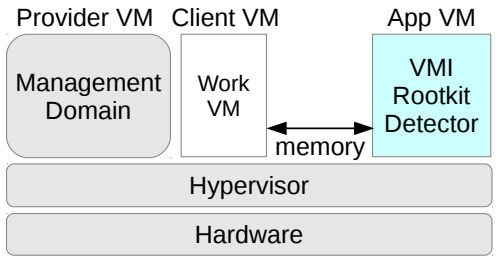
4.3 Taxonomy of Cloud Apps

In this section, we present a taxonomy of cloud apps based upon their functionality. We broadly categorize apps into four groups: (1) standalone apps, (2) low-level system apps, (3) I/O interceptors, and (4) bundled apps.

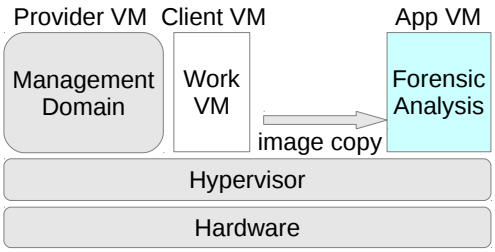
4.3.1 Standalone Apps

Standalone apps (Figure 4.1) represent the kind of apps that are currently available on cloud markets (e.g., AMI images available currently on Amazon's cloud market). These apps could include VMs with new operating system distributions, versions, or drivers supporting new hardware. Clients select cloud apps, instantiate them, and build software and services atop the provided environment. These cloud apps are self-contained and do not interact with the client's VM.

To create a standalone cloud app, a publisher configures and installs an OS along with one or more user-level tools customized for a specific workflow. For example, a Web server cloud app will likely contain the entire Apache toolchain. Likewise, it is also possible to imagine a security cloud app that is pre-installed with user-space anti-virus tools. Clients that purchase such a VM app will benefit from the pre-installed anti-virus tools, thus saving them the effort



4.2(a) An app that performs reads and writes



4.2(b) An app that only performs reads

Figure 4.2: Low-level system apps.

of installing and configuring these tools themselves. They can simply install their enterprise software, which will automatically be protected by the anti-virus tools. Note that in the setting of standalone cloud apps, a web-server cloud app does not directly benefit from a security cloud app because these VMs cannot interact with each other.

4.3.2 Low-level System Apps

As noted above, standalone apps do not interact with each other. Our vision of cloud app markets includes apps that can interact with client VMs to provide system-level services. Such *low-level system apps* contain specialized software to be used on other client VMs (or other cloud apps), and actively interact with the CPU, memory and operating system of client VMs. For example, a checkpointing app will read the CPU and memory state of a client VM to generate a snapshot of that VM.

Low-level system apps can empower clients by providing them with powerful security tools. On contemporary cloud platforms, clients that wish to protect their work VMs from malicious software largely resort to in-VM tools, such as anti-virus software. While such an approach can potentially detect malicious software that operates at the user level, attackers are increasingly exploiting kernel-level vulnerabilities to install rootkits, which in turn allow them to remain stealthy and retain long-term control over infected machines.

To counter kernel-level malware, researchers have proposed a number of solutions (*e.g.,* [92–94, 97]) inspired by virtual machine introspection (VMI). In such solutions, the virtual machine monitor acts as the trusted computing base, and helps maintain the integrity of VMs that it executes. The security policy enforcer (*e.g.,* a rootkit detector) itself operates within a privileged VM (*e.g.,* the management VM), and checks the state of a target VM. To enable security enforcement, the virtual machine monitor fetches entities from the target VM (*e.g.,* memory pages containing code or data) for inspection by the security policy enforcer. This architecture protects the enforcement engine from being tampered by malicious target VMs. Unfortunately, deploying VMI-based solutions on contemporary cloud infrastructures requires cooperation from the cloud providers, because VMI tools perform privileged operations on client VMs.

In our model, VMI-based intrusion detection tools can be implemented as low-level system apps. To use such an app, a client must be able to download and instantiate the app, and permit the app to inspect the memory of its work VMs to detect attacks. In Section 4.4, we describe how our cloud app model permits clients to assign specific privileges to its cloud apps, such as mapping memory pages of its work VMs into the cloud app's address space. Depending on their functionality, low-level system apps can either require two-way read/write access or one-way read access to the client's work VM state. For example, a rootkit detection app may send a request to the client VM to access to a particular page, and the client VM would respond by giving it suitable access. In response, the rootkit detector may modify the contents of the page (*e.g.,* to remove the rootkit). On the other hand, a checkpointing app or a forensic analysis app would only require read-only access to client VM memory. Figure 4.2(a) and Figure 4.2(b) illustrate both kinds of low-level system apps.

4.3.3 I/O Interceptors

An I/O interceptor is a cloud app that sits on the I/O path (either network or storage or both) of the client's work VMs. To use an I/O interception cloud app, clients set up the I/O path so as to direct the output of their work VM.

Such interceptors are already in popular use on many enterprise networks, as middleboxes, although not as cloud apps. For example, enterprises typically deploy a number of network security tools, such as firewalls, intrusion detection systems, and intrusion prevention systems,

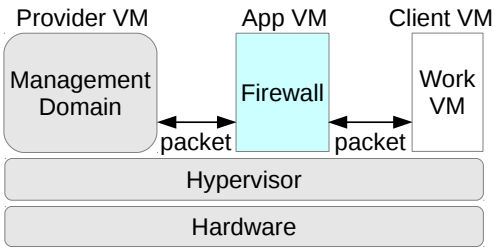


Figure 4.3: An I/O interceptor.

at the perimeter of networks to defend their in-house networks from attacks such as intrusions, denial-of-service (DoS), spam, and phishing, etc. While such middleboxes are relatively easy to deploy on networks controlled by the enterprise, this task becomes significantly more complex if the enterprise shifts its operations to public clouds. On contemporary cloud platforms, the enterprise must rely on the cloud provider to implement similar services on its work VMs.

I/O interceptor cloud apps provide enterprises the freedom to implement these services without having to rely on cloud providers. The enterprise simply downloads the network security cloud app, instantiates it, and configures it to receive both incoming and outgoing network flows to its work VMs. This would allow the enterprise to define and enforce arbitrary security policies flexibly, without relying on the cloud provider. A key requirement to support such apps is that clients must have the flexibility to create custom I/O channels.

4.3.4 Bundled VM Apps

It is often beneficial to combine the functionality of multiple cloud apps, and obtain a bundled cloud app that composes the functionality of its constituent cloud apps (Figure 4.4). Akin to the `pipe` primitive in operating systems, where the output of one command is input to another, in a bundled cloud app, the output of one cloud app is input to another. Individual cloud apps inside a bundle could be standalone, I/O interceptor or low-level systems apps, or themselves bundled apps.

Bundled cloud apps (Figure 4.4) are an ideal strategy for implementing and composing services. For example, a client interested in network security apps, such as intrusion detection systems and firewalls, downloads a network security app bundle. Upon instantiation of the bundle, all the apps inside it will create a chain of services. In Figure 4.4, for example, the client VM's packet will traverse the firewall, and then the network IDS, providing multiple

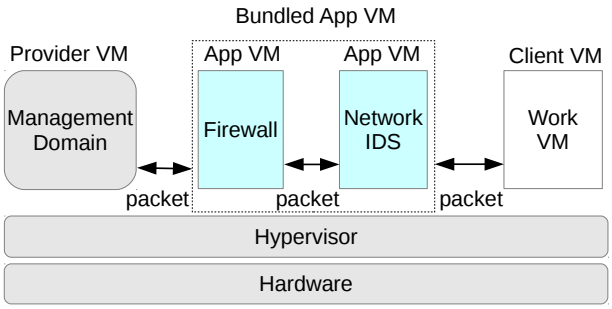


Figure 4.4: A bundled app VM along with a custom I/O channel connecting the client's work VM with the app VMs. FE stands for frontend, BE for backend, ND for network driver, and NIC for network interface card.

layers of security. To realize the bundled cloud app model, clients must fulfill the requirements of individual apps in the bundle. For example, a bundle consisting of low-level system and I/O interceptor apps will require both necessary privileges and custom communication and I/O channels to perform its task.

4.4 Design Considerations

In this section, we describe the components of a cloud app market ecosystem, and present a number of ways to design these components. The next section describes our prototype implementation, in which we have incorporated some of these designs.

4.4.1 Trustworthy Launch of Cloud Apps

The workflow of an app begins with the client downloading it from a cloud app market, configuring and starting the app. To ensure trustworthy operation, the client must have the ability to verify that the app booted correctly, with the same software stack as promised when the app was downloaded. This requirement is particularly important because clients will typically download cloud apps from the cloud market based upon the advertised functionality of the app. Because the cloud app will likely interact with the client's work VMs, it is important for the client to validate that the operating system and applications running atop the cloud app are as advertised by its creator. Such validation will establish trustworthy launch of cloud apps.

We expect that validation of trustworthy cloud app launch can be performed using attestation protocols developed by the trusted computing community [41]. This requires the cloud provider to offer hardware equipped with a trusted platform module (TPM) chip, and possibly

virtualize this hardware [98]. Attestation protocols "measure" software that is started on the system (usually by simply hashing the software), and store these measurements securely within registers of the TPM chip. A client can verify trustworthy launch by verifying the measurement received from the platform using the expected measurements of the cloud app.

4.4.2 Privilege Model

A number of useful VM-based services that could be deployed as cloud apps cannot be implemented on contemporary cloud platforms because they require additional privileges. For example, a VMI-based malware detector (a low-level system app) requires the ability to inspect the state of another VM, and therefore map its memory contents and CPU state. On contemporary cloud platforms, such applications can only be deployed with the cooperation of the cloud provider.

In our model, clients must have the ability to allow apps that they download from app markets to inspect/modify the state of their own VMs. Accomplishing this requires a careful examination of the privileges that cloud apps would need. In particular, a privilege model must satisfy two criteria:

(**1**) *It must be fine-grained*. Clients must be able to specify that a cloud app be given specific privileges (over its memory pages, or I/O channels) over client VMs.

(**2**) *It must prevent information leakage*. Cloud apps are expected to interact with client VMs, which may store and manipulate sensitive data. Low-level system apps and I/O interceptors can potentially access this information. Apps must not leak sensitive data to their publishers or other malicious third parties.

A privilege model that satisfies these criteria would allow clients to disallow apps from accessing the network, memory or other locations that store sensitive data, thereby preventing it from sending any sensitive data that it may read from the client's VMs. It may be possible to implement such defenses using a number of *ad hoc* hypervisor-specific techniques. For example, on Xen, such control can be implemented using Xen grant tables. A VM that wishes to communicate with the outside world shares pages with the management VM to pass data to hardware. Xen grant tables can be used to specify that a particular VM should never be able to establish shared pages with the management VM, hence restricting I/O.

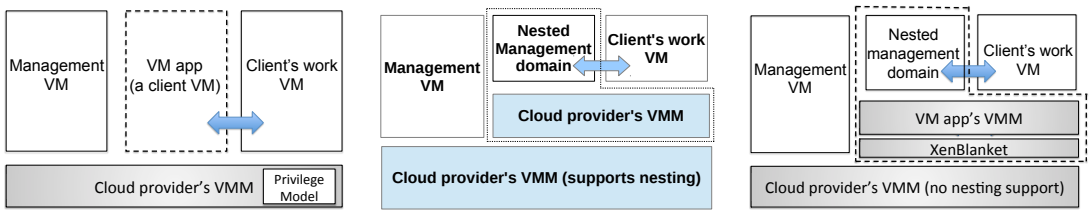
Attributes	Values
memory	kernel | user
network	on path | off path
location	must colocate | might colocate
description	Functionality of the cloud app
version	version of the cloud app

Figure 4.5: Attributes of a manifest file

Instead of relying on such *ad hoc* methods, we instead define a *privilege model* that must be implemented on any platform that supports cloud apps, leaving the specifics of the implementation to what works best on each platform. As already discussed, our model of a cloud app allows low-level system apps access to the CPU and memory state of a client's VMs. In our privilege model, apps that wish to obtain such access must specify their requirements up front as a set of permissions, akin to the concept of permissions currently used on mobile apps in certain platforms (*e.g.,* Android and Windows Phone). These permissions are specified in the app's manifest. Figure 4.5 describes the attributes (and the set of permissible values for these attributes) that are currently supported in our prototype implementation. We assume that as in mobile app markets, specifying permissions in a manifest allows the client to determine whether he wishes to use the app on his VMs.

Once approved by the client, the manifest is used to enforce the set of permissions requested by the app—Section 4.4.3 describes the design space of hypervisor-level modifications necessary to enforce such permissions. Our permission model also allows clients to dynamically revoke permissions that have been granted to cloud apps. In addition to allowing apps access to CPU and memory state, our app model also allows the client to specify how I/O interceptors must reside on the inbound or outbound I/O path of a client VM. We describe the policies used to do so in Section 4.4.4.

By design, our permission model combined with the policies used to govern I/O interception give a client fine-grained control over app behavior. We also believe that the permission model allows clients to determine how apps access sensitive information from their work VMs. Although careful use of permissions can regulate information flow from work VMs to apps to a certain extent, permissions on their own are not powerful enough to provide detailed, low-level information-flow controls.



(a) Hypervisor modification *(b) Nested virtualization* *(c) Paravirtualization*

Figure 4.6: The design space of implementation options for low-level system apps. In each figure, the components of the cloud app are demarcated using dashed lines.

The pemission model itself cannot directly provide low-level information-flow tracking (*e.g.,* as available via TaintDroid [99] for Android). As a result, it may be difficult to completely eliminate the risk of information leakage. For instance, Bugiel *et al.* [100] show that some cloud VMs store sensitive client data and do not erase it even when clients release the VM. To eliminate such threats, cloud infrastructure may have to be enhanced in the future to offer image-cleaning services and control mechanisms, as proposed by Mirage [101].

4.4.3 Hypervisor-level Support

Low-level system apps must have the ability to perform privileged system tasks on client VMs. Stock virtual machine monitors do not grant user domains such privileges. We consider three design options in the following subsections (illustrated in Figure 4.6), which require various degrees of modifications to the hypervisor. We evaluate each design with respect to two metrics:

(**1**) *Performance.* What is the runtime performance overhead imposed by the design? Note that cloud apps may intrinsically reduce the performance of client VMs, *e.g.,* the use of an I/O interceptor app will necessarily reduce the throughput of a network-bound client VM because of the additional network element introduced in the I/O path. In evaluating performance, we ignore this intrinsic overhead. Rather, we focus on the overhead introduced by the implementation technique used to build cloud apps.

(**2**) *Deployability.* Does the design require invasive changes to hypervisors, as deployed by cloud providers today?

Option 1: Hypervisor Modifications

Perhaps the most straightforward approach to realizing low-level system apps is to modify the hypervisor (Figure 4.6(a)). These modifications will primarily address the way in which

the hypervisor assigns privileges to client VMs. On commodity hypervisors, the management domain runs at the highest privilege level, and orchestrates all communication between VMs running as user domains. In this model, communication between VMs belonging to a single client goes through the management VM.

To support cloud apps, the hypervisor can be modified with a different privilege model, which allows apps belonging to a particular client to have specific privileges to over other VMs belonging to the same client. This would allow a low-level system app to map the memory pages belonging to the client's VMs. We expose these changes to the client via a set of hypercalls that allow an app to perform privileged operations on a client VM. These hypercalls allow a cloud app to read the register file, or map the user-space or kernel-space memory pages of the client's VM into its own address space. We describe our implementation of this design option, and the hypercalls that we added in Section 4.5.1.

We expect the runtime performance impact of this approach to be minimal. The performance overhead will likely stem from the impact of executing additional checks within the hypervisor to implement the privilege model. Cloud apps and client VMs continue to execute directly atop the modified hypervisor, and their performance will be comparable to VMs executing atop an unmodified hypervisor. However, this design option requires invasive changes to the hypervisor, which a number of cloud providers may be reluctant to deploy.

Option 2: Nested Virtualization

Hardware support for virtualization [102], coupled with software-level implementation tricks, have made low-overhead nested virtualization a reality [103]. Nested virtualization allows clients to execute a full-fledged nested hypervisor, also called an L1 hypervisor, atop a base hypervisor, called the L0 hypervisor, that supports nesting.

In this design option, a cloud app will be distributed as a L1 hypervisor together with a management VM that implements the advertised functionality of the app (Figure 4.6(b)). Clients can execute their work VMs atop the L1 hypervisor, which in turn has its own management VM. Thus, the app can implement its privileged services within the management VM of the L1 hypervisor. Only minimal changes are needed to the cloud provider's L0 hypervisor (described below).

With nesting, the L0 hypervisor executes at the highest processor privilege level and is the only software layer authorized to perform privileged operations, while the L1 hypervisor and client VMs execute with lower privileges. On the Intel x86 platform, every trap goes to the L0 hypervisor, which then either handles the trap or forwards it to the right layer. Thus, for instance, any operations by the app to map memory pages of the client's VM will trap to L0. The L0 hypervisor is modified to enforce the permissions in the privilege model; thus, it reads app manifest files, and enforces the permissions when it receives traps to perform sensitive operations.

We expect the runtime performance impact of this approach to be higher than in the case of direct hypervisor modifications. However, this option has the attractive advantage of only requiring minimal changes to the cloud provider's hypervisor. Our prototype also implements this design option.

Option 3: Paravirtualization

The third design option is a variant of the nested virtualized-based design described above. If the cloud provider's platform is based upon a hypervisor that does not have hardware support for efficient nested virtualization, it may still be possible to achieve many of the same benefits using paravirtualization. In particular, the XenBlanket project [104] demonstrates that it is possible to achieve nested virtualization atop a stock Xen hypervisor (*e.g.,* as deployed on Amazon EC2) that does not support virtualization. XenBlanket achieves this by building a thin software layer (a "blanket") that emulates privileged operations for a nested VMM executing atop the base Xen hypervisor (Figure 4.6(c)). Cloud apps based upon this design option will resemble those developed for the previous design option. The principal difference is that the software stack of these cloud apps and the client VMs must use a paravirtualized operating system.

We have not implemented this design option, but evaluate its merits using the XenBlanket implementation. XenBlanket was primarily developed to allow nested virtualization atop commodity cloud platforms. Although the reported overhead of the blanket layer is acceptable, the overheads of a cloud app implemented using this approach will likely be higher than if support for nested virtualization was available. Among our design options, we therefore expect

this approach to have the highest runtime overheads. However, the primary advantage of this approach is that cloud apps can be deployed today over stock services such as Amazon EC2.

Hybrid Designs

It may be possible to create a hybrid design for cloud apps that combines the designs discussed above. For example, suppose that a cloud provider decides to incorporate modifications to allow clients to plumb their I/O paths, in effect allowing clients to place middlebox cloud apps between their work VMs and the platform's management VM. However, the cloud provider may be reluctant to include VMM modifications that allow cloud apps to map memory pages of client VMs. In such cases, it is still possible to realize cloud apps such as rootkit detectors using nested virtualization using either the nested or paravirtualized designs.

4.4.4 Plumbing I/O

I/O interceptors need to have the ability to interpose on the traffic to and from the client VMs that they service. We rely on emerging technologies based on software-defined networking (SDN) to perform such interposition, and allow clients to plumb the I/O paths to and from their VMs.

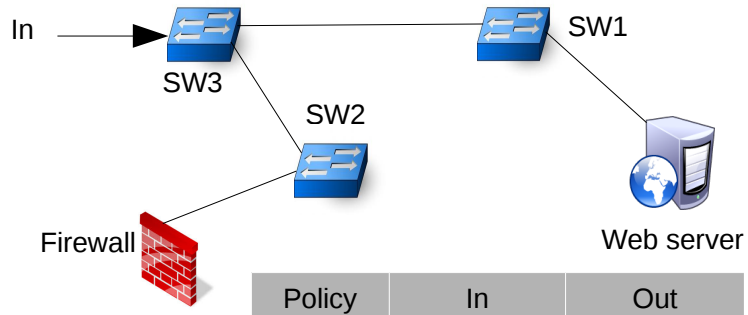
Plumbing Network I/O

Like recent work on using SDN to implement network middleboxes [105–107], we represent properties of network flows using *policy classes*. A policy class is a quintuple: <*source IP address, destination IP address, source port number, destination port number, protocol type*>. Clients use policy classes to specify *I/O policies*, which are rules of the form $C \rightarrow [M_1, M_2, \ldots, M_n]$, where C is a *policy class* and M_1, M_2, …, M_n, is a sequence of cloud apps. The semantics of this rule is that traffic that matches the policy class should traverse the sequence of cloud apps. Because each M_i is itself a VM, they can reside on separate physical machines. Regardless, traffic to and from client VMs must follow the I/O policies for the policy class that they match.

To enforce traffic flow compliant to I/O policies, we rely on the cloud provider's SDN controller. The cloud infrastructure's SDN controller tracks the physical placement of each of the middleboxes M_i. The SDN controller (which is based on the OpenFlow standard), uses I/O

Policy Class	In	Out
C	In	SW2
C	SW2	SW1

Policy Class	In	Out
C	SW3	Web Server



Policy Class	In	Out
C	SW3	Firewall
C	Firewall	SW3

Policy chain: Firewall → Web server

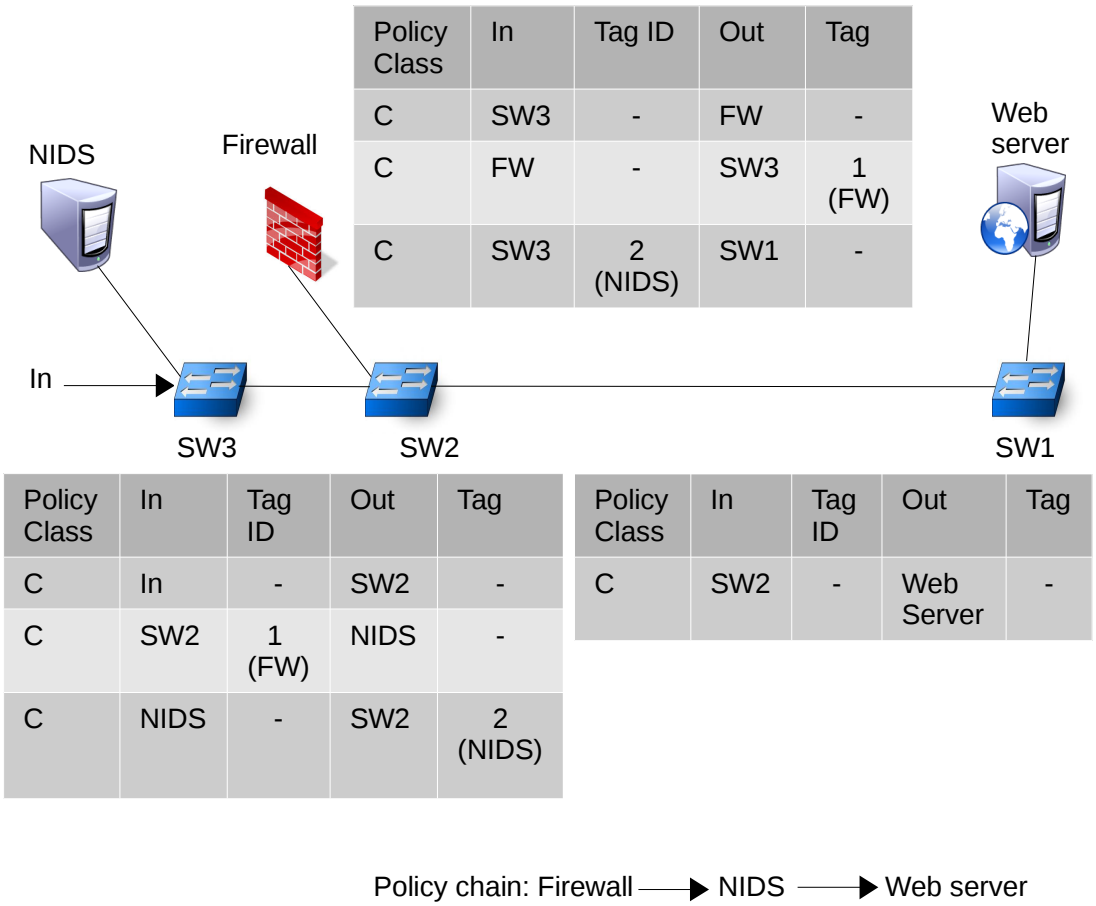
Figure 4.7: A firewall app deployed as a network middlebox for a web server

policies specified by the client to configure the software switches on each physical platform that executes a cloud app or a client VM that is affected by the I/O policy. Upon receiving a network packet, each switch in the middlebox app sequence identifies the apps that the packet has traversed. This helps the switch to determine the next hop in its forwarding table.

To make this discussion concrete, consider the network topology shown in Figure 4.7. In this example, a client specifies that each network request to the web server VM must go through the firewall VM. The policy class *C* can be specified as <*srcIP = external prefixes, dstIP = IP of the web server, srcPort = any, dstPort = 80, protocol = TCP*>. When SW3 receives a packet, it needs to know if the packet has traversed the firewall to decide the next hop to forward the packet. If the packet has not passed through the firewall, SW3 forwards the packet to SW2 (hosting the firewall). Otherwise, SW3 forwards the packet to SW1 (hosting the web server).

As discussed in prior work [105], there are two cases that affect building forwarding rules:

(**1**) There are no loops in the sequence of switches. In this case, the switches can use incoming interfaces to build forwarding rules. In Figure 4.7, SW3 forwards packets arriving on "In" to



Policy Class	In	Tag ID	Out	Tag
C	SW3	-	FW	-
C	FW	-	SW3	1 (FW)
C	SW3	2 (NIDS)	SW1	-

Policy Class	In	Tag ID	Out	Tag
C	In	-	SW2	-
C	SW2	1 (FW)	NIDS	-
C	NIDS	-	SW2	2 (NIDS)

Policy Class	In	Tag ID	Out	Tag
C	SW2	-	Web Server	-

Figure 4.8: A sequence of network middleboxes

SW2, and packets arriving on SW2 to SW1.

(**2**) There is a loop in the sequence of switches. This situation might cause an incorrect forwarding decision. An I/O policy Firewall→NIDS→Webserver as depicted in Figure 4.8 can confuse SW2 when forwarding packets. When SW2 receives packets from SW3, it needs to decide to whether forward the packets to the firewall or to the web server. We address this issue by tagging packets. In particular, after a packet traverses a middlebox, the switch hosting that middlebox tags the packet using VLAN tags. Each middlebox app is assigned an unique tag id. In Figure 4.8, SW2 uses the tag assigned to the NIDS to know if a packet it receives from SW3 has traversed the NIDS.

Plumbing Disk I/O

In our app model, I/O interceptors that work on storage devices work by intercepting storage streams of the client's VM to perform various storage related tasks such as storage deduplication and file integrity checking.

One approach to support storage apps is to modify the hypervisor. Commodity hypervisors handle I/O operations from guest VMs by forwarding them to an I/O emulator. The emulator can either be the management VM as in the Xen hypervisor or a user space process as in the KVM hypervisor. The hypervisor can be modified to forward I/O streams of the client's VM to storage apps before sending them to the I/O emulator. However, this solution, only works if storage apps run on the same physical machine with the client's VM.

We therefore process storage I/O streams as network traffic. In particular, we propose to use network file system (NFS) services to implement storage apps. We run an NFS client service inside the client's VM and NFS server services inside storage apps. All read/write operations from the client's VM take place by invoking the NFS services. The NFS client service communicates with the NFS server service to fulfill the operations. With this design, the storage I/O can be intercepted and manipulated in the same manner as network I/O.

4.4.5 Composing App Functionality

In a number of cases, it may be necessary to compose, or "chain," the effects of multiple cloud apps. Our design allows for such chaining by requiring the developer to specify a *composition policy* (also called a *chaining rule*), of the form $C \rightarrow [M_1, M_2, \ldots, M_n]$, where C is the identity of a client's VM and $M_1, M_2, \ldots, M_n$ is a sequence of cloud apps that either have system-level privileges over C or are I/O interceptors. The chaining policy indicates that the memory pages of the client's VM that are required by M_n should be processed by $M_1, M_2, \ldots, M_{n-1}$, in order, before being mapped by M_n. At any point during this chain, an app can specify that its results must not be passed to the next app, thereby breaking the chain.

We explain this in further detail with an example of a memory checkpointing app and a rootkit detection app as an example. The memory checkpointing app reads memory pages of a client's VM to create a memory snapshot. The rootkit detector inspects memory pages of the client's VM to detect malicious code. One benefit of chaining these two apps is that the

client can make the memory checkpointing app only take a snapshot of memory pages that are ensured to be benign by the rookit detector. The hypervisor receives a request from the memory checkpointing app to read a memory page of the client's VM. In turn, the hypervisor consults the chaining policy, which specifies *client's VM → Rootkit detector → Memory checkpointing app*. Thus, the hypervisor invokes the rootkit detector to scan the memory page, and indicate whether the page is malicious or not. If malicious, the rootkit detector breaks the chain, and the checkpoint is not created. Otherwise, the benign memory page is forwarded to the checkpointing app, which creates the VM image.

While the above example illustrated composition using two low-level system apps, the concept extends in a straightforward fashion to I/O interceptors. In this case, the composition policy is mapped in a natural way to an I/O policy, which is then enforced by the SDN controller by configuring virtual switches on each physical host.

4.5 Implementation

We implemented a prototype of the infrastructure to support cloud apps atop KVM version 3.12.8. The implementation adds roughly 600 LOC to the KVM kernel module. In addition, our implementation also comprises a user-space module consisting of rougly 500 LOC of Java code. We describe the components of our prototype implementation in the rest of this section.

4.5.1 Hypervisor-level Support

For low-level system service apps, our implementation supports the two design options discussed in Section 4.4.3 and Section 4.4.3.

The lifecycle of an app begins when the user inspects its manifest and installs the app in his domain. At this point, our infrastructure parses and reads the manifest. This functionality is implemented as a user-space module in the hypervisor's management domain. This user-space module communicates with the hypervisor to set up the privileges requested by the app. It does so using two new ioctl commands that we introduced in KVM, shown in Figure 4.9. Both design option 1 and option 2 incorporate this user-space module and the ioctl interface to control permissions.

Once an app has been installed, in option 1 it interacts with the hypervisor via hypercalls.

New Hypercalls (Reads only; we also added similar calls to write)
• vaddr_Read_VM_Memory(id_of_client_vm, virt_addr, bytes, buf) **Description:** Read memory of a client's VM starting at virtual address virt_addr.
• phyaddr_Read_VM_Memory(id_of_client_vm, phy_addr, bytes, buf) **Description:** Read memory of a client's VM starting at physical address phy_addr
• PFN_Read_VM_Memory(id_of_client_vm, page_frame_num, buf) **Description:** Read a page frame of a client's VM
• register_event(id_of_client_vm, virt_or_phy, addr) **Description:** Register to receive a trap from the hypervisor when a client VM address is modified.
• read_Register(id_of_client_vm, register_id, buf) **Description:** Read a register's content of a client's VM

IOCTL Commands
• kvm_ioctl_set_register_privilege(VM1_Name, VM2_Name) **Description:** Grant VM1 a permission to have access to VM2's registers.
• kvm_ioctl_set_memory_privilege(VM1_Name, VM2_Name, Memory_Type) **Description:** Grant VM1 a permission to have access to VM2's memory. Memory_Type indicates user space memory or kernel space memory

Figure 4.9: Description of hypercalls and ioctl commands added to KVM to support Option 1.

We modified KVM to add the hypercalls shown in Figure 4.9. Cloud apps use the hypercalls to map the memory and register state of client VMs into their address spaces. To map a physical page belonging to a client's VM into a virtual address space of a cloud app, a hypercall handler translates the client VM's physical address in memory to a virtual address via the extended page tables (EPTs) maintained by KVM. Once an app has mapped a client VM's pages into memory, it can modify the client VM's page tables and register with the hypervisor to receive traps on events of interest (`register_event`). For example, it can modify the page tables to receive a trap each time a page of the client VM is executed. This enables the development of cloud apps that can check code pages as they execute (*e.g.,* as was done in the Patagonix system [97]).

It is straightforward to support option 2 in KVM because it supports nested virtualization. The above hypercalls can be supported within the L1 hypervisor (which is part of the cloud app), and no changes are necessary on the L0 hypervisor (which is controlled by the cloud provider).

4.5.2 Support for I/O Plumbing

To intercept I/O and allow clients to plumb I/O paths, our prototype users a daemon within the management domain to process client I/O and composition policies. An I/O policy can be thought of as a state machine that the packet must traverse from the start state to the finish state. Thus, our goal is to tag each packet with the "state" that it currently is in.

We assume that the cloud provider supports a centralized controller, which stores the identities of the switches hosting the cloud app/client VM. This controller accepts the client's I/O policy and distributes the corresponding state machine to all the software switches. When a packet is processed by the cloud app running on a physical host, the software switch that executes within the management domain of that host tags the packet with the identity of the cloud app that just processed the packet. Once the packet reaches the "final state," of the state machine, it is forwarded to the client VM.

Our implementation incorporates the above tagging and forwarding support within the software switches using the Floodlight [108] Openflow controller. We communicate with Floodlight's static flow pusher via its REST API [109] to insert flows. We use the *mod_vlan_id* and *strip_vlan* actions to respectively tag and untag packets.

4.6 Evaluation

The main goal of our experimental evaluation is to understand the performance overhead introduced by the cloud app model. We do not focus on the effectiveness of the apps (*e.g.,* the effectiveness of a rootkit detection app at identifying real rootkits) because we believe that is orthogonal to the goal of this work. Rather, we show how apps that have been proposed in prior work can be implemented and deployed easily as cloud apps, and measure their performance overheads.

We conducted experiments for both low-level system apps and I/O interceptors. Our setup consisted of three physical machines in the same physical LAN. The throughput between each machine is 100 Mbps. The first machine is equipped with an Intel Core i7 CPU, 8GB of memory and an Intel Gigabit NIC. The second machine has an Intel Xeon(R) CPU, 24 GB of memory and two NetXtreme BCM5722 Gigabit Ethernet cards. We dedicate the third machine as the measurement host; this machine has an Intel Core i5 CPU and 4 GB of memory. The first

two machines run Ubuntu version 12.04 with kernel version 12.08. The measurement host runs CentOS with kernel version 2.6.32. Guest VMs run ubuntu version 12.04 with kernel version 3.5. Both L0 and L1 uses KVM kernel module version 12.08 as their hypervisors.

4.6.1 Low-level System Apps

In this section, we evaluate the impact of running low-level system apps. In particular, we implemented two system service apps that map memory pages of a client's VM for detecting rootkits and checkpointing memory. We use SPEC CINT2006 benchmark to measure the utilization of the CPU and memory of VMs. We run the benchmark in three setups: on the host OS, on the single-level guest and on the nested guest.

Rootkit Detection

We implemented a rootkit detection app that implements the same functionality as Patagonix [97], and detects rookits by checking code integrity. It ensures that all code executing on the client's VM belongs to a whitelist. The KVM hypervisor traps any executing code pages of the client's VM and allows the rootkit detector to map those pages for checking. We execute the benchmark inside the client's VM to measure the impact of the rootkit detector on the client's VM.

We implemented this app using both option 1 (hypervisor modifications) and option 2 (nesting). We compared the performance of both options to the traditional approach of running the service as a daemon within the management domain, as would be done on contemporary cloud platforms that do not support a cloud app model. This traditional approach serves as our baseline for performance measurements.

Figure 4.10 shows the results under the three setups. We observed a slowdown of 15.3% in the nested virtualization setup and an overhead of 8.5% in the hypervisor modification setup. As explained in prior work [103], most of the overhead in the nested case is caused by VM exits from the guests to the L0 hypervisor due to privileged instructions. Table 4.1 shows the CPU utilization in each setup.

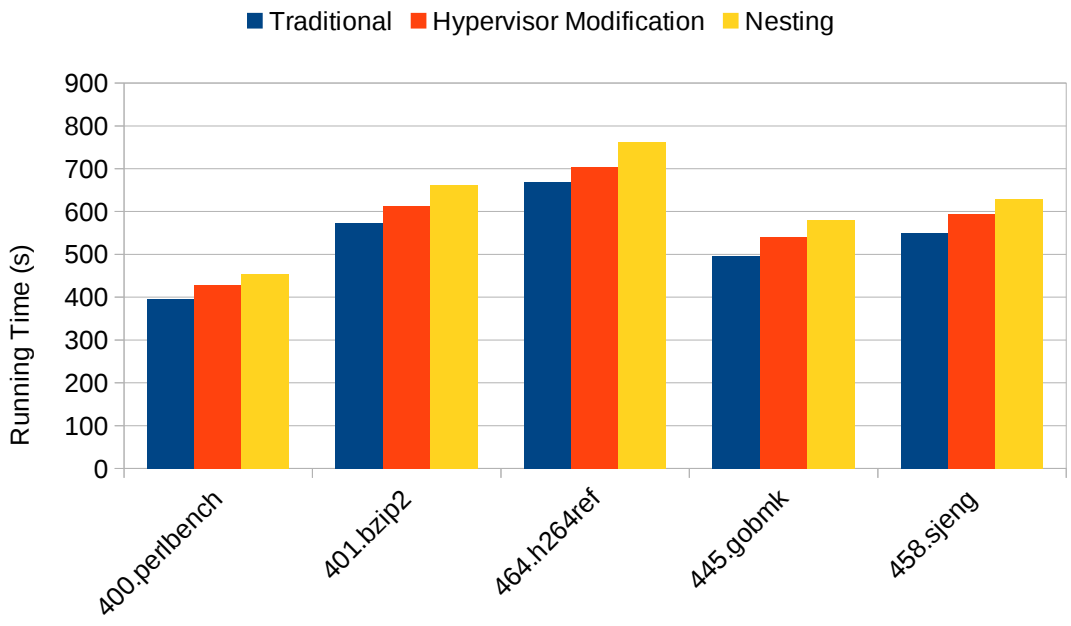


Figure 4.10: Running Time of SPEC CINT2006 under the three setups

Setup	CPU%
Traditional	99.55
Hypervisor Modification	99.83
Nested Virtualization	~100

Table 4.1: CPU utilization under the three setups

Memory Checkpointing

Our memory checkpointing app maps memory pages of the client's VM and checkpoints them for various purposes such as debugging or crash recovery. We evaluated this app by checkpointing three client's VMs with different memory footprints: 512 MB, 1024 MB and 2048 MB.

Figure 4.11 presents the results of our experiments, comparing the costs of implementing the memory checkpointing service within a cloud app (both design options) with the traditional setup. Our results show that the overhead of implementing the checkpointing service within a cloud app using hypervisor modification is 7.95%. The slowdown caused by implementing the same service using nested virtualization is 12.38%.

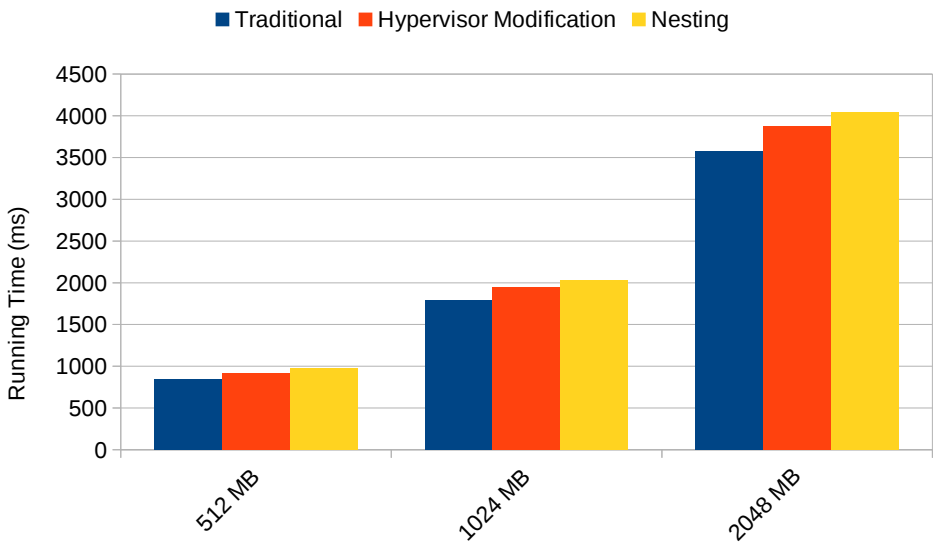


Figure 4.11: Running Time of a memory checkpointing service under three setups

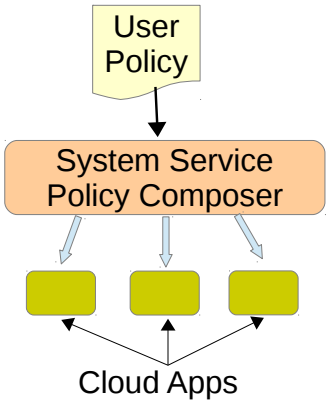


Figure 4.12: The SAMEHOST and DIFFHOST configurations. We only show the inbound network path. The outbound path is symmetric.

4.6.2 I/O Interceptors

To evaluate the performance of I/O interceptors, we compare the overhead of implementing the service as a cloud app, which runs as a separate VM, with the traditional approach of implementing the I/O interception service as a daemon within the management VM. Each cloud app in our experiment is assigned one virtual CPU and 2GB of memory. We performed each experiment under two settings, a SAMEHOST setting where the cloud app is co-located with the client VM that it services, and and a DIFFHOST setting, where the cloud app and the client VM are on different physical hosts, as shown in Figure 4.12.

Setup	Throughput (Mbps)	RTT (ms)
SAMEHOST configuration		
Traditional	93.87±0.560	0.713±0.043
Cloud App	90.64±0.485 (3.44%)	1.631±0.046 (2.29×)
DIFFHOST configuration		
Traditional	90.12±0.763	1.68±0.056
Cloud App	86.44±0.518 (4.08%)	2.281±0.186 (1.36×)

Table 4.2: Baseline overhead of networked services in the cloud app model as compared to the traditional setup

Baseline Overhead

Before measuring the overhead of an I/O interceptor we measure the overhead when a cloud app does no additional processing on the packets that it receives. This overhead is the best performance achievable for an I/O interceptor. The overhead of any additional services that the interceptor provides must be compared against this baseline overhead.

The measurement host is used to transmit and receive network packets to the client's VM. We measured the network throughput to the client's VM using iperf3 [110], and used ping to measure the round-trip time (RTT) of network traffic to and from the client's VM. Our results report averages over ten executions along with the standard deviations. Table 4.2 presents the results of our experiment. If the cloud app is co-located with the client's VM, the throughput drops to 3.44% compared to the traditional setup. The RTT overhead also increases 2.29× as a result of having to traverse the cloud app on the path to the client's VM. When the cloud app and the client's VM are on different hosts, the RTT overhead increases 1.36× and the throughput reduces to 4.08% compared to the traditional setup.

Network Intrusion Detection

In the cloud app model, clients can deploy and configure NIDS apps as network middleboxes. As an example, we used Snort [111] to build an NIDS app. Snort uses libpcap [112] to capture network traffic. Our setup uses the Stream5 preprocessor that performs TCP reassembly and handles both TCP and UDP sessions. We used the latest snapshot of signatures available from the Snort website. Table 4.3 shows the results of our experiments.

Setup	Throughput (Mbps)	RTT (ms)
SAMEHOST configuration		
Traditional	91.58±0.980	0.980±0.028
Cloud App	87.15±0.650 (4.84%)	1.871±0.031 (1.91×)
DIFFHOST configuration		
Traditional	88.36±0.897	1.92±0.032
Cloud App	83.37±0.551 (5.65%)	2.58±0.043 (1.34×)

Table 4.3: Overhead of an NIDS service in the cloud app model as compared to the traditional setup

4.6.3 Storage Services

We implemented two storage I/O interceptors that intercept storage streams from the client's VM to respectively detect intrusion and check the integrity of the file system. We conducted the experiments by writing a file of size 500MB from the client's VM. We measured the throughput of the write operarations and the total time to write the file. We compared the results against the traditional setup where the storage service is implemented within the host OS. As before, we compared the results against a baseline overhead when a storage app does no additional processing on the storage streams it receives.

Storage Intrusion Detection System (SIDS)

We implemented the SIDS by running an NFS server service and using Snort as an intrusion detection system (IDS). We use the same configuration for Snort as described earlier, with additional rules for storage packets. Any write operations from the client's VM are sent to the NFS server service in form of network packets. The IDS discards any packets that violate its rules. The performance results are shown in Table 4.4.

Setup	Throughput (Mbps)	Time (s)
BASELINE configuration		
Traditional	89.1±0.178	45.15±0.43
Cloud App	84.7±0.367	47.22±0.27
SAMEHOST configuration		
Cloud App	86.3±0.4	46.34±0.39
DIFFHOST configuration		
Cloud App	82.13±0.284	48.5±1

Table 4.4: Storage Intrusion Detection Service.

State	Throughput (Mbps)
Disabled	88.7±0.294
Enabled	74±0.342

Table 4.5: File Integrity.

Mode	VM size (MB)	Running time (ms)
Traditional	512	3670.178
Bundled App	512	4352.831 (15.67%)
Traditional	1024	7096.947
Bundled App	1024	8452.464 (16.04%)

Table 4.6: Running time of a memory checkpointing app when running inside a bundled app.

File Integrity Checking

Our file integrity checking app checks the integrity of files of the client's VM at block level. We implemented this app by fetching inode information of the file that the client wants to check. We then collect the number of blocks of the file and hash them to form a whitelist. The app then can check the integrity of the file by re-hashing it and compare the results with the whitelist. Table 4.5 presents the evaluation results.

4.6.4 Composing Cloud Services

In this section we evaluate the impact of composing cloud apps. In particular, our experiments examine the composition of low-level system apps and I/O interceptors.

Composition of system-level service apps

In this experiment, we composed the memory checkpointing app with a rootkit detection app so that the checkpointed memory pages are ensured to be benign. Table 4.6 shows the results. The total overhead incurred by the bundled app is about 16%. This is because memory pages have to be checked by the rootkit detection app before being checkpointed by the memory checkpointing app.

We also composed a firewall app with a NIDS app to form a policy chain Firewall→NIDS→Client's VM. The firewall app contains a set of rules that include a list of IP addresses and open ports that are accepted. The two apps run on the same physical machine while the client's VM run on the other. The results shown in Table 4.7 indicate that the throughput drops to 9.3%. The RTT also increases 1.75× due to having two middleboxes in front of the client's VM.

Setup	Throughput (Mbps)	RTT (ms)
SAMEHOST configuration		
Traditional	88.36±0.301	1.964±0.0454
Bundled App	80.14±0.503 (9.3%)	3.432±0.051 (1.75×)

Table 4.7: Overhead of a bundled app as compared to the traditional setup

Finally, we composed storage I/O interceptors by chaining our SIDS app with a storage deduplication app. In Unix file systems, inodes hold the block numbers of disk blocks. Our storage deduplication app works by having inode pointers reference a single block rather than duplicate blocks. We use the MD5 checksum of the block's contents as its fingerprint.

We conducted the experiments by writing two files of size 500 MB from the client's VM. The first file is written two times while the second file is written one time. We measured the throughput of the write operations as well as the disk usage at the storage deduplication app either when both cloud apps are on the same host or on a different host with the client's VM. We compared the results against baseline overheads when storage apps do not perform additional processing on the storage streams they receive.

The results are shown in Table 4.8. We consider both the SAMEHOST case, where the bundled app and the client VM are on the same host, and the DIFFHOST case, where the bundle executes on one machine and the client VM on the other. We observe that the throughput reduces to 10% and the deduplication app saves approximately 4.3% of disk space.

4.6.5 Hybrid Apps

A hybrid app bundles an I/O interceptor with a system-level service. We evaluated an app that implements an application-level firewall (based on prior work [94]) by intercepting network packets from the client's VM. It then performs memory introspection of the client's VM to identify the process that is bound to the network connection. If the process belongs to a whitelist, then the app permits the network flow, otherwise it blocks the network connection. This app therefore provides both a system-level service and a networked service and must be co-locate with the client's VM.

Table 4.9 presents the results of our experimental evaluation that compares the implementation of this firewall as an app versus a traditional setting, where it is implemented as a daemon within the management VM. The app's throughput reduces by 2.93% when the firewall service

File Name	Throughput (Mbps)	Disk Usage(MB)
BASELINE configuration		
No File	NA	280
File 1 (500M)	89.1±0.328	780
File 1 (Copy)	89.9±0.247	1330
FIle 2(500M)	89±0.153	1380
SAMEHOST configuration		
No File	NA	280
File 1 (500M)	85.2±0.146	780
File 1 (Copy)	85.6±0.483	820
FIle 2(500M)	84.9±0.398	1320
DIFFHOST configuration		
No File	NA	280
File 1 (500M)	79.4±0.216	780
File 1 (Copy)	79.7±0.387	820
FIle 2(500M)	79.1±0.178	1320

Table 4.8: Composition of Storage based Service

Setup	Throughput (Mbps)	RTT (ms)
Traditional	92.16±0.528	0.925±0.021
Cloud App	89.46±0.441 (2.93%)	1.412±0.029 (1.53×)

Table 4.9: Overhead of the application-level firewall service in the cloud app model as compared to the traditional setup

is implemented as a cloud app. The RTT increases 1.53× due to having network packets going through the app.

4.7 Summary

In this chapter, we present the design and implementation of a rich model that allows cloud apps developed by third-parties to perform privileged tasks on a client's VMs. Clients can implement a variety of services, such as low-level system services, I/O interception, and even bundle several services into a single app. We discussed the infrastructure support needed to support cloud apps, explored various design options to implement cloud apps, demonstrated and evaluated the practicality of cloud apps using various examples. We believe that our work demonstrates the potential of and a practical way to realise the vision of security as a service, where security services are implemented as apps that can be paid for, downloaded and used by clients on their cloud-based VMs.

Chapter 5

Related Work

Secure systems based on Intel SGX. A number of recent projects have applied Intel SGX for trusted computation on cloud platforms. Microsoft's Haven [113] was the first project that leveraged Intel SGX to enable unmodified Windows binaries to run on Intel SGX-based cloud platforms. Haven allows an application to be linked with a runtime library OS variant of Windows 8 and loaded into an enclave. The confidentiality and integrity of this code and data is protected from the cloud provider. VC3 [114] is another effort to leverage SGX to provide security for enclaves that perform MapReduce-style computations. VC3 also recognized that enclave code with memory safety errors could pose a threat to confidentiality of client data, and proposed instrumenting client code with a form of control-flow integrity instrumentation.

SecureKeeper [115] leverages Intel SGX to keep ZooKeeper-style computations confidential. S-NFV [116] uses Intel SGX to address security issues of today's NetWork Function Virtualization (NFV) infrastructures by securely move the states of NFV applications in enclaves. SGX processors are also used to improve the performance of privacy preserving multi-party machine learning [117].

While SGX provides attractive hardware-based security guarantees, it places considerable burden on the enclave code programmer to ensure that computations executing within the enclave do not accidentally leak information. Similarly, vulnerabilities such as memory safety errors in enclave code can lead to exploits that leak confidential data. Moat [118] takes a first step towards this goal by statically analyzing x86 machine code to be loaded within the enclave to check for information-flow violations. `/CONFIDENTIAL` [55] extends the approach proposed in Moat, and provides enclave authors with a library that they can link their enclave code against. As long as code is linked against the `/CONFIDENTIAL` library, and sensitive data sources are identified, the library ensures that sensitive data does not accidentally leak from enclaves (a property

called information-release confinement). /CONFIDENTIAL achieves this goal by restricting enclave communication with external memory through a narrow interface.

The key difference between /CONFIDENTIAL (and also Moat and VC3) and EnGarde is in the threat model—in /CONFIDENTIAL, the client compiles his code against the the /CONFIDENTIAL library, but the cloud provider does not check that the code has been compiled against this library. Thus, /CONFIDENTIAL is developed for the benefit of the client. In contrast, EnGarde focuses on mutual trust and SLA compliance. With EnGarde, the cloud provider can check that the client has compiled his code against a library such as /CONFIDENTIAL. The cloud provider therefore obtains an assurance that the client's code is policy-compliant. However, he does not learn any further facts about the client's code, thereby protecting client confidentiality.

Intel SGX does not protect applications against side-channel attacks and EnGarde also does not attempt to eliminate this attack vector. Yuanzhong *et al.* [119] demonstrate that by exploiting the fact that page table management in SGX is under the control of the OS, a malicious OS can manipulate page tables and page faults to learn memory access patterns of an enclave and therefore can infer private information of that enclave. Similarly, AsynShock [44] controls page access permissions of a multi-threaded enclave-based application to exploit synchronization bugs that might lead to memory corruption or crashes. It offers a tool to widen the attack window in synchronization bugs by interrupting a thread by removing the read and execute permissions from enclave pages and then scheduling another thread whose execution causes synchronization bugs.

Finally, recent work on Ryoan [120] has leveraged the Intel SGX to build a sandbox for distributed applications. Like EnGarde, Ryoan also relies on NaCl [42] to enforce restrictions on code loaded inside an SGX sandbox, but does so for an entirely different purpose. While Ryoan uses NaCl to ensure that code loaded into an enclave only has restricted control transfers, EnGarde uses NaCl only for reliable disassembly.

Recognizing Functions in Binary Code. EnGarde assumes that client binaries are shipped with symbol-table information (binaries that do not contain this information are auto-rejected by EnGarde). This helps identify functions in binary code which might be needed by the policies on verifying the binaries. Recognizing functions in COTS programs which do not contain

debug information has become a growing interest in recent years. For instance, both supervised machine learning [121] and neural network-based algorithms [122] have been applied to recognize functions in stripped binary executables. However, these approaches are still in their infancy, and cannot guarantee 100% accuracy [122]. As these techniques develop and improve in their accuracy and performance, EnGarde can be enhanced to even consider stripped binaries as enclave code.

Instrumenting Code to Thwart Attacks. For years, various solutions have been proposed to defend against control flow hijack attacks due to software bugs. These include stack canaries to protect return addresses and other control data on the stack [123] uses stack canaries, and various forms of control-flow integrity protection (*e.g.,* [59, 124–126]) use binary rewriting to enforce CFI protection. Cloud providers may require clients to compile their code with such instrumentation. As we saw in this chapter, EnGarde can accommodate a variety of policy modules that check that enclave code has been instrumented as agreed-upon by the cloud provider and the client.

Ransomware Detection The emergence of ransomware in recent years has captured the attention of the security community. Kharraz et al. [127] examined the behavior of 15 ransomware families with a focus on their encryption mechanisms and financial incentives and proposed several defenses. HelDroid [128] detects Android ransomware using static and dynamic analysis including text analysis techniques to detect ransomware ransom notes and screen lockers. Recently, researchers [16–18] have focused on monitoring filesystem activities to track the real-time changes of user data to detect ransomware instead of directly inspecting the program making the changes. Therefore, these approaches make it more difficult for ransomware to evade detection.

Malware Detection Using Hardware Performance Counters Demme et al. [64] first demonstrated the feasibility of building malware detectors using HPCs. They collected microarchitectural traces of Android malware and Linux rootkits on ARM and Intel platforms respectively and applied machine learning classification algorithms to show that their approach is able to

detect new variants of known malware. There are several key differences between their approach and HRD. *First*, their approach categorizes malware by their families and attempts to detect each individual malware family. In contrast, we focus on detecting a malware behavior and specifically ransomware-like behaviors. As a result, our approach is more successful in classifying ransomware. *Second*, they focus on offline analysis of malware instead of online detection. This stems from the fact that they need to collect performance data of hundreds of events to produce multi-dimensional time series data but the processors they use implement only four performance counters (Intel's Nehalem) and six performance counters (ARM Cortex-A9 cores). They proposed a modification to existing processor architecture to support for online malware detection. HRD, on the other hand, is able to perform online ransomware detection given the existing processor architecture. Other works focused on instruction opcodes to detect malicious activities either by measuring opcode distribution [68, 81] or identifying sequence signatures of the opcodes [129, 130] or by comparing graphs of opcode sequences [131].

Tang et al. [65] leveraged HPCs to detect exploitation of IE and Adobe PDF Reader. They categorized malware exploits into multiple stages and applied machine learning to select the top events for each exploit stage. There is a distinction between their work and our approaches. *First*, they only focus on malware exploits of two programs while we focus on a broad family of ransomware. *Second*, instead of performing feature selection for hundreds of performance events, they shortlisted 19 events they think are useful for their system. While these shortlisted events are effective to detect malware exploits of the two programs, it is uncertain if we only need them to detect ransomware due to the difference between ransomware and other types of malware. [66] explored the use of ensemble learning to improve the performance of hardware based malware detector. They attempted to detect different malware families by using low-level hardware features including opcode-based features, memory reference patterns, and architectural events. Singh et al. [67] proposed to use HPCs to detect kernel-level rootkits. CFIMon [69] used performance counters to monitor control flow of programs to detect violation of control flow integrity.

Adversarial machine learning While machine learning systems have been successfully deployed in various domains including malware detection systems, machine learning is unfortunately susceptible to adversary attacks [132–139] . An adversary with the knowledge of some parameters of a machine learning model can manipulate training or testing samples to defeat the deployed machine learning model. For example, a spammer can cause a Bayesian filter to misclassify legitimate emails by including dictionary words in spam emails [139]. This dissertation does not consider adversarial activity in building classifiers to detect ransomware, techniques to protect classifiers from such adversarial attacks can be the subject of future research.

Customized services provided by current cloud platforms Prior work by virtualization and cloud vendors has attempted to implement some of the features that we have described in our cloud app model. In an effort to manage multi-tiered applications, VMware introduced the concept of a virtual applicance (vApp) [88]. A vApp is a container that contains one or more VMs. It allows resource control and management of VMs. In particular, VMWare vApp offers users different options to allocate IP addresses for VMs. IP addresses can be fixed, allocated by a DHCP server or transient (allocated at power-on and released at power-off). Also, entire vApps can be cloned, powered on and powered off. Users also have the capability to specify startup as well as shutdown order among VMs. For example, in a vApp that holds a web server VM and a database server VM, a user can require that the database server to start before and shutdown after the web server. VMWare vApp, however, provides no explicit feature for composing middleboxes among a group of VMs. To our knowledge, it also does not give clients an API to implement low-level system services as apps.

The Amazon virtual private cloud (Amazon VPC) [140] lets users define a virtual network for their virtual machines. Users can select IP address range, create subnets as well as configure network gateways. With Amazon VPC, users can specify simple middlebox policies that involve only one middlebox. For example, a user can deploy a firewall middlebox in her network by configuring the firewall VM to function as the gateway of her network. Complex middlebox policies that entail the traversal of network packets among a sequence of middleboxes are hard to achieve with Amazon VPC.

While both of these techniques implement support for grouping and controlling collections of VMs, to our knowledge, our work is the first to provide a unified framework for implementing system and network-level services as cloud apps, consider various design options, and develop the framework to support third-party cloud apps (*e.g.,* permissions, I/O policies, composition policies)

There is a nearly a decade of prior work on virtual machine introspection-based services, starting with the seminal work of Chen and Noble [141] and Garfinkel and Rosenblum [92]. This body of work (references too numerous to list here) has developed a number of innovative approaches to building security services using virtual machines. While these prior techniques have focused on building the capabilities and precision of the security tools, they have largely assumed that the tools themselves will be implemented in a separate virtual machine (*e.g.,* as a daemon in the management domain).

On traditional cloud platforms, implementing these techniques requires the cloud provider to deploy the services within the management domain *for each client*. In contrast, our work considers a baseline infrastructure that the cloud provider must offer (*e.g.,* hypervisor-level changes, modifications to software switches, and supporting the permission model) that would empower clients to independently deploy these services as apps on their own VMs. We laid some of the foundations for these infrastructural changes in our prior work on self-service cloud computing [142], where we developed the basic ideas to support low-level system service apps. This chapter significantly extends those ideas, and introduces the notions of and provides supporting infrastructure for I/O interceptors, app permissions and composition policies.

Chapter 6

Conclusion

In this dissertation, we have shown that recent hardware advances and rethinking the security model in today's cloud platforms provide substantial security benefits for clients on the cloud.

In chapter 2, we introduced EnGarde, an enclave inspection library that allows the cloud provider to verify the client's SGX-based enclave against a set of policies mutually agreed by the cloud provider and the client. EnGarde also preserves the security benefits offered by SGX. EnGarde achieves its goal by using SGX's hardware attestation and having an encrypted channel set up between the cloud provider and the client. We have demonstrated that EnGarde can effectively enforce three popular security policies for various real world applications.

In chapter 3, we presented an alternate solution to detecting ransomware. We rely on HPCs to collect hardware events and apply machine learning to select representing features and build optimal classifiers that can be used to effectively detect ransomware. Our method is able to detect unseen ransomware including variants of the WannaCry ransomware with high accuracy and was able to distinguish ransomware from benign programs with high precision. We have also demonstrated that our approach consumes small system resources and does not require hardware modification.

In chapter 4, we explored the design and implementation of a new cloud computing model that allows cloud apps developed by third-parties to perform privileged tasks on a client's VMs. Clients can implement a variety of services, such as low-level system services and I/O interception. We discussed the infrastructure support needed to support cloud apps, explored various design options to implement cloud apps, demonstrated and evaluated the practicality of cloud apps using various examples.

Overall, this dissertation proposes an initial approach to using recent hardware advances and redesigning the security model in today's cloud platforms to enhance the security and

privacy of cloud-based applications. We believe that our work serves as the foundation for future directions for further enhancements in security and privacy of cloud-based applications.

References

[1] H. Takabi, J. B.D. Joshi, and G. Joon Ahn. Security and privacy challenges in cloud computing environments. In *IEEE Security and Privacy*, volume 8, 2010.

[2] S. Subashini and V. Kavitha. A survey on security issues in service delivery models of cloud computing. In *Journal of Network and Computer Applications, Elsevier*, 2011.

[3] Google docs leaks out private data. In *Info Security Magazine*, 2009. https://www.infosecurity-magazine.com/news/google-docs-leaks-out-private-data/.

[4] Borko Furht and Armando J. Escalante. Handbook of Cloud Computing, 2010.

[5] AWS CloudHSM. https://aws.amazon.com/cloudhsm/.

[6] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 85–100, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043566. URL http://doi.acm.org/10.1145/2043556.2043566.

[7] Florian Hahn and Florian Kerschbaum. Searchable encryption with secure and efficient updates. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 310–320, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2957-6. doi: 10.1145/2660267.2660297. URL http://doi.acm.org/10.1145/2660267.2660297.

[8] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. Innovative technology for CPU based attestation and sealing. In *HASP Workshop*, 2013.

[9] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *HASP Workshop*, 2013.

[10] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanhbogue, and U. R. Shevegaonkar. Innovative instructions and software model for isolated execution. In *HASP Workshop*, 2013.

[11] David Kaplan, Jeremy Powell, and Tom Woller. Amd memory encryption. In *White Paper*, 2016.

[12] Aws acceptable use policy. 2016. https://aws.amazon.com/aup/.

[13] Hackers find a home in Amazon's EC2 cloud. https://www.infoworld.com/article/2629005/hacking/hackers-find-a-home-in-amazon-s-ec2-cloud.html.

[14] Google security whitepaper. 2017. https://cloud.google.com/security/whitepaper.

[15] Cerber: A Case in Point of Ransomware Leveraging Cloud Platform. https://blog.trendmicro.com/trendlabs-security-intelligence/cerber-ransomware-leveraging-cloud-platforms/.

[16] N. Scaife, H. Carter, P. Traynor, and K. R. B. Butler. CryptoLock (and Drop It: Stopping Ransomware Attacks on User Data. In *36th IEEE International Conference on Distributed Computing Systems (ICDCS 2016)*, June 2016.

[17] A. Kharaz, S. Arshad, C. Mulliner, W. Robertson, and E. Kirda. UNVEIL: A Large-Scale, Automated Approach to Detecting Ransomware. In *25th USENIX Security Symposium*, August 2016.

[18] A. Continella, A. Guagnelli, G. Zingaro, G. De Pasquale, , A. Barenghi, S. Zanero, and F. Maggi. ShieldFS: A Self-healing, Ransomware-aware Filesystem. In *Annual Computer Security Applications Conference (ACSAC) 2016*, December 2016.

[19] Intrusion Detection and Prevention Systems. https://aws.amazon.com/mp/scenarios/security/ids/.

[20] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. *SIGPLAN Not.*, 43(3):2–13, March 2008. ISSN 0362-1340. doi: 10.1145/1353536.1346284. URL http://doi.acm.org/10.1145/1353536.1346284.

[21] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. Inktag: Secure applications on an untrusted operating system. *SIGPLAN Not.*, 48(4):265–278, March 2013. ISSN 0362-1340. doi: 10.1145/2499368.2451146. URL http://doi.acm.org/10.1145/2499368.2451146.

[22] Richard Ta-Min, Lionel Litty, and David Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 279–292, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1. URL http://dl.acm.org/citation.cfm?id=1298455.1298482.

[23] Jisoo Yang and Kang G. Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, pages 71–80, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-796-4. doi: 10.1145/1346256.1346267. URL http://doi.acm.org/10.1145/1346256.1346267.

[24] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *23rd ACM Symposium on Operating Systems Principles*, 2011.

[25] J. Greene. Intel trusted execution technology. In *Intel Technology White Paper*, 2012.

[26] ARM. Building a secure system using trustzone technology. In *PRD29-GENC-009492C*, 2008.

[27] Google Cloud Platform: Customer Responsibility Matrix. https://cloud.google.com/files/PCI_DSS_Shared_Responsibility_GCP_v32.pdf.

[28] S. Davenport and R. Ford. SGX: The good, the bad and the downright ugly. In *Virus Bulletin*, January 2014. https://www.virusbtn.com/virusbulletin/archive/2014/01/vb201401-SGX.

[29] J. Rutkowska. Thoughts on Intel's upcoming Software Guard Extensions (part 1), August 2013. http://theinvisiblethings.blogspot.com/2013/08/thoughts-on-intelsupcoming-software.html.

[30] J. Rutkowska. Thoughts on Intel's upcoming Software Guard Extensions (part 2), September 2013. http://theinvisiblethings.blogspot.com/2013/09/thoughts-on-intelsupcoming-software.html.

[31] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry. Minibox: A two-way sandbox for x86 native code. In *USENIX Annual Tech. Conf.*, 2014.

[32] Symantec Security Response. Ransomware: A Growing Menace. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/ransomware-a-growing-menace.pdf, .

[33] Trend Micro. Ransomware Bill Seeks to Curb the Extortion Malware Epidemic. https://www.trendmicro.com/vinfo/us/security/news/cybercrime-and-digital-threats/ransomware-bill-curb-the-extortion-malware-epidemic.

[34] Symantec Security Response. What you need to know about the WannaCry Ransomware. https://www.symantec.com/connect/blogs/what-you-need-know-about-wannacry-ransomware, .

[35] Wikipedia. WannaCry ransomware attack. https://en.wikipedia.org/wiki/WannaCry_ransomware_attack.

[36] Zdnet. How WannaCrypt attacks. http://www.zdnet.com/article/how-wannacrypt-attacks/.

[37] BBC. Massive ransomware infection hits computers in 99 countries. http://www.bbc.com/news/technology-39901382.

[38] Amazon Marketplace. https://aws.amazon.com/marketplace.

[39] Intel. Intel Software Guard Extensions programming reference (rev. 1) #329298-001US, September 2013.

[40] Intel. Intel Software Guard Extensions programming reference (rev. 2) #329298-002US, October 2014.

[41] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *USENIX Security*, 2004.

[42] B. Yee, D. Sehr, G. Dardyk, J. Bradley Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE S and P*, 2009.

[43] SGX protected memory limit in SGX, 2016. Platform and Technology Discussion, Intel Software Guard Extensions (Intel SGX), https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/670322.

[44] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza. AsyncShock: Exploiting synchronization bugs in Intel SGX enclaves. In *ESORICS*, 2016.

[45] P. Jain, S. Desai, S. Kim, M. Shih, J. Kee, C. Choi, Y. Shin, T. Kim, B. Kang, and D. Han. OpenSGX: An open platform for SGX research. In *NDSS*, 2016.

[46] Intel software guard extensions (Intel SGX) SDK, . https://software.intel.com/en-us/sgx-sdk/download.

[47] Y. Yu. Intel software guard extensions for Linux OS, 2016. https://01.org/intel-softwareguard-eXtensions.

[48] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas. Intel software guard extensions (Intel SGX) support for dynamic memory management inside an enclave. In *HASP Workshop*, 2016.

[49] B. Xing, M. Shanahan, and R. Leslie-Hurd. Intel software guard extensions (Intel SGX) software support for dynamic memory allocation inside an enclave. In *HASP Workshop*, 2016.

[50] U. Drepper. How to write shared libraries. https://software.intel.com/sites/default/files/m/a/1/e/dsohowto.pdf.

[51] SCO. System v application binary interface, intel386 architecture processor supplement. http://www.sco.com/developers/devspecs/abi386-4.pdf.

[52] Intel64 and IA-32 architectures software developer's manual volume 2 (2a, 2b, 2c & 2d): Instruction set reference, A-Z, 2016.

[53] musl libc: standard C/POSIX library and extensions. https://www.musl-libc.org.

[54] The GNU C library (glibc). https://www.gnu.org/software/libc/index.html.

[55] R. Sinha, M. Costa, A. Lal, N. Lopes, S. Rajamani, S. Seshia, and K. Vaswani. A design and verification methodology for secure isolated regions. In *ACM PLDI*, 2016.

[56] QEMU: a generic, open-source machine emulator & virtualizer. http://qemu.org.

[57] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Tech. Conf.*, 2012.

[58] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for C. In *ACM PLDI*, 2009.

[59] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC and LLVM. In *USENIX Security Symposium*, 2014.

[60] T. Roeder. LLVM - Add forward-edge control-flow integrity support. https://reviews.llvm.org/D4167.

[61] The Cyber Threat Alliance. Lucrative Ransomware Attacks: Analysis of the CryptoWall Version 3 Threat. https://www.cyberthreatalliance.org/pdf/cryptowall-report.pdf.

[62] J. Oberheide, E. Cooke, and F. Jahanian. CloudAV: N-Version Antivirus in the Network Cloud. In *Proceedings of the 17th USENIX conference on Security*, July 2008.

[63] Intel Software Developer's Manuals. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf, .

[64] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo. On the Feasibility of Online Malware Detection with Performance Counters. In *The 40th International Symposium on Computer Architecture (ISCA)*, June 2013.

[65] A. Tang, S. Sethumadhavan, and S. Stolfo. Unsupervised Anomaly-based Malware Detection using Hardware Features. In *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses*, September 2014.

[66] K. N. Khasawneh, M. Ozsoy, C. Donovick, N. Abu Ghazaleh, and D. Ponomarev3. Ensemble Learning for Low-level Hardware-supported Malware Detection. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions and Defenses*, November 2015.

[67] B. Singh, D. Evtyushkin, J. Elwell, R. Riley, , and I. Cervesato. On the Detection of Kernel-Level Rootkits Using Hardware Performance Counters. In *Proceedings of ACM Asia Conference on Computer and Communications Security (ASIACCS) 2017*, April 2017.

[68] D. Bilar. Opcodes as predictor for malware. In *International Journal of Electronic Security and Digital Forensic, Volume 1 Issue 2*, May 2007.

[69] Y. Xia, Y. Liu, H. Chen, and B. Zang. CFIMon: Detecting violation of control flow integrity using performance counters. In *Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2012.

[70] G. Ammons, T. Ball, and J. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the SIGPLAN 97 Conference on Programming Language Design and Implementation (PLDI)*, May 1997.

[71] R. Azimi, M. Stumm, and R. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *Proceedings of the 19th annual international conference on Supercomputing*, June 2005.

[72] J. Levon and P. Elie. OProfile: A System Profiler for Linux. http://oprofile.sourceforge.net, 2010.

[73] Performance Counters. https://msdn.microsoft.com/en-us/library/windows/desktop/aa373083(v=vs.85).aspx, .

[74] NGINX. NGINX, a free open-source, HTTP server. https://www.nginx.com/.

[75] Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

[76] VMWare. VMWare Workstation. http://www.vmware.com/products/workstation.html.

[77] Intel's Haswell CPU Microarchitecture). https://en.wikipedia.org/wiki/Haswell_(microarchitecture).

[78] M. Verleysen and D. Francois. The curse of dimensionality in data mining and time series prediction. In *Computational Intelligence and Bioinspired Systems*, 2005.

[79] Regularization. https://en.wikipedia.org/wiki/Regularization_(mathematics).

[80] R. Tibshirani. Regression Shrinkage and Selection via the Lasso, 1996.

[81] M. Ozsoy, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev. Malware-Aware Processors: A Framework for Efficient Online Malware Detection. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, February 2015.

[82] Intel's Floating Point Assists. https://software.intel.com/en-us/articles/x87-and-sse-floating-point-assists-in-ia-32-flush-to-zero-ftz-and-denormals-are-zero-daz, .

[83] Intel 64 and IA-32 Architectures Optimization Reference Manual. https://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf, .

[84] Out-of-order Execution). https://en.wikipedia.org/wiki/Out-of-order_execution.

[85] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. In *Journal of Machine Learning Research*, volume 12, pages 2825–2830, November 2011.

[86] Intel's Nehalem CPU Microarchitecture). https://en.wikipedia.org/wiki/Nehalem_(microarchitecture).

[87] PCC. Pearson Correlation Coefficient. https://en.wikipedia.org/wiki/Pearson_correlation_coefficient.

[88] VMWare vApp. http://www.vmwarehub.com/managing-VMWare-vApp.html.

[89] . http://www.netex.com.

[90] . http://www.sourcefire.com.

[91] AWS MarketPlace find and buy server software and services that run on the AWS Cloud, . http://aws.amazon.com/amis.

[92] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *NDSS*, 2003.

[93] libvmi. https://github.com/bdpayne/libvmi.

[94] A. Srivastava and J. T. Giffin. Tamper-Resistant, Application-Aware Blocking of Malicious Network Connections. In *RAID*, 2008.

[95] 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.

[96] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *CCS*, 2012.

[97] L. Litty, H. Andres, L-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. In *USENIX Security*, 2008.

[98] Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vtpm: Virtualizing the trusted platform module. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267336.1267357.

[99] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems*, 2014.

[100] Sven Bugiel, Stefan Nürnberger, Thomas Pöppelmann, Ahmad-Reza Sadeghi, and Thomas Schneider. Amazonia: When elasticity snaps back. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 389–400, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0948-6. doi: 10.1145/2046707.2046753. URL http://doi.acm.org/10.1145/2046707.2046753.

[101] Jinpeng Wei, Xiaolan Zhang, Glenn Ammons, Vasanth Bala, and Peng Ning. Managing security of virtual machine images in a cloud environment. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, CCSW '09, pages 91–96, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-784-4. doi: 10.1145/1655008.1655021. URL http://doi.acm.org/10.1145/1655008.1655021.

[102] Intel Virtualization Technology Extensions for High Performance Protection Domains, . http://www.lynuxworks.com/virtualization/virtual-machines-intel-virtualization.php.

[103] M. Ben-Yahuda, M. D. Day, Z. Dubitsky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B. Yassour. The Turtles project: Design and implementation of nested virtualization. In *USENIX OSDI*, 2010.

[104] D. Williams, H. Jamjoom, and H. Weatherspoon. The Xen-Blanket: virtualize once, run everywhere. In *EuroSys 12*, 2012.

[105] Z. Qazi, C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying middlebox policy enforcement using SDN. In *SIGCOMM*, 2013.

[106] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *NSDI*, 2012.

[107] Dilip A. Joseph, Arsalan Tavakoli, and Ion Stoica. A policy-aware Switching Layer for Data Centers. In *SIGCOMM*, 2008.

[108] Project Floodlight. http://www.projectfloodlight.org/projects/.

[109] Floodlight REST API. http://www.openflowhub.org/display/floodlightcontroller/REST+API.

[110] iperf, February 2014. https://github.com/esnet/iperf.

[111] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *USENIX LISA*, 1999.

[112] libpcap, December 2013. http://sourceforge.net/projects/libpcap/.

[113] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with Haven. *ACM TOCS*, 33(3), 2015.

[114] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *IEEE S and P*, 2015.

[115] S. Brenner, C. Wulf, M. Lorenz, N. Weichbrodt, D. Goltzsche, C. Fetzer, P. Pietzuch, and R. Kapitza. Securekeeper: Confidential zookeeper using Intel SGX. In *ACM Middleware*, 2016.

[116] M. Shih, M. Kumar, T. Kim, and A. Gavrilovska. S-NFV: Securing NFV states by using SGX. In *1st ACM Intl. Wkshp. on Security in SDN and NFV*, 2016.

[117] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security Symposium*, 2016.

[118] R. Sinha, S. Rajamani, S. Seshia, and K. Vaswani. Moat: Verifying confidentiality of enclave programs. In *ACM CCS*, 2015.

[119] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE S and P*, 2015.

[120] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *ACM/USENIX OSDI*, 2016.

[121] N. Rosenblum, X. Zhu, B. Miller, and K. Hunt. Learning to analyze binary computer code. In *AAAI Conference*, 2008.

[122] E. Chul Richard Shin, D. Song, and R. Moazzezi. Recognizing functions in binaries with neural networks. In *USENIX Security Symposium*, 2015.

[123] U. Erlingsson, Y. Younan, and F. Piessens. Low-level software security by example. In *Handbook of Information and Communication Security*, 2010.

[124] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: software guards for system address spaces. In *ACM/USENIX OSDI*, 2006.

[125] B. Niu and G. Tan. Monitor integrity protection with space efficiency and separate compilation. In *ACM CCS*, 2013.

[126] Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *IEEE S and P*, 2010.

[127] A. Kharraz, W. Robertson, D. Balzarotti, L. Bilge, , and E. Kirda. Cutting the Gordian Knot: A Look Under the Hood of Ransomware Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 12th International Conference, DIMVA*, July 2015.

[128] N. Andronio, S. Zanero, and F. Maggi. HelDroid: Dissecting and Detecting Mobile Ransomware. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses*, November 2015.

[129] I. Santos, F. Brezo, J. Nieves, Y. K. Penya, B. Sanz, C. Laorden, and P. G. Bringas. Idea: Opcode-Sequence-Based Malware Detection. In *Engineering Secure Software and Systems. ESSoS 2010*, 2010.

[130] G. Yan, N. Brown, and D. Kong. Exploring discriminatory features for automated malware classification. In *Proceedings of the 10th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'13)*, July 2013.

[131] N. Runwal, R. M. Low, and M. Stamp. Opcode graph similarity and metamorphic detection. In *Comput Virol 2012*, April 2012.

[132] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion attacks against machine learning at test time. In Hendrik Blockeel, Kristian Kersting, Siegfried Nijssen, and Filip Železný, editors, *Machine Learning and Knowledge Discovery in Databases*, pages 387–402, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-40994-3.

[133] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *arXiv preprint arXiv:1412.6572*, 2014.

[134] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *Security and Privacy (SP)*, pages 39–57, 2017.

[135] Nicolas Papernot, Patrick D. McDaniel, and Ian J. Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *CoRR*, abs/1605.07277, 2016. URL `http://arxiv.org/abs/1605.07277`.

[136] Weilin Xu, Yanjun Qi, and David Evans. Automatically evading classifiers. In *Proceedings of the 2016 Network and Distributed Systems Symposium*, 2016.

[137] Mehran Mozaffari-Kermani, Susmita Sur-Kolay, Anand Raghunathan, and Niraj K. Jha. Systematic poisoning attacks on and defenses for machine learning in healthcare. In *IEEE journal of biomedical and health informatics 19, 6*, pages 1893–1905, 2015.

[138] Blaine Nelson, Marco Barreno, Fuching Jack Chi, Anthony D. Joseph, Benjamin I. P. Rubinstein, Udam Saini, Charles Sutton, J. D. Tygar, and Kai Xia. Exploiting machine learning to subvert your spam filter. In *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, LEET'08, pages 7:1–7:9, Berkeley,

CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1387709.1387716.

[139] Octavian Suciu, Radu Marginean, Yigitcan Kaya, Hal Daume III, and Tudor Dumitras. When does machine learning fail? generalized transferability for evasion and poisoning attacks. In *Proceedings of the 27th USENIX Security Symposium*, 2018.

[140] . http://www.http://aws.amazon.com/vpc/.

[141] P. M. Chen and B. Noble. When Virtual is Better than Real. In *HotOS*, 2001.

[142] S. Butt, A. L-Cavilla, A. Srivastava, and V. Ganapathy. Self-service Cloud Computing. In *ACM CCS*, 2012.