DevSecOps of Containerization

by

Pinchen Cui

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
August 8, 2020

Keywords: Container, Monitoring, Machine Learning, Blockchain

Approved by

David Umphress, Chair, Professor of Computer Science
Anthony Skjellum, Professor of Computer Science
Ujjwal Guin, Assistant Professor of Electrical and Computer Engineering
Tao Shu, Assistant Professor of Computer Science

Abstract

Containerization is a new concept of virtualization, one that has attracted attention and occupied considerable amount of market size due to its inherent lightweight characteristics. However, the lightweight advantage is achieved at the price of security. The isolation of containers cannot be as strong as with traditional hypervisor-based virtualization. Attacks against weak isolation of the container have been reported, and the use of shared kernel is another targeted vulnerable point. This work focuses on providing security for the containerization. We aim to provide secure monitoring of containerized application, which can help the infrastructure owner ensure the running application is harmless. The monitoring is non-intrusive and lightweight with no user data privacy and performance overhead problems being incurred. We propose use of machine learning techniques combined with container introspection tools to perform intelligent monitoring. We establish an unique public dataset to provide better emulation of real application behaviors and better coverage of attacks with expanded feature space. Sufficient related work is surveyed, and a proof-of-concept monitoring system is implemented and evaluated. In addition, we also investigate the containerization of Hyperledger blockchain systems. Smart contract is one of the most important and promising feature of blockchain, and it relies on the use of virtualization. Hyperledger implements its chaincode (smart contract) based on containerization. Thus, the DevSecOps of containerization also determines the security of Hyperledger systems. The potential risk of Hyperledger containerization lifecycle have been illustrated and discussed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, products and services no longer have local installation constraints because users are able to access their data and applications from anywhere using online cloud services. The concepts of Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS) have emerged to provide a stratified collection of computing functions (as shown in Figure 1.1). Collectively referred to cloud computing, they enable the dynamic resource pooling in a multitenant model, where physical or virtual resources can be assigned and reassigned according to tenant demand [7]. The development and deployment of cloud computing technology saw services transferred from offline to online, systems migrated from physical to virtual, and the environments offering better availability and superior extensibility. The market has validated the need: almost $260 billion revenue has been made by worldwide cloud services in 2017 [8].

Although the cloud infrastructure presents a relatively simple concept, overall system complexity is increased, and along with security in such complex system confront significant challenges. In addition to being subject to common security issues such as Rootkit, Malware, DDoS attack, and Replay attack, the cloud environment poses specific and distinctive security vulnerabilities. The virtualized nature of the cloud infrastructure mean, many attacks against virtual machine (VM) can also be performed in the cloud, *e.g.,* VM escape, VM cloning and VM rollback, and Cross-VM attacks [9]. The loss of physical control to the data stored in the cloud may also cause problems, namely, the data integrity and confidentiality cannot be ensured [10]. The shift away from individually operated system to consolidated centralized architectures requires the service providers guard against of abusive use of cloud resources. For instance, the

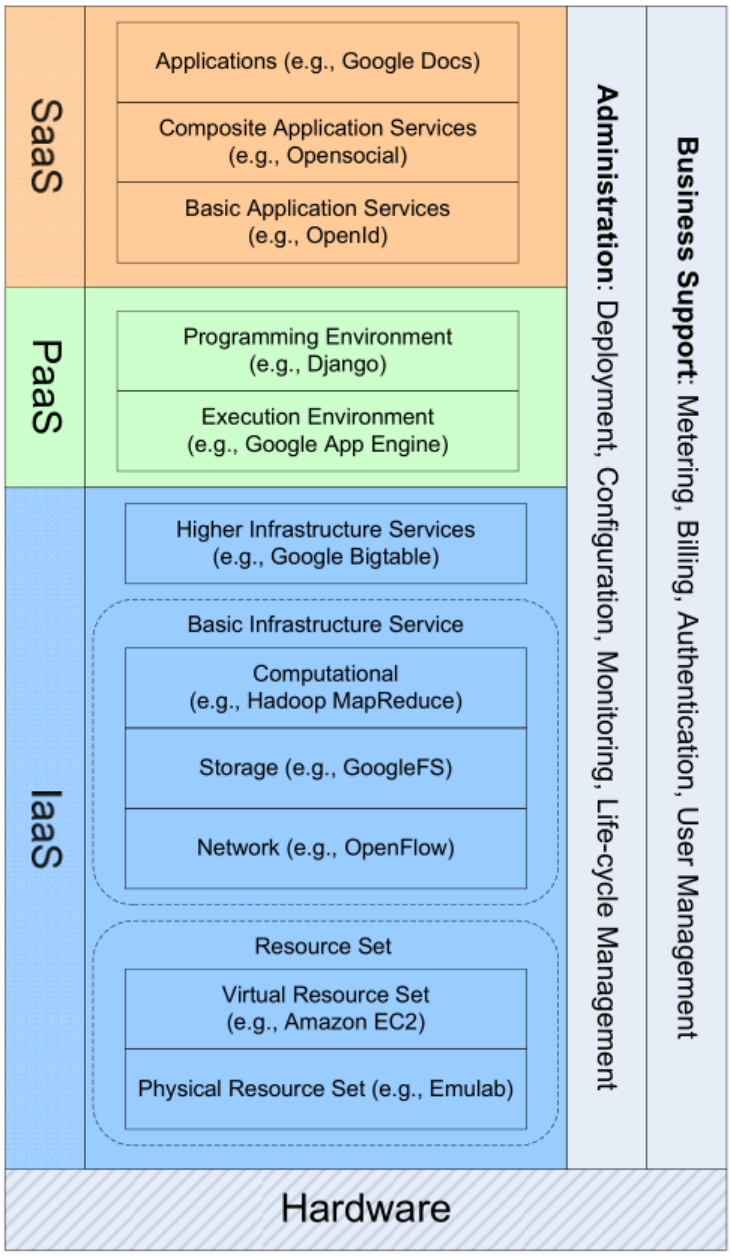


Figure 1.1: Architecture of Cloud Computing [1]).

Amazon cloud service has been used as a launching point for botnet operations, and it also has be used to host instances of malware [11]. A Microsoft intelligent report indicates cloud-based attacks increased by 300% in 2017 [12].

The aforementioned security concerns are only the tip of an iceberg since the complexity of the systems are still increasing day by day. At this point in time, hypervisor-based virtualization solutions are no longer the only player in the cloud environment; container-based lightweight virtualization solutions have begun to show their market potential. Recently, many advanced IT companies have launched container based solutions, *e.g.,* Amazon ECS, Azure Container Service, and Google Container Engine [13]. The overall market size has increased 40% every year and revenue will reach $5 billion by 2023 [14, 15]. The key difference between container and hypervisor architectures is that hypervisor-based systems require full virtualization of an operating system whereas, in the container-based systems, containers and applications share an operating system, thus decreasing the cost of deployment. There is a trade off: containers cannot be run across operating systems (Linux/ Windows) and the isolation cannot be as strong with hypervisors. Containers are, in short, less portable and less secure than hypervisors [16]. And in fact, attacks against weak isolation of the container have been reported in the past few years, and the use of shared kernel is another targeted vulnerable point [17, 18].

Though containers are arguably less secure than VMs, their popularity is increasing. This work focuses on providing security for the containerization. The security problem of the containerization is twofold: (i) For the infrastructure owner's perspective, each virtualized instance or each virtualized layer in the system may belong to different tenants. The service provider/ infrastructure provider should take care of the security for these users in different manners. It would be non-trivial to develop varies security frameworks for all types of applications, because do so incurs tremendous cost for data analysis and mechanism design if every single application needs to concern separately. Reliance on a general security monitoring system may not be sufficient. (ii) Although customers run their applications/services on remote servers, they do not want to fully expose the details of their applications/services, choosing, instead, to ensure data privacy from the infrastructure providers. Meanwhile, developing or providing security features and frameworks are challenging. This is a tough situation for both sides: the

customer cannot afford the cost of re-development and needs security guarantees that prevent leaking of privacy and data; the service provider needs to develop reliable and available tools to monitor and secure unknown applications.

We suggest that the dilemma of monitoring cloud environments in a comprehensive, yet noninvasive, fashion can be unraveled by augmenting virtual machine intelligence (VMI) [19] tools with computational intelligence. A VMI-based solution may be an optional solution for the non-cloud systems, but in the cloud scheme, the whole system is built on a virtualized environment. VMI is no longer an option but an absolute necessity for reliable and secure monitoring. By adopting intelligent techniques such as machine learning and data mining, a self-adapting and self-learning program can be trained to automatically monitor a system. We envision an intelligent based monitoring agent being deployed out of the target system (in the upper layer), automatically extracting data from the target and using the data to train monitoring tools how to discern what to monitor and how to monitor for the particular application, and, ultimately, detecting anomalies. Our goal is to carry out the entire monitoring process in the absence of the human interaction, meaning, the customer does not need to have the detailed security knowledge and the service provider cannot gain or reveal any sensitive data from the monitoring.

In order to provide reliable monitoring data and environment, we built a new open source system call-based dataset for anomaly detection using state-of-art introspection tools with novel underlying dataset designs. The traditional and widely used datasets [20–26] have distinct pitfalls for the objective of anomaly detection (especially for application monitoring, more details can be found in Section 4). For example, these datasets may be limited to the network traces (mainly generated with tcpdumps). However, mis-operation, software faults, bad configurations, and errors in the containerization life-cycle (e.g. hypervisor fails, a malicious docker exec from a compromised user, or docker escape) can all produce anomalies in a monitored target system. Network traces are sufficient for network-based intrusion detection, but they are not comprehensive enough for general anomaly detection. Traces based on system calls can provide more sophisticated and granular behavioral details. Previous work has demonstrated

the utility of system call traces [21, 22, 27] but has relied on the system call name accompanied, in some cases, by a timestamp. We extend the feature space with caller process name and system call arguments to enhance the interpretation of application behaviors.

On the other hand, cloud computing is not the only application that get benefits from containerization and virtualization. Smart contract and blockchain are rising and promising techniques that recently attract tremendous public attention. In addition, smart contract systems are built upon the underlying virtualization and containerization as well, so that the contract can be securely executed on different platform with guaranteed consistency. The DevSecOp of containerization determines the security of smart contract system.

Thus, we also investigate the containerization environment of Hyperledger Fabric system to reveal the potential risks. We aim to perturb the execution of the smart contract system. This procedure can be achieved by leveraging the flaws of the underlying smart contract life-cycle or virtualization mechanism (runtime). The Ethereum Virtual Machine in Ethereum, and the Docker container environment in Hyperledger Fabric are the main target. This attack method is completely different with state-of-art attack methods. More details are presented in Chapter 5.

The major content of this work can be summarized as follows:

- We introduce the background in Chapter 2. We present an overview of the virtualization, containerization, machine learning and blockchain. We demonstrate the concepts and related works of VMI and container security. We also perform a detailed analysis and discussion on the emerging solutions in this space. Related intelligent algorithms and works are depicted and compared.

- We verify the proposed concept via a simple Proof-of-Concept (PoC) implementation in Chapter 3 before further investigation and implementation of the proposed method in real production environment.

- In Chapter 4, we propose, implement, and evaluate unsupervised introspection approach over the aforementioned dataset. The evaluation not only demonstrates the efficiency

of the proposed methods, but also explains the necessity of an extension of features for detecting certain classes of anomalies.

- We elicit a new attack vector in the smart contract ecosystem in Chapter 5. Instead of focusing on the smart contract programming, we propose to perturb the smart contract execution at the life-cycle and runtime level (containerization or virtualization layer).

- Finally, we conclude the paper in Chapter 6.

Chapter 2

Background

2.1 Overview of Virtualization

This chapter introduces the general background in this space. We overview the basic concept of virtualization and also summarize traditional monitoring and detection methods. Additionally, we review and introduce the fundamental concepts of VMI and briefly demonstrate the security of container.

2.1.1 Virtualization

Although it may seem that virtualization is a development of the last decade or so, it has a history of research of more than 40 years [2]. Virtualization was first introduced by IBM in 1960's, and was originally used to partition large mainframe computer into multiple logical instances. Currently, the term, "virtualization", is defined as a resources management technique that enables the abstraction of computer system resource and allows a user to virtually partition the computational power, storage space and networking resource. The abstraction of a system resources is provided by the software abstraction layer, which is also known as the Virtual Machine Monitor (VMM) or Hypervisor. Each of the sub-system in the partitioned system is called Virtual Machine (VM). The resource partition and sharing is only the one benefit of virtualization; another primary advantage is the isolation which makes each of the VMs in the system work as independent and non-interfering units [28]. A VM is logically equivalent to a physical machine except it cannot directly execute sensitive system calls, all these privileged calls would be captured and handled by the VMM.

Basic Concepts

Generally, the operating systems work as the abstraction layer of the hardware that offers the interfaces for interacting and managing hardware resources. Normally, only one single operating system can be run at any given time in most of the circumstances. The operating system can be considered as a piece of privileged software and any programs running inside the operating system are less privileged [2].

A VMM is another privileged software that also provides hardware abstraction. VMMs fall into two main categories, depending on their location in the abstract computing stack (see Figure 2.1).

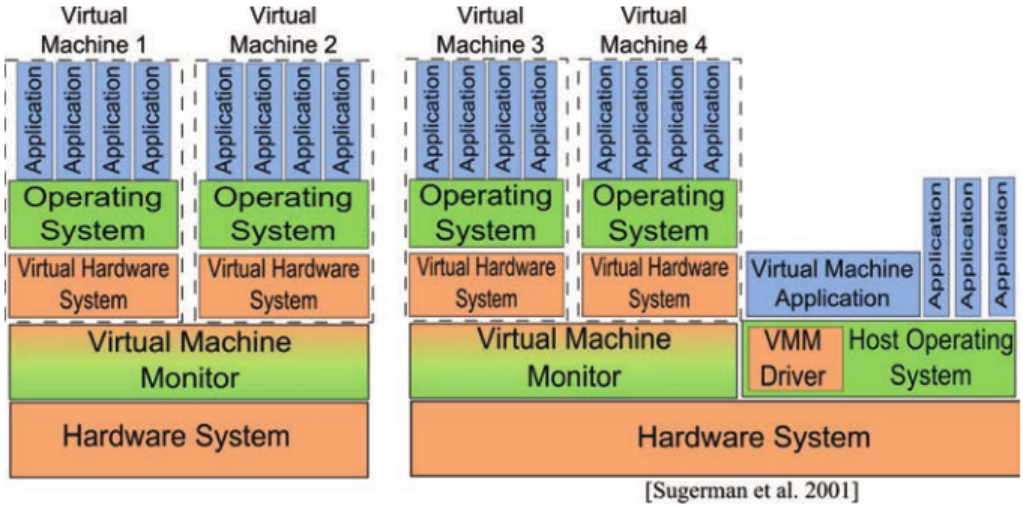


Figure 2.1: Generalized architectures for Type-I (left) and Type-II (right) Virtual Machine Monitors (adopt from [2]).

- *Type 1*: The VMM runs as the first layer above the bare metal and has no host operating system, or the host operating system is just a privileged VM. Some notable implementations are VMWare vSphere/ ESXi, Mircosoft Hyper-V, and XEN.

- *Type 2*: The VMM either runs alongside or above a host operating system. Depending on the implementation, instructions may be directly issued to hardware via shared drivers (with a host OS), or the host OS may handle all the hardware calls. Some notable implementations are Virtual Box, VMware Workstation and KVM.

The advantage of Type 1 virtualization is that the VMM has full control of the system which equates to better performance and security. And in contrast, Type 2 virtualization is more usable in that users can install the hypervisor inside their current OS.

Other terms such as Full-Virtualization and Para-Virtualization are also widely used to classify virtualization. Full-Virtualization solution allows an unmodified operating system to run as a guest and it provides full and transparent virtualization of hardware and operating system. Para-Virtualization requires the modification of guest operating system to accommodate virtualization. In this case, hardware may not be fully virtualized, requiring specific APIs to the added into guest operating system. VMs are able to directly use bare metal resources via added APIs, thus reducing the overhead of the hypervisor. Para-Virtualized VMs are aware they are running in the virtualized environment since the specific system calls are replaced by API calls. "Type 1 or 2" and "Full or Para" are two different taxonomies of virtualization technique and they do not have a mapping relationship. For instance, Type 1 solution XEN support both Full or Para and Type 2 solution KVM also support both of them*.

Note that Type 1 or Type 2 are only used to describe general system virtualization which offers hardware abstraction and operating system virtualization. Other types of virtualization include the Java Virtual Machine (JVM) for application-level virtualization and WINE as a library-level virtualization of Windows features. Containers can be considered as a special case of Type 2 which only virtualized the operating system level. Generally, there are five levels of virtualization: Application level, Library level, Operating System level, Hardware Abstraction level and Instruction Set level. In this work only the hardware level and operating system level virtualization will be discussed.

System and Hardware Virtualization

The VMM or hypervisor is the middleware between the VM and the hardware that controls access to the hardware. How the actual hardware is virtualized and then monitored by VMM is

*KVM does not support para-virtualization for CPU but may support para-virtualization for device drivers to improve I/O performance [29].

another question. The hardware layer consists which can be grouped into three main categories: CPU, Memory and I/O [30]. *

- *CPU Virtualization*: As the most important component of the hardware, the CPU handles all the instruction executions. There is little need to fake or emulate a CPU to execute instructions for each of the VMs. Since the instructions will be ultimately executed by the real physical CPU, adding layers of virtualization or emulation above CPU introduce the execution delay. In practice, CPU virtualization is the mechanism that VMM uses to allocate CPU resources to the VMs. The VMM time-slices physical processors across all VMs so each VM runs as if it has its specified number of virtual processors. If multiple VMs run above the same hypervisor, they have to share the physical resources based on the hypervisor scheduler. Computing can be allocated such that each VM receive an equal share of CPU per virtual CPU, or it can be proportional to user based priority.

  VM instructions are typically executed in two ways:

  1. Software-Based: The unprivileged application codes is run directly on the physical CPU with all the privileged calls intercepted by the hypervisor for further translation or processing before being executed by the CPU. The drawback is slower execution time for all privileged calls.
  2. Hardware-Assisted: In 2005 and 2006, Intel and AMD announced their processor extensions to support x86 architecture virtualization. The Intel-VT and AMD-V (also called AMD-SVM) provide hardware assistance for CPU virtualization [32, 33]. With the hardware assistance, the hypervisor runs in a special CPU mode (*e.g.,* VMX Root Mode) thus, eliminating the need to translate privileged calls and slowing performance.

*Note that, the detailed virtualization concepts and procedures of these three are based on the white paper (or guideline) of the specific virtualization solutions in the market, *i.e.,* KVM [29], XEN [30] and VMware vSphere 4 - ESX and vCenter Server [31]. And all the *virtualization* mentioned in this work are based on x86 architecture. It is also worth mentioning that x86 architecture is not designed for full virtualization [2, 30], and the virtualization of x86 based system had difficulty before the hardware assistance been introduced. *Software-Based* virtualization † of CPU, Memory and I/O have been used to overcome the difficulty of x86 architecture virtualization, but the performance cannot be as well as native; And *Hardware-Assisted* virtualization reduces the performance overhead and solves series of security problems [32, 33]. However, the detailed information of either Software-Based or Hardware-Assisted is not the main interest of this work. This section will only introduce the fundamental concept of the hardware virtualization.

- *Memory Virtualization*: Virtual memory enhances the system performance by providing memory capacity without adding more main memory and has been the main story of modern operating system. Memory virtualization is the technique by which the VMM provides and manages memory space for VMs, as opposed to virtual memory which maps disk space as memory.

  A guest OS inside a VM has both physical and virtual memory, as does the host OS, if it exists. The hypervisor handles memory resource allocation in a similar fashion to CPU virtualization. Memory overcommitment can be achieved by some hypervisors in cases where the total assigned memory size is larger than the total physical memory the machine has. For example, user can have a host with 4GB memory and run four virtual machines with 2GB memory each. This can be done only when some VMs are lightly loaded while others are heavily loaded, and the relative activity levels may vary over time. Some of the VMMs also support memory sharing, *e.g.,* several VMs running the same applications, housing the same components, holding common data, might share memory in an effort to eliminate redundant copies.

  Allocation of memory is only one part of memory virtualization; the most important parts are the memory management and memory address translation. Suppose a guest OS is running on top of a host. There are three types memory addresses in this system: Guest Virtual Address (GVA), Guest Physical Address (GPA), and Host Physical Address (HPA). Figure 2.2 shows this virtualized memory space architecture. The guest OS considers the GPA as a real consistent address space, however, the address space allocated to guest could not be always consistent in the HPA. Although the guest holds its own page table which maps GVA to GPA, the GPA cannot be understood by the CPU. The hypervisor is needed to provide the functionality which translates the memory access inside the guest correctly and efficiently. Implementation falls into two categories, software-based and hardware-assisted.

  An example of a software-based implementation is the KVM Shadow-Page-Table (SPT), which is used to translate GVA to HPA. Every guest OS process has its own SPT. Since

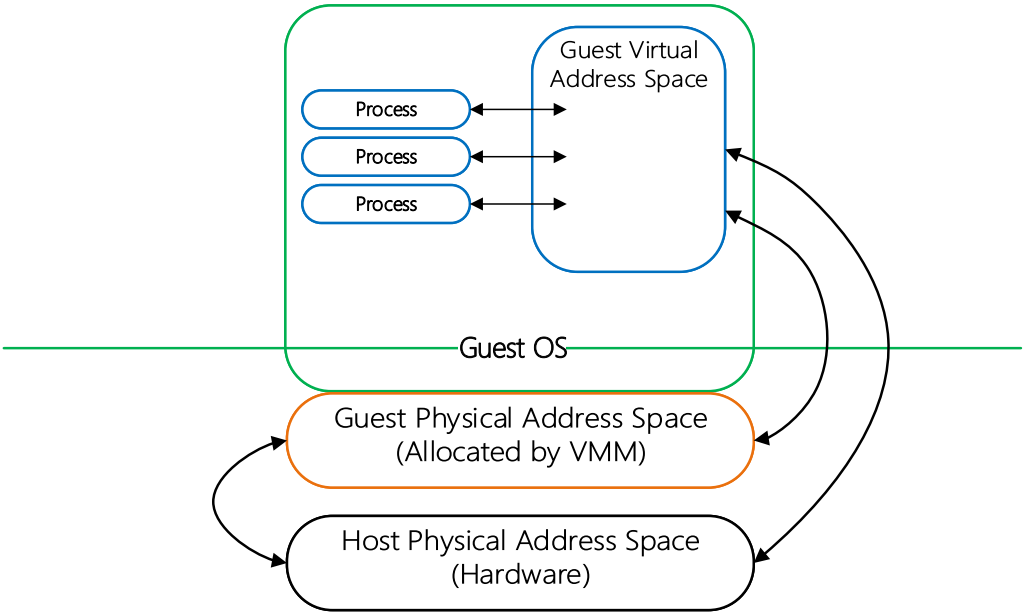


Figure 2.2: Memory Address Spaces of Virtualized System.

different processes have different virtual address spaces, the KVM hypervisor maintains a hash table to record and manage all the SPTs. When a guest OS needs to write an entry into its page table or anytime it raises a page fault, the operation/exception is captured by the KVM hypervisor. The hypervisor checks the guest page table and updates corresponding SPT to map the GVA to HPA. And finally, the SPT is then pushed back to the CPU. The drawbacks of the SPT are: (i) It incurs a large amount of page faults; (ii) The hypervisor needs to maintain large amount of SPTs; (iii) Walking the guest page table requires considerable overhead.

In contrast to the software-based approach, hardware-assisted solution translates GPA to HPA using the Extended-Page-Table (EPT), a feature built into the Intel hardware for memory virtualization. A hypervisor using EPT only needs to maintain one EPT for each VM because it contains the mapping between GPA to HPA.

- *I/O Virtualization*: Virtualization of I/O is one of the most complicated part of system virtualization, because each I/O device type has specific characteristics and needs to be controlled in its own way [34]. Overall, as shown in figure 2.3, there are three types of I/O virtualization strategies [3]: virtual devices, virtual drivers, and passthrough I/O.

Virtual Device is the most straightforward strategy in that it entails constructing a virtual device inside VM and virtualization of I/O activity through the device [34]. When the guest requests the use of a virtual device, the request is intercepted by the hypervisor and then converted to an equivalent request to the actual physical hardware. Virtual Driver (as known as Paravirtualization) has the guest calling the virtual device instead of the real hardware. The VMM provides particular virtual drivers inside the guest OS (which are also called front-end drivers in paravirtualization). Any I/O request made by the guest OS is not handled locally but is forwarded by the front-end driver to the VMM. Passthrough I/O allows the guest to directly use the real hardware. This is based on providing multiple hardware I/O devices, *i.e.,* multiple disks, several NICs, and the VMM provides the scheduling for the I/O devices. Passthrough I/O requires hardware assistance (Intel-VT or AMD-V).

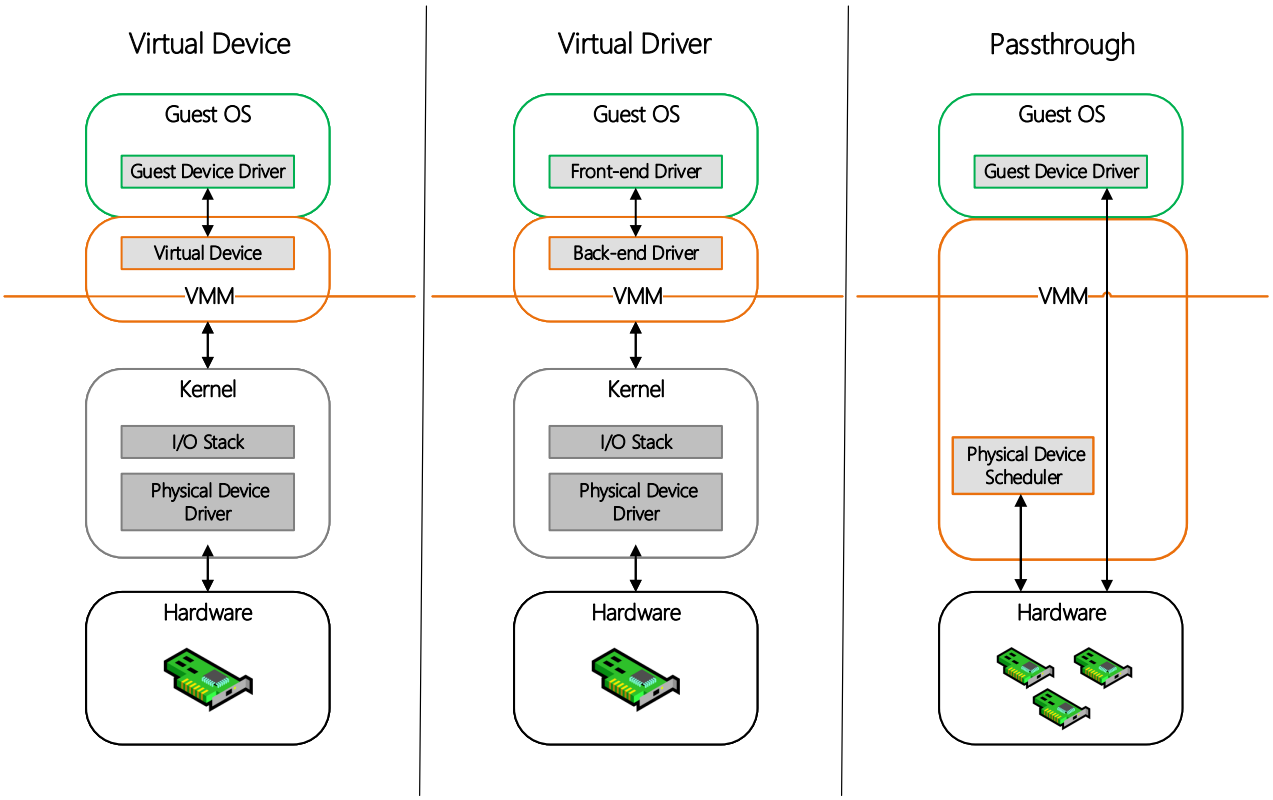


Figure 2.3: Virtualization of I/O [3]

2.1.2 Containerization

Hardware virtualization provides isolation and resource management at the hardware level. Operating system level virtualization provides abstraction of the bare metal and resource management inside a particular operating system environment. Operating system virtualization, also known as the containerization it allows multiple applications (different OS-based apps, Figure 2.4) to operate on top of a single kernel [16]. The root of containerization can be traced back to 1979 when the Unix *chroot* command was introduced as a part of Unix version 7 to provide the isolation functionalities. Twenty years later, FreeBSD implemented *Jail* as an extended-version of *chroot*. Finally, the *container*, was introduced by Solaris 11 in 2010. The idea of containerization is still originated with Linux, but has been recently extended with the Windows arena*. The concepts, though, still remain Linux-centric.

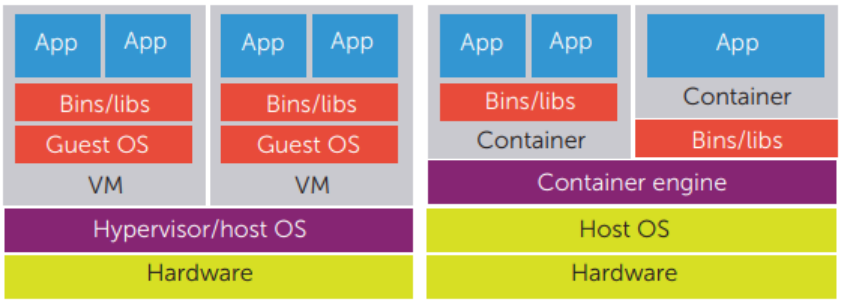


Figure 2.4: Architecture of Containerization [4]

Over the past decade, notable implementations of containerization emerged, including Linux Containers (LXC), Linux-VServer, Warden Containers, Docker, OpenVZ, and Google lmctfy. All the solutions share the same characteristics: the isolation in containerization is achieved using inherent Linux system features such as *chroot*, *cgroup*, *namespace*, *iptable*, and *mount* [16, 18, 35]. Figure 2.5 describes the isolation environment in containerization. Process isolation is achieved by utilizing *namespace*. In particular the PID namespace provides process isolation by allowing each container to has its own process id. *Cgroup* is the underlying mechanism of container resource isolation, which also controls the resource management. *Cgroup* ensures that each container obtains a fair share of the resources, and preventing any container from consuming all resources. Filesystem isolation relies on *Chroot* , *mount*, and *namespace*

*https://www.docker.com/microsoft

which enable the processes in different mount namespaces can have a different view of the file system, achieving isolation. In addition, a root file system (rootfs) is created as the underlying file system for containers by mounting specific directories from host system. The virtualization or the isolation of a network is based on the *namespace*, virtual network , and *iptable*. Different network namespaces are created to host the containers. Each container holds a private IP and a virtual eth interface connected to the bridge. The bridge acts as a "shim" between the container and underlying operating system, and a layer which can translate, filter, or block network communication can be formed with the help of *iptable*.

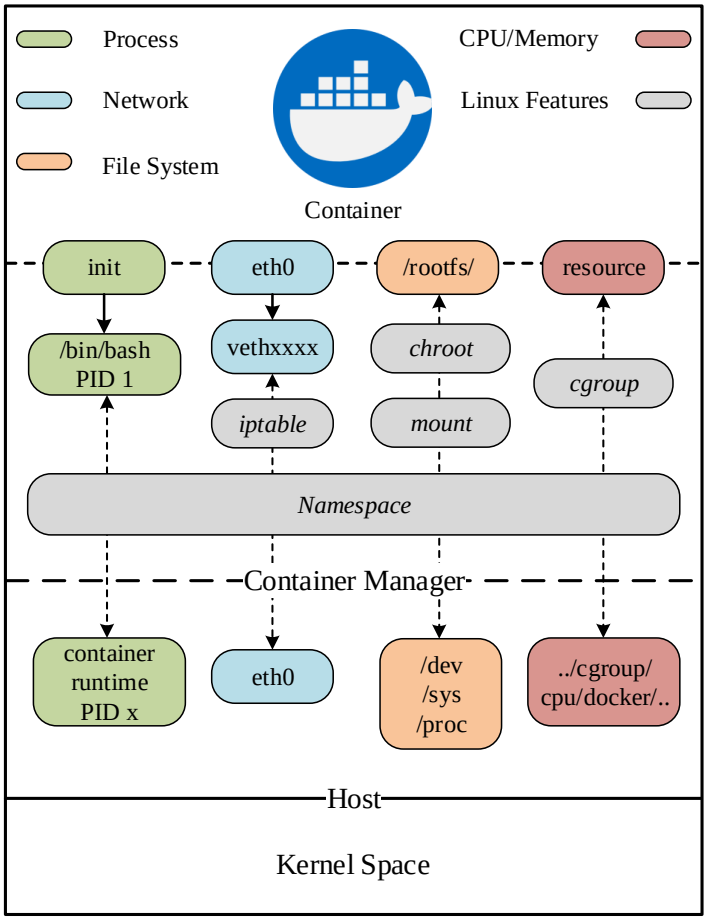


Figure 2.5: Isolation in Containerization

Different containerization solutions have their own specific features, however, such differentiated features are out of the scope of this work. Overall, the performance of containerization has been proved to be better than virtualization, although it still cannot fully reach the native non-virtualized performance [36].

2.2 Cloud Computing

The World Wide Web allowed services and business to be launched online and enabled users to remotely access and share data via network. Virtualization techniques advanced computing further by giving users the means to share computing resources. The Amazon Elastic Compute Cloud service launched in 2006 introduced customers to renting a fraction of a remote physical machine inside cloud. The combination of virtualization, resource management and tenant service mode thus launched a new era: cloud computing [37].

2.2.1 Basic Concepts

The US National Institute of Standards and Technology (NIST), which defines cloud computing as "*a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction.*" [7]. This definition emphasizes five essential characteristics, three service models and four deployment models [7, 38].

Characteristics [7, 38]

- On-demand self-service: The cloud resource could be registered and used without human association to the cloud administration supplier.
- Broad network access: Resources can be accessed and utilized by any type of network-enabled device such as tablets, smart phones or laptops.
- Resource pooling: The resources of a physical machine or a physical cluster can be shared by multiple tenants.
- Rapid elasticity: A tenant can easily and rapidly rent more resource or return unneeded resource.
- Measured service: Resource utilization is measured by monitoring CPU use, storage consumption and network usage, etc.

The attraction to cloud computing is due to the ability of a user to finish a hard and heavy computational e without having to know or address all the detailed IT knowledge.

Service Model

Contemporary, cloud service can be roughly divided into three types: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS).

- Software as a Service (SaaS) provides hosted application services for customers which mainly are accessed by a web browser or an API. The users do not have the control of all the underlying framework. Notable examples are Google Doc [39] and Salesforces CRM [40].
- Platform as a Service (PaaS) provides a platform for the customers to run their own applications. Customers create and produce an application from among a set of programming languages and tools that are supported by the PaaS provider. As with SaaS, PaaS users do not have the control of the whole framework. Prominent examples are Google App Engine [41] and Microsoft Azure [42].
- Infrastructure as a Service (IaaS) provides customers with control over low-level the capabilities of processing power and storage space. It also allows the customers to run any type of software and including operating systems. Customers typically launch their own VMs and gain full control of the guest OS. The Amazon Web Service EC2 [43] is a well known example.

Deployment Model

In contrast to the service model, which defines how the cloud provides service to its user, the deployment model describes how the cloud is being managed by its owner and how can the customers access the cloud. NIST enumerates four different cloud deployment models: public cloud, private cloud, community cloud, and hybrid cloud [7].

- A Public Cloud is owned, managed and operated by all organizations and is available to a broad audience. Public cloud is often used by large corporations to offer services,

that extend beyond the boundaries of the company. Public clouds are marked by their openness and inclusiveness but are subject to security concerns.

- A Private Cloud is permissioned cloud that is exclusively used by a specific organization, company or individual.

- A Community Cloud is a cloud shared by several trusted organizations. Community clouds are still permissioned but used by a group of users that have shared concern, such as schools within a university [38].

- A Hybrid Cloud is comprised of a combination of public cloud, private cloud or community cloud. Hybrid cloud is usually used to provide different levels of data privacy.

2.3 Virtualization Security

2.3.1 General Anomaly Detection

Security threats generally fall into two types: internal threats and external threats. Internal threats usually refer to the anomalous or malicious behaviors originated from native internal components and applications. Key to detecting such threats is defining normal/abnormal behaviors. Anomaly detection examines behaviors and compares them to find what is deemed ”normal”. Behaviors that exceed the threshold of normal are classified as ”abnormal”. Alternatively, ”abnormal” behaviors can be characterized with all unknown events that are not abnormal considered as normal. This method is known as misuse detection. Misuse detection is effective in detecting known attacks with low errors, however, it cannot detect newly created attacks that do not have similar properties to the known attacks [44]. Anomaly detection can identify new attacks but is not effective as misuse detection. The comprehensive solution for internal security includes leveraging both of the anomaly detection and misuse detection.

A system also can be attacked or compromised by a remote attacker, making the threat an external one. Intrusion detection is a frequently used to address such threats. Note that the overlaps exist in the three basic detection methods as shown in the 2.6. Figure 2.6 illustrates the relationship among threats. It should be noted that dealing with threats does not have clear

cut boundaries. For example, anomalous behaviors caused by intrusion or misuse, or intrusion detection relies on the underlying idea of anomaly detection or misuse detection.

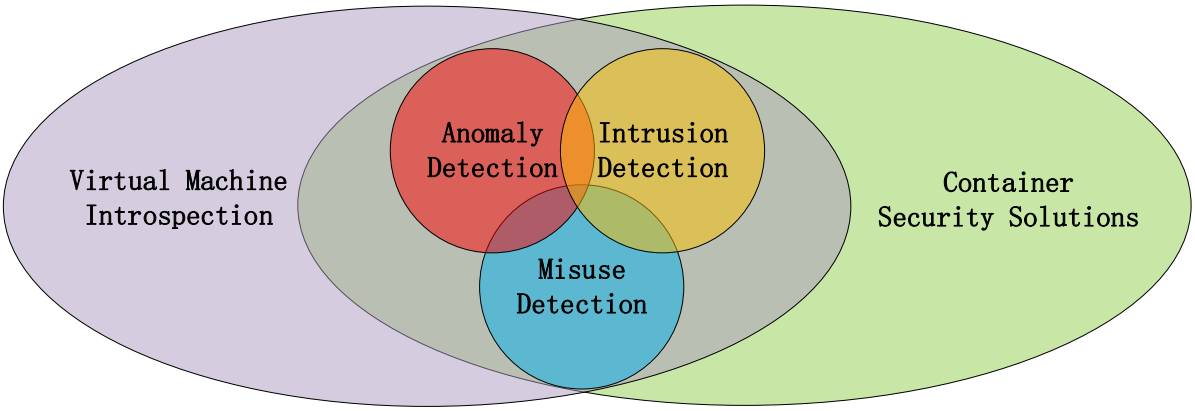


Figure 2.6: General Methods

This research focuses on anomaly detection. The basic idea of anomaly detection is to find and detect the patterns or traces in data that do not conform to defined normal behaviors. These abnormal behaviors are known as anomalies or outliers and the anomaly detection problem starts with the definition of anomaly. The following are conventional perspective of anomalies [45]:

1. Point Anomalies: An individual data instance that deviates from the majority (normal behaviors) is considered a point anomaly. As shown in Figure 2.7.(a) N_1 and N_2 are defined regions of normal behaviors, points o_1 and o_2 as well as the points in the points cluster, O_3, are classified as anomalies.

2. Contextual Anomalies: Also known as conditional anomalies, contextual refers anomalies to behaviors that is unwanted or unexpected based on the surrounding context. Figure 2.7.(b) shows an instance of a contextual anomaly, where data in a time interval differs from the data in preceding and succeeding intervals.

3. Collective Anomalies: A collection of data instances that appears to be abnormal with respect to entire data set is considered a collective anomaly. Individual data instances in the collection may not be anomalous when considered individually, but the data when

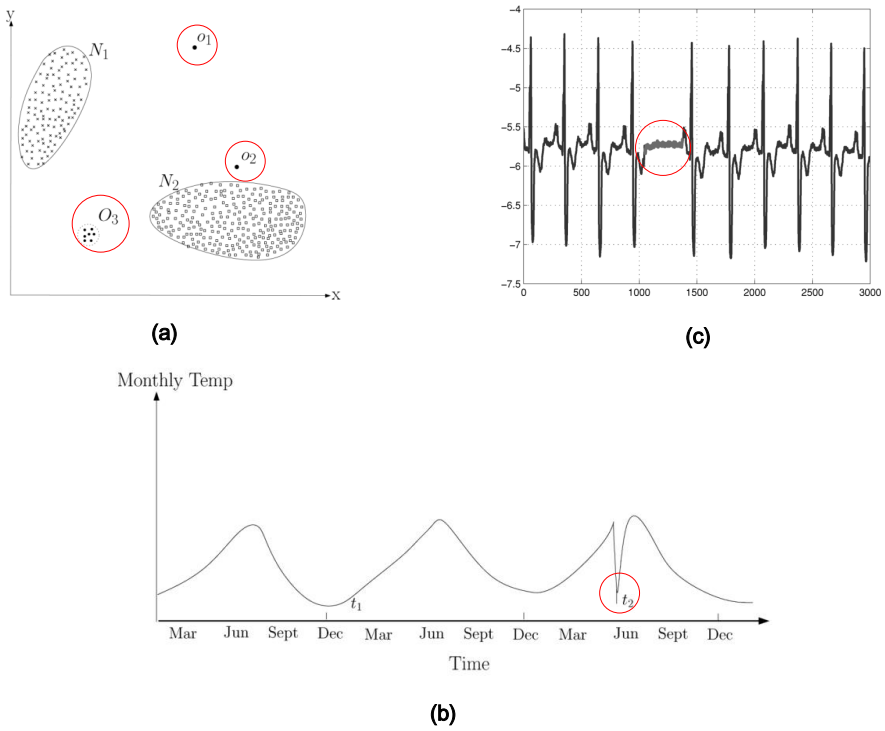


Figure 2.7: Anomalies Classification (adopted from [5]): (a) Point Anomalies; (b) Contextual Anomalies; (c) Collective Anomalies.

taken as a whole represents an anomaly. Figure 2.7.(c) illustrates a example of the collective anomalies where the data in the highlighted area is anomalous since it contains normal inconsistent low values.

Classifying the normal and abnormal behavior is complicated by a number of factors [5]:

- It maybe infeasible to define all the normal behaviors due to the system or application complexity;
- Normal behaviors may evolve overtime;
- Behaviors can be masked to make the anomalous observations appear normal;
- The boundary or scope of an anomaly may differ among domains.

Numerous solutions have been proposed to bypass these issues and provide usable and reliable anomaly detection for the computer systems. They generally consist of activities: Input Data, Detection Technique, and Output Result [5].

1. Input Data: A collection of data instances is used to generate a baseline profile of anomalies. The input data has two main attributes:

    - Natural data type. There are different types of input data, *e.g.,* sequence data, spatial data, and graph data. Understanding how input data is consumed shapes the creation of anomalies and effects the accuracy of the whole method.

    - Data Label. The input data can be pre-processed manually by labeling each of the data instances. If both normal and abnormal classes are labeled, then the detection is supervised detection. If only normal instances are labeled, then it is semi-supervised. If no data is labeled, then it becomes unsupervised. Obtaining labeled data that representative of all types of behaviors is expensive but it is necessary and required, since an unsupervised detection typically suffers from a high false alarm rate.

2. Detection Technique: Using input data as training data, the actual detection techniques are the approaches that can classify and detect the anomalies in the runtime (online) data. Common examples of detection techniques include:

    - Classification-based: Classification-based approaches use labeled input data to generate a classifier categorizes data having similar properties. Notable example techniques are Neural Networks, Bayesian Networks, and Support Vector Machines. These techniques are effective, yet need accurately labeled data for training.

    - Clustering-based or cluster-based techniques are a form of unsupervised detection which groups data based on similarity. Clustering logic varies. Normal data may belong to clusters, whereas anomalies do not fall outside clusters; normal data is close to the cluster centroid while anomalies are far away from the centroid; or normal data belongs to large and high density clusters and anomalies lie in either small or sparse clusters. Clustering-based methods have the advantages of working in unsupervised mode, but performance is highly dependent on what clustering algorithm is deployed and how the data is structured.

- Statistical-based: Statistical techniques identify anomalies by assuming that normal data occurs in high probability region while anomalies occur in low probability regions. These techniques rely on the assumption that the data conforms to a particular distribution. If the assumption is true, statistical based methods can provide a confidence interval that can be used to provide additional information for making decisions. Absent a distribution, which is often the case for high dimensional datasets, these techniques are not effective. Some well-known methods are Gaussian Model and Regression Model.
- Information Theory based: Information theory contends that anomalies manifest on irregularities in the information content of the data. Detection is carried out by analyzing information content of data using techniques such as: Kolomogorov Complexity, entropy, and relative entropy.

3. Output Result: A label or score is output as a result of the detection technique. The label can be as simple as an indicator that each data instance is normal or abnormal, or an anomaly score can be assigned to each data instance to show the degree to which has been deemed abnormal.

An anomaly detection system usually goes through an initial offline training procedure that includes above three main activities before it can be used to handle real data. Over time, the detection method needs to be revised and improved by retraining to improve accuracy. It could be nontrivial to provide a comprehensive and elaborate training set, since the initial definition of normal region and the labeling of data are challenging and expensive. Inadequate offline training, on the other hand, comes at a price of inaccurate result.

2.3.2 Virtual Machine Introspection

The ultimate goal of computer system monitoring is to have a comprehensive view of the system, including the monitor system itself. Virtualization assists with this because each VM is fully managed by the hypervisor. Monitoring or detection approaches can take place using

virtual machine introspection (VMI), meaning "*inspecting a virtual machine from the outside for the purpose of analyzing the software running inside it*" [19].

VMI works in one of two ways: from the VM or outside the VM. An in-VM monitoring is similar to HIDS (Host-based Intrusion detection, see Section 2.2.3), since the monitoring module is inside the monitored system. It extracts the OS and process states natively, providing fast and effective monitoring with rich abstractions. Its fundamental limitation is that it can be attacked because it is executing at the same privilege level as the system it is protecting. The predefined policies/profiles and the security enforcing mechanism can be tampered with by malwares, rootkits, and/or attacks. The out-of-VM monitoring, by contrast, is more tamper resistant, since the monitoring module is located outside the VM and is running on a separate level. The out-of-VM approach is easy to deploy because it eliminates the need of installation monitoring modules inside each VMs. It provides a more complete and elaborate view of the monitored VMs than in-VM based approaches because the hypervisor has full access to all of the memory, register, and disk state.

The primary advantage of in-VM systems is their direct access to OS level abstractions. Out-of-VM solutions offer reliability and accessibility at the price of losing the rich semantic abstractions inside the OS. They have to reconstruct information based on the access of raw data provided through the hypervisor. This results in a semantic gap. To effectively monitor behavior, we need the OS state, resource usage, and kernel events in a human understandable form, thus, the out-of-VM solutions have to bridge the semantic gap. In-VM and out-of-VM based solution can still use the same underlying detection approaches with intrusion detection (anomaly detection and misuse detection), so we will not focus on the in-VM based solutions since they are out-performed by the out-of-VM approaches.

Approaches to bridge the semantic gap fall into five major categories [46]: Manual, Debugger-Assisted, Compiler-Assisted, Binary Analysis-Assisted, and Guest-Assisted.

1. Manual Approach: This approach relies on knowing advance the structure of the guest OS and filling in the structure with data from hypervisor. This is labor-intensive and error prone because it depends on manual analysis of the documentation, debug information, source code, or binary code of a guest OS. For example, the Linux kernel task structure

(*i.e.,* task_struct) is organized in a double-linked list that contains all running processes. We can retrieve all running processes in a system by traversing the list. The limitation of this approach is that manual interpretation is needed for every single data structure and every different OS.

2. Debugger-Assisted Approach: Instead of manually analyzing data structures and translating the information, a debugger-assisted approach uses data structure knowledge available in debugging information. The semantic gap is thus bridged by leveraging debuggers. The debugger becomes a analysis mechanism using kernel dumps or live memory to derive the needed abstractions. The limitation of this approach is similar to manual approach: using debugger is specific to a particular OS. Moreover, the monitored guest OS needs to be recompiled so that it produces debug information, which has an effort on system performance.

3. Compiler-Assisted Approach: In a compiler-assisted approach, the compiler is used to automate the process of finding abstractions in the source code of the guest kernel. For example, the kernel integrity can be dynamically monitored by extracting global variables and creating a graph of kernel variable types from the kernel source code [47]. While this allows monitoring mechanisms to be tailored to specific needs, it also requires that kernel modules be recompiled or modified.

4. Binary Analysis-Assisted Approach: Kernel source code is not always available, so the previous solutions may not work. The Binary Analysis-Assisted Approach is a sophisticated, yet practical solution, that only uses the compiled binary of the guest OS. It is a dynamic analysis approach that dynamically access and analyze registers, memory and instructions to generate the final abstraction. This analysis procedure can be done offline or online. An offline analysis is based on inspecting the kernel state of a trusted OS and then a training VMI tool provides the monitoring. An online analysis, sets up the abstraction by dynamically retrieving information from a live monitored system.

5. Guest-Assisted Approach: The Guest-Assisted Approach retains the hypervisor-level view of the guest OS and avoids the semantic gap problem by waiving part of the security advantages. This approach installs an agent program inside the guest OS from which the hypervisor can retrieve desired information. The advantages of the Guest-Assisted approach is that the internal agent can be customized to yield useful information, but that agent suffers from being a potential security vulnerability.

VMI has matured as a research area since the first VMI based solution is proposed in 2003. Solutions in the early stage were limited to certain versions of hypervisor or kernel, until the binary analysis approach emerged in 2011 [46]. At that point, VMI based monitoring become more flexible, effective, and practical. The capability of VMI systems is extended beyond detection and pioneered attack prevention and recovery functionality. By leveraging the power of VMI, we can reduce the need of human intervention whenever there is a problem [46]. Further discovery and research on methods or mechanisms to monitor, filter, and analyze the interaction between the virtualized host and the underlying virtual or physical hardware is still needed [48]. Beyond the semantic gap problem, more work is needed to effectively and accurately monitor the VM's internal state, which returns to the initial research question of detection or monitoring, that is what to monitor, how to build a profile of what to monitor and how to refine monitoring overtime.

2.3.3 Machine Learning based VMI

Recently, training based monitoring has become a popular method because of its efficiency and automated features. Many intelligent monitoring approaches and works are briefly introduced in [5, 46, 49, 50], however, here we only interested in the intelligent virtual machine introspection solutions.

Virtuoso [6] is the first training based introspection solution proposed in 2011, this project aims to migrate Linux native command to hypervisor level (such like *ps, lsmod, etc.*). By deploying an agent program inside the monitored guest system, the instruction sequence traces can be obtained to train out a common VMI tool which enables the capability of performing introspection on the hypervisor level. As the first training based methods, Virtuoso depicted

the feasibility of automatic introspection, however, it needs an reliable offline training procedure and the training itself based on the understanding of the underlying security information of the specific application. And the introspection only focuses on the system calls. Moreover, the generated VMI tool indeed bridge the semantic gap and also can provide meaningful information at hypervisor level, but it still need the human-interaction to understand and use the information.

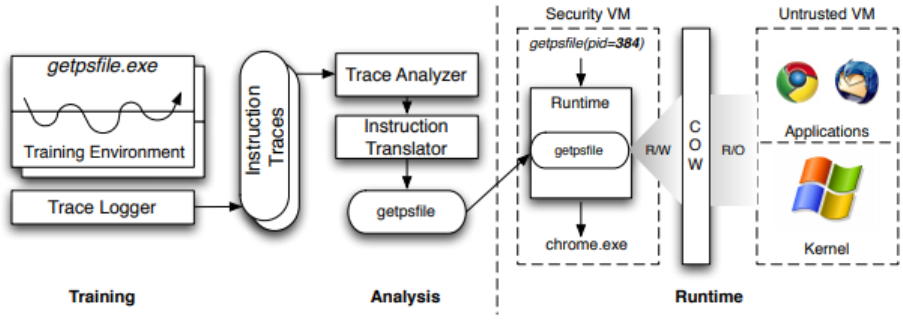


Figure 2.8: An Example of Virtuoso Usage (Adopted from [6])

Another follow up work proposed by the same authors of Virtuoso is the Tappan Zee Bridge (TZB) [51], which introduces the memory access mining into the picture. Instead of only focusing on the system calls, by examining the memory space of the guest system, *tap points* can be found out to provide active monitoring of applications or operating systems. A *tap point* is a point in a system used to capture a series of memory accesses for introspection purposes. In the TZB system, the trained program is able to automatically locate the candidate *tap points*, and after further information processing, the user could be notified when certain system events occur. Generally, the TZB already provides a baseline intelligent monitoring, though the searching, clustering and using of *tap point* still needs some manual examinations. TZB may also confront the overhead limitations, since it keeps tracking an arbitrary number of callers for each process to extract the callstack information.

Combining the concepts of Virtuoso and TZB is promising, it would be ideal if an program can be trained out to examine both system calls and memory forensics on a live system. However, even the combined system would be constructed, the problem is still not fully solved. The

requirements of the monitoring we need is multitude: First, the semantic gap needs to be automatically bridged; second, the monitor must automatically extract the needed information from the system; and third, the system needs to provide an intelligent anomaly detection that does not need the user to have a understanding of anomaly. Though many other works like Virtuoso and TZB have been proposed in the past few years and some open source VMI frameworks like LibVMI [52] are also available, the latter two problems are not completely solved.

It is non-trivial and meaningful to enable the capability of automatic semantic bridging, like by using LibVMI where all the system calls and memory space activities can be observed. However, it is not practical and nor necessary to keep monitoring everything, since the overhead and processing delay cannot be tolerated. Therefore, the problem is twofold: First, the data should be extracted and translated automatically and efficiently. Second, it is also needed to perform a monitoring metrics selection to achieve better accuracy and performance.

2.3.4 Container Security

Containerization is a lightweight form of virtualization with thinner isolation and arguably weaker security. Container introspection and monitoring is a relatively new research area and the security of the most of container-based systems relies on the security mechanisms provided by the specific container implementation. The underlying mechanisms that provide isolation for containerization, such as *cgroup* and *Chroot,* also enable a baseline security guarantee. Since the majority of containers are based on Linux systems, some of the Linux kernel features are also available to help enhance the container security. One notable Linux features is the secure computing (*seccomp*), which is designed to eliminate system calls from a contained application. This allows software deployers to reduce the footprint of Linux kernel through software and tailor it to each deployed application. Seccomp uses a whitelist, allowing users to list only system calls permitted for a specific application. It bars applications access to knowingly flawed and obsolete Linux features and helps ensure that applications only use system calls that are needed to operate [53].

Containerization also supports multiple forms of mandatory access control which are provided by the native Linux system. Notable examples are SELinux [54] and AppArmor [55].

Originally, Linux system used standard Discretionary Access Control (DAC) to manage the ownership and permission settings of an object. The DAC mechanism allowed users to have full control of their objects. This gave an attacker access and control all the owner's objects, if compromised. SELinux countered this problem by providing an additional layer of permission checking by labeling objects. SELinux is an enhancement to the security Linux kernel module which defines the access policies for users, applications, and objects. If a process or user attempts to access an object, SELinux co-operates with the kernel to check the access vector cache (AVC) where policies are defined and stored [13]. AppArmor is also a mandatory access control solution similar to SELinux, however, it restricts its scope to individual applications. Each application has an associated policy profile which limits the capability of this application. The profile defines the files and resource that the application could access and use. AppArmor can run in enforcement mode or complain mode. The former enforces policy and reports violation, whereas only logs the violation [16].

Other built-in monitoring modules are available in cloud management systems, such as, Observability Metrics in Mesos and Ceilometer in Openstack [56,57]. Practical out-of-container monitoring has been proposed and achieved by many other researchers and organizations [58–61]. Technically, container introspection is not the same as VMI. In containerization, each container still shares the same kernel with the host OS, which means that monitoring a container, if done outside the container, is just monitoring a part of the host OS. In short, there is no actual semantic gap which makes container monitoring of container more practical or effective than traditional VMI. On the contrary, container monitoring needs to take into account the kernel exploits (privilege escalation) because the shared kernel not only provides convenience but also potential risks.

2.4 Machine Learning Based Classification

Machine Learning [62] , Neural Networks (NN) [63], and Deep Learning [64] contribute significantly to solutions that involve classification of data. A number of well-known methods such like Support Vector Machine (SVM) [65], Nearest Neighbours [66], Naive Bayes [67], and

Linear Discriminant Analysis [68] have been applied in many solutions. This section briefly introduce the fundamentals of Machine Learning based classification techniques.

2.4.1 Fundamentals

Machine learning is a computational technique that optimizes or solves a problem by using example data or past experience [69]. Machine learning relies on the theory of statistics to build a mathematical model for making interference. The learning of the data and the procedure of the model building are known as training. Training falls into two categories: supervised and unsupervised. Supervised learning techniques use known correct and incorrect samples as training, whereas unsupervised learning techniques aim to discover relationships and regularities without knowing in advance the nature of the data. Supervised learning is, thus, similar to calculating a statistical regression and unsupervised learning is akin to density estimation [69].

Machine learning can traced back to the early 1950s. Alan Turing proposed his "learning machine" in 1950 and the first neural network machine was introduced by Marvin Minsky in 1951 [70]. Although research suffered from "AI Winters" *, the enthusiasm in machine learning never died. Decades of development and investigation have matured machine learning to the pint where it has been applied in speech recognition, program generation, auto-driving, data mining, and etc.

2.4.2 General Classifiers and Algorithms

Several general categories of machine learning are applicable to anomaly detection: Support vector machine (SVM), K Nearest Neighbors (KNN), General Regression Neural Network (GRNN), and etc.

Support Vector Machines (SVM)

Support vector machine (SVM) were originally designed for binary classification [72]. The goal being to find a suitable hyperplane which separates the training data into two categories. Because not all the real-world problems can be simplified into two class problems, multi-class

*"An AI winter is a period of reduced funding and interest in artificial intelligence research, which first appeared in 1984 as the topic of a public debate at the annual meeting of AAAI [71].

classification usually required. The most widely known and applied schemes used to transform binary SVM into multi-class forms are one-vs-all and one-vs-one [73]. The one-vs-all method generates a SVM sub-model for each class, where each of model labels the samples in current class as positive and labels all the other samples as negative. The final decision is made by passing the sample into all the models and selecting the one with highest confidence score. The one-vs-one scheme creates $\frac{k(k-1)}{2}$ binary models, where k is the number of classes. In this case, each of the sub-model classifies a pair of the classes of the training set. A sample is passed into all the trained sub-models with the class with most votes picked as the final decision.

K Nearest Neighbors (KNN)

KNN classification is a fundamental and well-known classification method, having been first mentioned in an unpublished US Air Force School of Aviation Medicine report in 1951 [74]. After several decades development, the native KNN follows a straight-forward strategy of identifying items in a data set that are closest to a specific instance, where "closest" is qualified by a measure such as Euclidean Distance [75]. Variety of refinements on KNN have been proposed: new rejection approaches, distance weighted approaches, soft computing methods and fuzzy methods [74].

General Regression Neural Network (GRNN)

GRNN is a variation of the Radial Based Neural Network (RBNN) proposed by Specht in 1991 [76]. It is a one-pass algorithm with a highly parallel structure and is suitable for a wide range of the regression problems.

A general mathematical expression of the GRNN is described as following:

$$\mathbf{d_q} = \frac{\sum_{i=1}^{N} hf_i(t_q, t_i) d_i}{\sum_{i=1}^{N} hf_i(t_q, t_i)}.$$

$$\mathbf{hf_i}(\mathbf{t_q}, \mathbf{t_i}) = e^{-\frac{||t_q - t_i||^2}{2\sigma^2}}.$$

Gaussian kernel function hf with Squared Euclidean Distance metrics is used to perform the prediction (other distance metrics can be used as well). The d_q is the predicted output vector

of data instance t_q, where the t_i and d_i are the training instance and its corresponding desired output.

Long Short Term Memory (LSTM)

Recurrent neural networks (RNN) with long short-term memory (LSTM) have emerged as an effective and scalable model for several learning problems related to sequential data [77]. LSTM is a variant of RNN which generally solves the problem of input memorization. Unlike human-being, it is hard for machines to understand a sentence or predict a following word while it needs a long-term memorization of previous inputs.

LSTM achieves long-term input tendencies by dividing its neural network cells into layers: forget gate layer, input gate layer, and output gate layer. Upon a new input is received by the neural cell, forget layer decides the information needs to be removed from the cell, input gate layer decides what could be added into cell and update cell state, and finally a output gate layer output the result based on the current cell state.

2.4.3 Unsupervised Anomaly Detection and Intelligent VMI

Traditional anomaly detection and intrusion detection systems require either certain prior knowledge of the system, or identical signatures of previously seen attacks, thus the data used for classification profile generation can be labeled. This labeling procedure is expensive and not always feasible. Detecting the unknown without knowledge is a challenging task. Several unsupervised detection systems have been proposed and evaluated in the past decades. Casas et al. [78] introduces an unsupervised intrusion detection system with using clustering techniques. The authors proved that even without any knowledge, the unsupervised detection system can still achieve satisfied detection accuracy on KDD99 dataset [20]. Amer et al. [79] leverages one class support vector machine (SVM) to achieve unsupervised anomaly detection, and the proposed enhanced SVM outperforms the clustering-based methods. In addition, neural network based auto-encoder is used in unsupervised anomaly detection on KPI dataset [80], which outperforms the traditional supervised methods. Self Organizing Maps (SOM) [81] is used in [82]

for unsupervised detection, SOM ensures the traces of the transformation from benign to malicious to be recorded, interpreted, and further analyzed. Dromard et al. [83] use a grid clustering algorithm combines with the concept of sliding window, which provides detailed prediction and reliable classification on the streamed data. Hierarchical Temporal Memory (HTM) algorithm is used in [84] for robust unsupervised detection on streaming data, which can memorize the data over time and provide reliable classification for collection anomalies.

Different unsupervised techniques such as clustering algorithms, one class support vector machine (SVM), neural network based auto-encoder, Self Organizing Maps (SOM), and Hierarchical Temporal Memory (HTM) algorithm have been all investigated for anomaly detection [78–80, 82–84]. The integration of unsupervised learning algorithms and virtual machine introspection techniques have also been investigated. Tappan Zee Bridge (TZB) [85] monitors the memory space of the guest system and uses *tap points* (points in a system used to capture a series of memory accesses for introspection purposes) to provide active monitoring of applications or operating systems. TZB employs an unsupervised clustering-based trained program to automatically locate the candidate *tap points*, and after further information processing, the user could be notified when certain system events occur. TZB already provides a baseline intelligent monitoring, though the searching, clustering, and using of tap point still needs some manual examinations. Li et al. [86] proposed to use SOM to characterize the malicious behaviors of VMs, and *strace* is used to provide system call trace dumps. Although the proposed framework can detect specific attacks, the experiments were performed on the Windows NT systems using out-of-date attack approaches. EyeCloud [87] uses unsupervised clustering algorithms with LibVMI [52] and Volatility framework to perform botcloud detection. The violation of security rules and modification of security configurations can be detected by the EyeCloud. The proposed system works in a misuse detection style, which is tailored to prevent the adversary from abusing cloud resources.

For the containerization perspective, researchers introduce several prototype anomaly detection frameworks [88–91] but the majority of these methods still use supervised algorithms or statistical models to classify the behaviors. Although a Hierarchical Hidden Markov Model [90] based approach achieves the anomaly detection in a semi-unsupervised manner, the framework

only works for overload detection and the general anomaly detection beyond the resource management cannot be accomplished. The comprehensive and unsupervised VMI still needs further investigation, and the introspection of containerization is yet in its infancy.

2.5 Blockchain

Blockchain was first introduced by Bitcoin [92] and is now widely used by the cryptocurrencies. Blockchain is known as a distributed and shared digital ledger, where all the transactions and records are hashed and stored in the chain to provide both integrity and transparency. Certain blockchains also support a concept known as smart contract [93,94], allowing the user to run Turing-complete scripts on the chain. Using a smart contract (also known as chaincode in HyperLedger) enables the user to store and manage data inside of the blockchain. Various applications such as Filecoins [95] and Storj [96] have been proposed.

2.5.1 Basic Concepts

The basic idea of blockchain is to collaborate on recording information in such a way that the participants in the network do not trust each other [92]. A public, auditable ledger helps provide verification and accountability for data [97]. Whenever an entity initiates a transaction, a group of "volunteer recorders" start to write down this new transaction in their individual local ledger. And after a period of time, a selected ledger from these recorders, which contains a set of transactions, will be verified and attached to the public ledger [98]. The recorders are known as miners in the blockchain systems. The local ledgers owned by the recorders are known as data blocks. The algorithm used to select a block to attach to the main public ledger is called a consensus algorithm, and the public ledger formed by these selected data blocks is called the blockchain [98].

This whole procedure is protected by digital signatures; that is, each transaction is digitally signed using the private key of the sender [92]. As a result, the validity and integrity of a single transaction are guaranteed. The integrity of the entire blockchain is ensured with hash computations. To encourage miners, coins are assigned to them as a reward once they successfully append a block to the ledger [99].

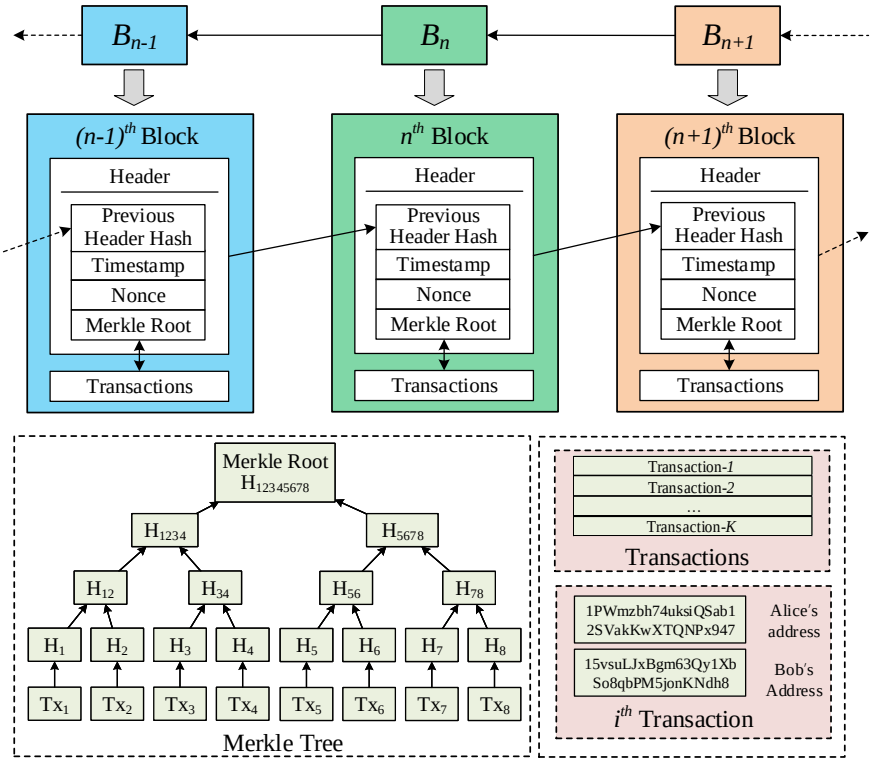


Figure 2.9: Blockchain Architecture (Based on Bitcoin)

To initiate a transaction, the participant needs to associate the transaction with the cryptography credentials. Each participant in the blockchain network holds a pair of public and private keys. By applying a series of hashing and encoding functions on the public key, a short and unique address is generated as the public address of the participant (as depicted in Figure 2.9). Normally, the transaction contains the address of the sender and receiver, and is signed by the sender's private key. Once a participant initiates a transaction, this participant needs to broadcast the transaction to its connected peers, and the peers that receive this transaction will continuously relay the transaction to subsequent peers [100]. Each transaction and the block that contains all transactions are verified by the miners, and a valid transaction is not approved and executed until specific requirements are satisfied (Requirements vary depending on the blockchain. Examples include computation of a certain amount of proof of work in Bitcoin, the transaction is approved until 6 blocks are appended which is a countermeasure to the double spending) [101].

In order to ensure fairness and security of the transaction appending in the blockchain system, the consensus algorithms are proposed and applied. The consensus algorithm is a required verification step for adding transaction records to the public ledger. As mentioned above, a new generated block is actually a selected ledger from all the miners; thus, a mechanism is needed to make all the miners reach the consensus about the selection. Key to this process is be unbiased. In an ideal model, every miner's ledger (in the processing block) should have the same chance to be selected. However, one could run multiple nodes to increase the possibility to be picked in the case of random selection. An attacker that controls a large number of nodes can practically manipulate the selection procedure. Some notable consensus algorithms are:

- *Proof-of-Work (PoW):* In the PoW system, all miners compete to finish a computational resource–intensive task, and the first one solves the task can append the current block to the public ledger. For example, the PoW in the Bitcoin system requires finding a nonce such that the hash of the current block header (including the nonce) is less than a specific value.

- *Proof-of-Stake (PoS):* The PoS is an alternative consensus algorithm that requires less computational power than the PoW does. Instead of proving a certain amount of work has been done, miners must prove ownership of a certain amount of stake in the blockchain system.

- *Byzantine Fault Tolerance (BFT):* The Byzantine Generals Problem [102] describes a conceptual situation that needs to reach a consensus among several generals. And the solutions of the BGP can be used as the consensus algorithm in blockchain. For instance, Practical Byzantine Fault Tolerance (PBFT) [103] can be used in blockchain, where each new block is selected when it is supported by more than 2/3 of nodes.

The development and evolution of blockchain never stops, and now several variants of blockchain have been proposed. The new blockchains have various structures and attractive features, ensuring that the fundamental concept will endure for sometime to come.

2.5.2 Virtualization and Containerization in Blockchain

Certain blockchain implementations support Turing-complete scripts running on the chain, namely the smart contract. All the nodes run a particular script and obtain a same output regardless what operating system and platform are using in these nodes. This capability relies on the use of virtualization. The blockchain virtualization implementation styles can be generally divided into two types: application-level virtualization based, and containerization based.

Well-known application-level virtualization examples are Java Virtual Machine (JVM) and Lua Virtual Machine. This type of virtual machine runs as a normal application inside a host OS to provide a platform independent runtime for programming languages. Similar to JVM, Ethereum implements its own Ethereum Virtual Machine (EVM), which is also a stack based virtual machine. EVM has its own instruction set and opcodes (*e.g.,* MOV, ADD, MUL, etc), and each smart contract is executed in its own instance of EVM [104].

Containerization is used as another style in Hyperledger. Instead of using a application-level virtual machine, all the services (orderer, chaincode execution, data recording) of Hyperledger are run in Docker containers. Each chaincode also has its own instance of container. More details are elaborated in Chapter 5.

2.5.3 Smart Contract Security

A "smart contract" is a computation that is performed on a blockchain. The term is an oblique reference to the traditional notion of a legal contract in that it signifies signatories entering into some binding agreement regarding something. "Smart" signifies that software automatically triggered by the agreement carries out a series of actions that define the terms of the agreement; "contract" signifies that the results of the actions are recorded onto an indelible transaction ledger, such as a blockchain. The transactions themselves, once stored onto a blockchain, are considered, for the most part, secure. Executing the smart contract, on the other hand, raises questions. How open to vulnerabilities is the "smart" part of "smart contracts"?

Several attacks and flaws in smart contract systems have been observed and reported in the past a few years. The majority of the attacks used the internal programming vulnerabilities in

smart contracts, such as call to the unknown, exception disorder, re-entrancy, and etc [105,106]. One of the most infamous attacks was on a decentralized autonomous organization (DAO) platform [107] using the unchecked fallback functions in the underlying infrastructure. The vulnerability allowed attackers to recursively withdraw coins from the smart contract in a single transaction, enabling the attacker to steal the majority of the cryptocoins. Other attacks exploited the gas mechanism of Ethereum. By only including a small amount of gas in a refund transaction, the transaction can be intentionally made to fail [108]. A similar exploit involved continuously storing data in the array of the smart contract so as to increase the cost of contract storage operations to the point where normal operations of the smart contract fail [105, 106].

Intentional backdoors in smart contract can also cause tremendous financial losses. A trade between Australian company Byte Power Party and Singaporean company Soar Labs was tricked by the Soar Labs in 2017. Soar Lab purchased a stake of Byte Power Party via a smart contract. However, the payment of $6.6 million worth of Soarcoins reported to be withdrawn from the contract by a backdoor after the stake has been transferred [109]. Other reported backdoors allow the caller to generate and destroy tokens after the Initial Coin Offering (ICO) stage [110]. Because the total number of token should be considered as fixed after the ICO process, thus attackers can manipulate the market price of this particular token and/ or gain profits from it.

From one point of view, the security of a smart contract significantly relies on how formal and secure the contract has been programmed. Since the smart contract is designed to be a public application, any internal programming vulnerabilities can incur enormous influence on all the contract users. Therefore, several evaluation and verification frameworks at the programming level [111–117] have been proposed. These frameworks and tools evaluate the validity and security of the smart contracts at the programming level by creating certain rules and boundaries for smart contract programming. With the examination of smart contract code context, the smart contract can be converted, compiled, and regulated to a secure form. These frameworks focus on the security of the smart contract only in the Ethereum platform [93]. While there exists another platform, Hyperledger [94], which provides competitive smart contract functionalities, no similar smart contract security investigation is evident.

Chapter 3

Proof of Concept for Intelligent Monitoring

Before further investigation on the proposed method, we implemented a proof-of-concept prototype to evaluate the feasibility of the proposed intelligent monitoring. A sample program was created as the data source. It was able to perform both predefined normal and abnormal behaviors. All the behaviors were captured by a monitoring framework and analyzed. The normal behaviors were defined as sequence of operations allowed by the system *e.g.,* read specific file, communicate with specific server, running in a specific usage range. The abnormal behaviors were defined as:

- *Abnormal System call:* A single system call which is not allowed. *e.g.,* SSH is not allowed in the container.
- *Abnormal System call parameters:* A set of abnormal parameters that be used in a valid system call. *e.g.,* Read sensitive directory.
- *Abnormal Collection of System Calls:* A illegitimate or malicious collection of system calls which is consist of a specific sequence of valid system calls.
- *Abnormal Resource Usage:* Abnormal memory and cpu usage spikes.

The sample program was packaged into a container and run as the only non-kernel procedure when the container is booted. The details of the sample program is shown below:

Sample Program Functionality

- **Normal**: *(i)* Read a file in location L_1. *(ii)* Access google.com. *(iii)* Simple integer addition.
- **Abnormal**: *(i)* Access Bing.com *(ii)* Access google.com 10 times. *(iii)* Chmod a file. *(iv)* Read a file in location L_2. *(v)* Raising usage (By using hash matching).
- **Random**: Randomly performs normal or abnormal behaviors.

3.1 Out-of-Box Introspection Framework

We used Sysdig [61] as the monitoring tool to provide comprehensive and real-time container monitoring. Sysdig is an open source container monitoring framework, which provides detailed real-time information of the target container, such as system calls, I/O usage, and network traces.

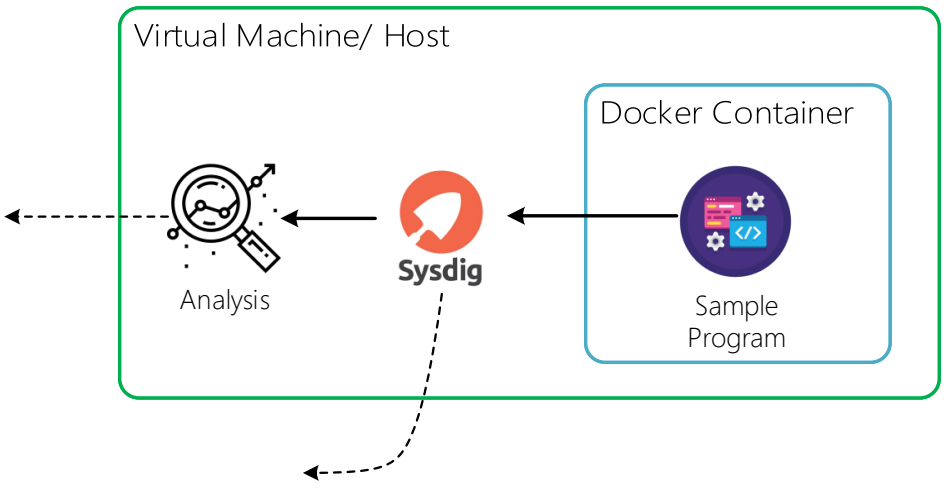


Figure 3.1: Sample Design of the Prototpye

In this proof of concept system, we used Sysdig to capture only system call traces, which provided the evidence of network communications and I/O requests. Although the raw data captured by the Sysdig is well-organized, readable and easy to understand, it includes unrelated

and unnecessary information (see Figure 3.2). We felt, it would be non-trivial and impractical to use the entire raw trace as the training sample in the later learning procedure. Instead, we extracted the most significant metrics.

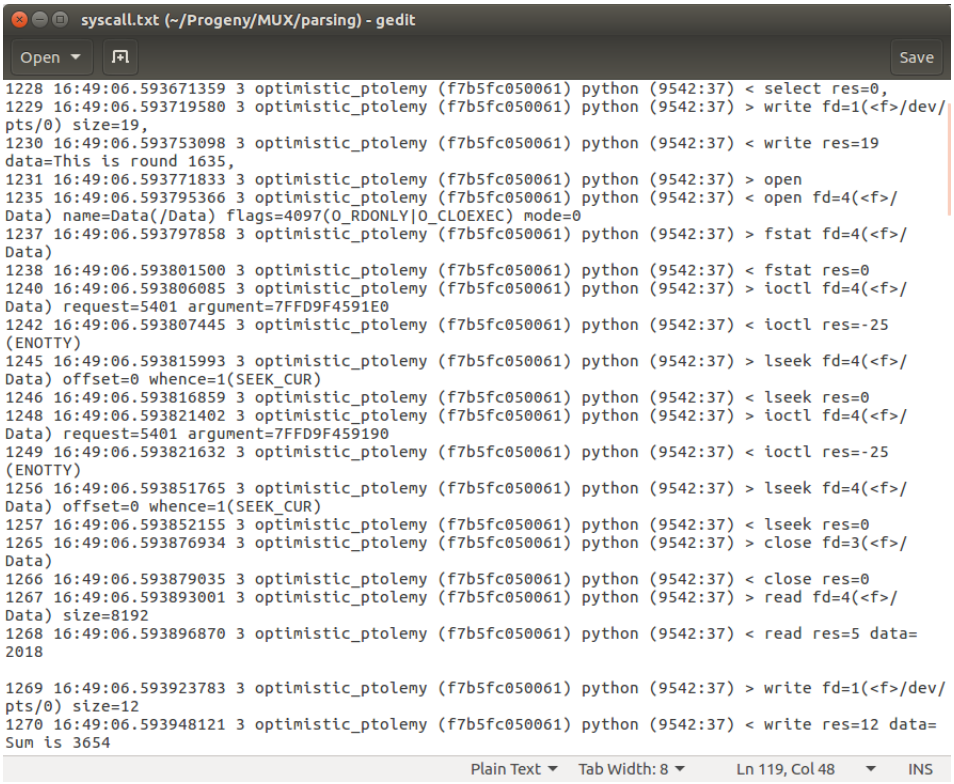


Figure 3.2: System Call Traces Captured by Sysdig

3.2 Feature Selection

Figure 3.2 shows that the raw traces from Sysdig contains useful metrics such as count, timestamp, container name, process name, process ID, system call name and system call parameters. The system call names are the most valuable metric because they directly indicate the type and goal of a single operation. They should be always a selected feature in the training procedure. In addition. The timestamp of the system call is useful as well because it relays information on sequence and relationship to other system calls. Therefore, it is also reasonable to include timestamp as a feature. As the sample program is a single process application, the process name and process ID are not relevant to the behavior classification, so they are not be used in the later training.

It is questionable whether to include the system call parameters into the training. The parameters contain unpredictable content, and introducing them as a training metric is expensive and challenging. A certain way of representing all the possible parameters is needed, while the specific knowledge of the application is assumed to be missing. There are many ways of achieving such purpose, and we use the unigram to perform the the system call parameters transformation. The used 95 ungrams are shown in Figure 3.3. Each system call is transformed to a frequency array based on this unigram transformer.

(space)	!	“	#	$	%	&	‘	(	)
*	+	,	-	.	/	0	1	2	3
4	5	6	7	8	9	:	;	<	=
>	?	@	A	B	C	D	E	F	G
H	I	J	K	L	M	N	O	P	Q
R	S	T	U	V	W	X	Y	Z	[
\	]	^	_	`	a	b	c	d	e
f	g	h	i	j	k	l	m	n	o
p	q	r	s	t	u	v	w	x	y
z	{	|	}	~					

Figure 3.3: The Unigrams used in System Call Parameter Transformation

3.3 LSTM based Core Classifier

Even when features are selected and the data is well preprocessed, may not be possible to label a single system call as being benign or malicious. Determining intent is more likely to require observing a number of system calls that, collectively, reveal benign or malicious behavior. The classifier must track the internal relationships and overall history of sequential data.

We used an LSTM based model as the core classifier of the prototype system to maintain the knowledge in the sequential system call data flows. The sample model is shown in the Figure 3.4.

```
def main():
    #n_timesteps = 10
    X, Y, X_test, Y_test= dataCreator(path)
    print(np.shape(X))
    print(np.shape(Y))
    # define LSTM
    # create the model
    model = Sequential()
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    #print(model.summary())
    model.fit(X, Y, epochs=100, batch_size=8)
    accr = model.evaluate(X_test, Y_test)
    print('Test set\n  Loss: {:0.3f}\n  Accuracy: {:0.3f}'.format(accr[0], accr[1]))
    yhat = model.predict_classes(X_test, verbose=0)
    for i in range(len(X_test)):
        print('Expected:', Y_test[i], 'Predicted', yhat[i])
```

Figure 3.4: Sample LSTM Model

3.4 Data Collection

All the data in this proof of concept system was generated from the python sample program running in a Ubuntu docker container launched in a Ubuntu desktop. Table 3.1 describes the environment. A series of python and shell scripts processed the raw data by translating system call names to numerical form by using Linux system call references. A sample result is shown in the Figure 3.5. Only translated system call names, timestamps, and system call parameters were included in the processed file. Each line in this file, is referred to system call trace.

We collected 300 files, with each files containing 10 second trace data obtained from the running sample program. 150 files were collected by running the sample program with pre-defined benign behaviours, and the rest collected from the malicious behaviors.

Name	Specification
Host Machine	Ubuntu 16.04 System with 2.20Ghz Xeon E5-2650 cpu and 64GB memory.
Container	Ubuntu 16.04 Docker Container
Sample Program	A program written in Python 2.7 with 277 SLOC.
Monitor	Sysdig Monitor. Sample commands: sysdig -M 10 -pc container.name=name >> file-name.txt

Table 3.1: Implementation Specification

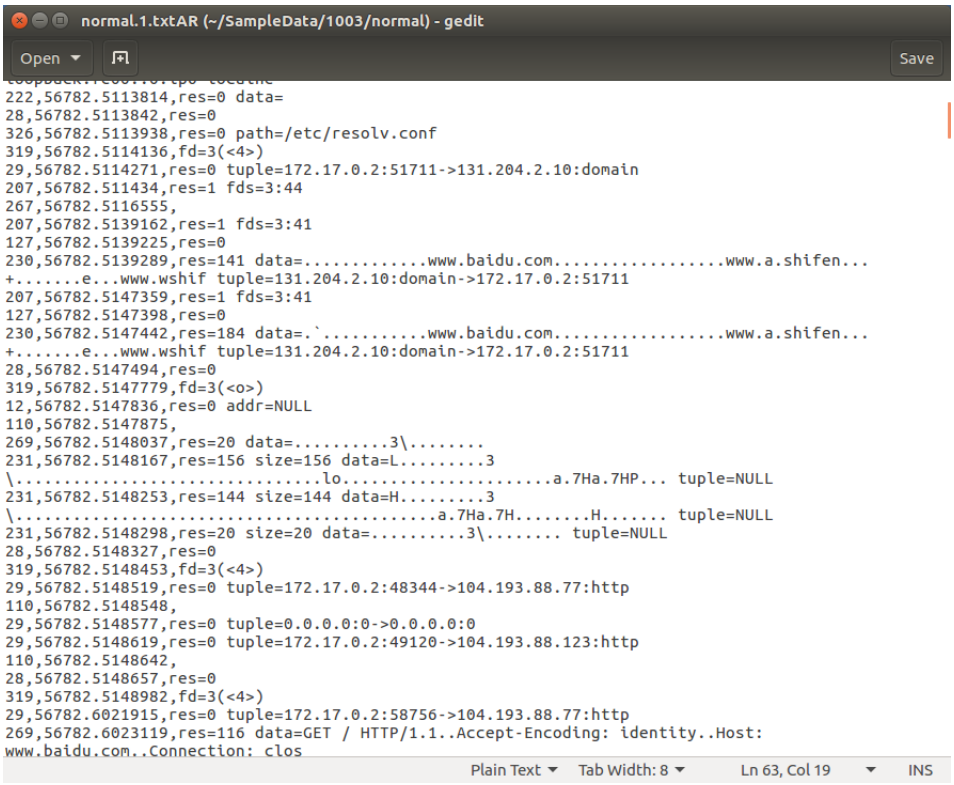


Figure 3.5: Pre-processed Raw Data

3.5 SVM based Labeling

The ultimate goal of the framework is to provide secure monitoring around a black-box application. We used knowledge of the sample program to demonstrate the feasibility of differentiation normal from abnormal behavior, however, we recognize such a priori knowledge is not always available in real world scenarios. A automatic mechanism of labeling the collected data must be implemented. We propose to use a SVM based labeling system.

Instead of labeling the system call with only labels, we assign each of the system call an additional confidence score. A statistical model would be created first based on the collected data. Each of the system calls would be labeled based on this model (calls that fall within the model thresholds would be deemed "nomral" and calls falling outside the thresholds as "abnormal"). The detailed procedure is described in Figure 3.6.

Assume a black-box application is running in a secure and trusted environment while the monitoring framework keep monitoring it for a period of time. Since the collected data are the traces of the benign behaviors of the application, a statistic model could be created by computing the distances of the traces. For example, each of the system call trace could be considered as a string and for each different character found in two strings is a counted distance (*i.e.,* 10 character difference results a 10 distance). The overall variance V of all the traces is calculated, and the max distance M and min distance m of traces are calculated as well. For each subsequent system call trace, the distances to all the traces in the model would be calculated, the max distance M' and min distance m' would be also calculated. If $m' \leq m$, then this trace would be labeled as benign. If $m < m' < M - m - V$, then it would be labeled as suspect. Otherwise, this trace would be labeled as malicious.

After the first step labeling, all the traces associate with a label, and all the parameters would be transformed to unigram form for further processing. An example result of the current step is shown in Figure 3.7.

The SVM grading procedure is then performed on these traces. The underlying idea is simple in that we know during the classification of a sample, the decision function of the SVM classifier generates out distances (scores) to all the categories. The one with the least distance

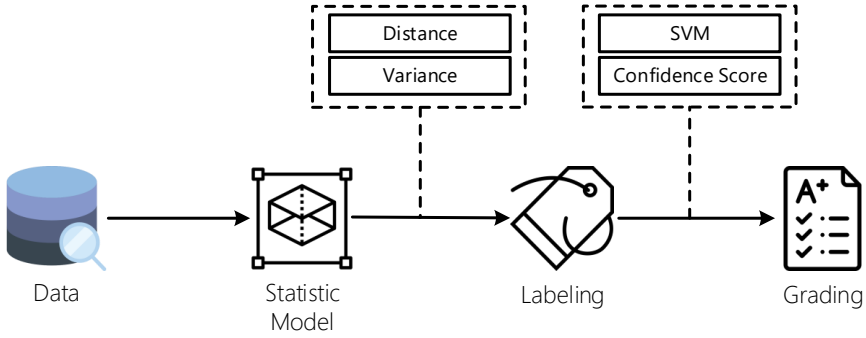


Figure 3.6: SVM Labeling Procedure

Figure 3.7: Labeled Data Samples

would be picked as the final predicted category. Instead of using the final prediction, we utilize the corresponding decision function output DF. The $\frac{1}{DF}$ would be applied as the score of the trace, since it also provides a baseline confidence. The larger the score the more SVM confidence on it belonging to current category. In addition, if the category is benign (good), the score is positive, otherwise it is negative. An additional weight could be used to bias the score system, such as multiplying, all the scores in the malicious category by 10. In this way, the difference of the suspect and malicious would be enlarged. An example scored data sample is shown in Figure 3.8.

Figure 3.8: Scored Data Samples

3.6 Evaluation

We ran our classification system in three phases. First, we ran it with only system call names as features resulting in 57.41% of the samples being correctly classified. The low accuracy was due to the inability to confidently link calls to some forms of malicious behavior. For example, downloading a file from Google is benign, however, downloading for 1000 times in a row is

not. We added a timestamp to the classifier to provide a temporal context. Doing so resulted in 70.43% accuracy. The disappointed accuracy showed still undetected malicious behaviours. Using system call name and timestamp ignores such events are attempting to access protected files. This indicates the need to include information that accompanies system calls.

We fed the classifier with all three features and combined it with labeling. The final result shows that 100% of the samples are classified correctly.

Training	**Result**
Only System Call Names	57.41%
With Timestamp	70.43%
With Feature and Labeling	100%

Table 3.2: Evaluation of the Prototype System

3.7 Summary

As shown in the Figure 3.9, this system consists of a sample program, a sysdig monitor, several preprocessing scripts, a SVM based labeling and grading module, and a LSTM based classifier. By using this framework, all the behaviours of the sample program running in the container could be correctly classified. The monitoring is performed in an out-of-box manner, and the knowledge of the program could be eliminated.

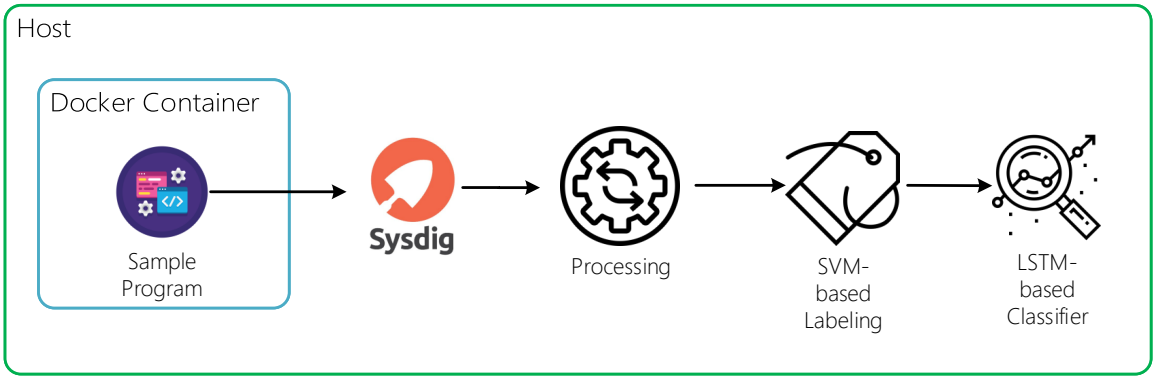


Figure 3.9: The Final Implemented Prototype

Chapter 4

Towards Unsupervised Introspection of Containerized Application

Based on the success of the proof of concept, we moved to unsupervised introspection of containerized application in a real production environment. In this chapter, we aim to provide secure monitoring of containerized application, which can help i) the cloud infrastructure owner to ensure the running application is harmless, ii) the application owner to detect the anomaly behaviours. We use unsupervised introspection tool to perform the non-intrusive monitoring, which leverages the system call traces to classify the anomalies. Since the traditional dataset used for anomaly detection either only focuses on network traces or limited to few attributes of system calls, we crafted and collected various normal and abnormal behaviours of containerized application, and an optimized and open source system call based dataset has been built. Unsupervised machine learning classifiers are trained over the proposed dataset, a comprehensive case study has been performed and analyzed. The results show the feasibility of unsupervised introspection of containerized applications.

The major content of this chapter are summarized as follows:

- We propose using non-intrusive introspection tools to provide system call level monitoring on containerized applications, as opposed to previous work focusing on using Linux security features to regulate the security boundary of containers [13, 16, 17, 118, 119]. While monitoring behavior by observing system calls has been widely investigated, the approach bears revisiting from the perspective of containers.

- We built a new open-source system call based dataset for system call based monitoring using state-of-art introspection tools with novel underlying dataset designs. The traditional and widely used datasets [20–26] have distinct pitfalls for the objective of system call based detection. The problems of the previous dataset have been comprehensively analyzed in [120]. Therefore, we aim to optimize the dataset with extended feature space and application behaviors. Note that, we provide a baseline dataset in this work, but the further optimization of the proposed dataset will be performed in the near future.

- We also created a public repository for sharing the container image and the code used in the data generation, one could use these materials to reproduce/scale/customize the data.

- We proposed, implemented, and evaluated unsupervised introspection approach over the aforementioned dataset. The proposed LSTM-based framework comprises a scaling factor normalization module and a tunable threshold module, which enables the framework to achieve accurate classification on unseen behaviors within the unsupervised manner. The evaluation result not only demonstrates the efficiency of the proposed methods, but also explains the necessity of an extension of features for detecting certain classes of anomalies.

4.1 Threat Model for Containerized Applications

A general containerization threat model is defined in [119], where the threats in containerization can be summarized and depicted as Figure 4.1. The containerized application can intentionally perform malicious behaviors such as remote code execution and privilege escalation, or the application/ image can be embedded with malware and rootkits. The application running in the container may lead to unauthorized access and intrusions. The adversary can also use one container to scan other containers in the system or even Denial of Service (DoS) and spoofing other containers. In addition, the container escape attack enables the adversary to access and tamper the data in the host environment. If the adversary has access to the upper level

(host/VM), he/she can profile the in-container application behaviors and manipulate the container life-cycle. In the past few years, several notable vulnerabilities have been reported, the corresponding CVE entries are also listed in Figure **??**.

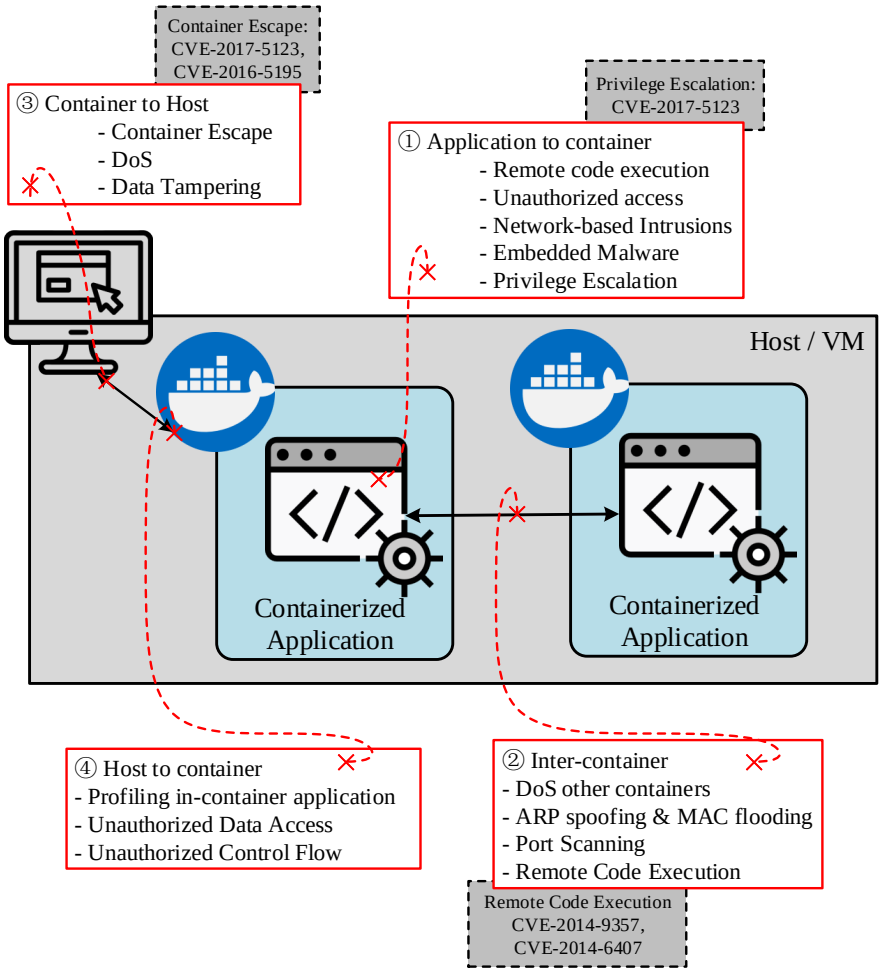


Figure 4.1: Threats in Containerization

4.2 System Call Based Anomaly Detection Dataset

Publically available datasets are available for computer system anomaly detection [20–26]. These datasets are limited in a number of ways:

- Some of the datasets are created with network traces. Although it may seem the majority of the attacks need the involvement of network communications, it still leads the dataset to be more intrusion-oriented. Internal abnormal behaviors in the monitored system related to the running process, memory access, and file system operations cannot be provided by the network traces.

- Several system call based datasets are introduced by other researchers, but the scale and coverage of the datasets are limited [120]. These proposed datasets consider only the system call itself, other information such as the caller and the arguments are missing. In addition, the attack types that the covered by the previous attacks are limited as well.

- A notable problem of the previous dataset is that benign behaviors are limited or unknown. The benign behaviors can be just simple login or system commands, such as, ps, and *scp*. Although the attacks used in these datasets are clearly demonstrated, benign behaviors are not always clearly described. In addition, it is hard to know whether the monitored application/machine is idle or not. The majority of the previous system call based dataset [22, 27] does not indicate any evidence of polling/listening, an approach commonly used in real-world applications

- None of the datasets is container-oriented. From the system call traces perspective, the major difference between a native host system and the containerized system is that not all the kernel modules are also directly running in the container, thus the traces generated from a container is significantly less noisy. This also provides a favorable foundation for further introspection and analysis.

To overcome these shortfalls, we generated a new dataset that is tailored for container-oriented system call based anomaly detection. We use Docker [121] as the containerization platform, the docker version is 18.03.1-ce, build 9ee9f40. The details of the dataset are demonstrated in the following subsections.

4.2.1 Introspection Tool

Traditional system call based solutions use system call capture tools such as *strace* or *BSM* [21, 86, 88]. These tools are limited in that they can only generate basic level system call traces in a summary style, where only the counts of different system calls are recorded, or they can only generate system call traces for the overall system or a single process, while the filtering of noises (system calls generated by other irrelevant process or kernel modules) and collection of all processes running in a container need to be further implemented. We employed the container

introspection tool, *Sysdig* [61]. *Sysdig* is an open-source container monitoring framework, which provides detailed and comprehensive real-time information of the monitored container, such as system calls, resource usages, and network traces. Instead of only providing a summary of all the calls, the system call name, caller process, timestamp, and passed arguments of each system call can be recorded independently in *Sysdig*. In addition, all the processes and system calls in the same container can be monitored together without including any other noises.

4.2.2 Customized Containerized Application

In order to simulate the real-world application behaviors and generate benign data for training, we first created a sample application. The sample application needed to cover reasonable operations as well as initiate network commutations and file system operations. We selected the containerized MySQL server as the sample application. Random user and password pairs were generated, inserted, and updated within random tables in this MySQL database server by a remote MySQL client. The creation of the tables/entries in the MySQL container would trigger the modifications to the file system. All the behaviors occurred at random intervals, with MySQL polling regularly for incoming SQL commands. Note that, this sample application may not be representative for all production environment applications, but it can provide a baseline data flow with needed operation traces. In addition, the backend database containers are widely deployed as the components in the production environment.

4.2.3 Selected Attacks

Aiming to provide comprehensive malicious behavior of the containerized MySQL server, we selected and performed eight attacks during data generation, the details are demonstrated in Table 4.1.

4.2.4 Collected Dataset in Open Source

The collected dataset, customized and containerized data generation container image, and sample data generation scripts are all included in the Github repository. The formats of the captured raw data are shown in Figure 4.3. A basic preprocessing procedure has been performed on the

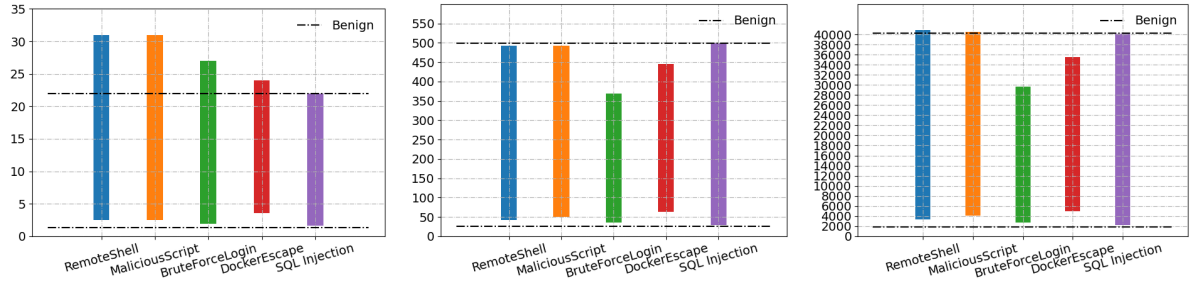
Attack	Description
Brute Force Login	Straightforward brute force login attack onto the MySQL server using Hydra. The attack comes with random attack duration but will continue the attack with previous progress in the last attack session. The attack progress will be reset for each data collection period.
Simple Remote Shell	A simple remote shell implementation using Netcat. Additional openssh and Netcat packages are installed in the MySQL container
Meterpreter	A malicious executable file is created with Metasploit and it is pre-loaded into the MySQL container. Upon the running of the file, the adversary can send a particular payload to activate the remote shell.
Malicious Python Script	A sample python script that downloads and runs an executable in an in-memory manner, and no artifacts leaves in the file system. Python related packages are installed in the MySQL container.
SQL Misbehavior	The normal behaviors only include INSERT and UPDATE, however, if the client tries to DELETE entries from the database then the operation would be considered as abnormal. Maintaining the database without using DELETE is more reasonable.
SQL Injection	A SQL injection attack is simulated by SELECT out all the tables and entries stored in the database.
Docker Escape	Implementation of CVE-2019-5736 [122, 123]. This attack requires the docker version older than 18.03.1-ce, build 9ee9f40. Note that, this attack modifies the underlying docker runtime (*runc*).
Selected Malwares	40 malware samples were selected from the 10K ELF malware collection (VirusShare_ELF_20190212 [124]). We validate all the malware samples using VirusTotal as well [125]. The malware collection including backdoor, DDoS agent, Trojan, Worm, Ransomware, and cryptocurrency miners. More details will be described in the Github repo. These malware samples can be used to generate real malicious behaviors in the container. All the malware samples are pre-loaded into the MySQL container or can be downloaded from the host via ftp.

Table 4.1: Selected Attacks

Metric \ Category	Benign	Remote Shell & Meterpreter	Malicious Script	Brute Force Login	Docker Escape	SQL Attacks
Different calls used	56	52	59	62	40	56
Calls not used in benign	0	20	22	29	25	0
Number of calls captured	7,144,780	595,630	819,692	172,156	546,191	123,565

Table 4.2: Summary of Collected System Calls

raw data, where all the data values are transformed into training-ready numerical values. The Table 4.2 describes the captured system calls of different behaviors (Note that, the malicious behaviors are performed along with the benign behaviors, but the Table 4.2 summarizes benign and malicious behaviors separately for better interpretation). The benign behaviors used for unsupervised training include 56 different types of system calls, whereas most of the malicious behaviors will use around 20 other system calls that are not used by the benign behaviors. The SQL-based attacks only generate different SQL workflows thus no additional system calls are involved. Some of the representative system calls used by the malicious behaviors are *bind, chdir, chroot, getpid, statfs, socketpair, readlink, recvmsg, seccomp*, and etc. The detailed system calls used by different behaviors are not fully listed here due to the limited space.



(a) (b) (c)

Figure 4.2: System Call Arguments Comparison of Different Behaviors. (a) The number of fields used in the arguments. (b) The length of arguments. (c) The value of arguments. Since all the system calls may have 0 field and 0 length arguments, the range of each bar is from "average to max".

The system call arguments used by different behaviors are compared in Figure 4.2. Since all the behaviors (benign and all the attacks) may use system call with 0 fields and 0 length arguments, the range of each bar is from "average to max" instead of "min to max". As shown in Figure 4.2, benign behaviors have an average of 1.36 fields in the arguments with 25.4 characters average length. Most of the attacks use more fields in the system call arguments and also result in a larger argument length. For example, the Docker Escape attack has an average of 3.58 fields in each system call arguments with an average of 62.85 characters. In addition, the values of arguments (accumulated ASCII value of all the characters) in different behaviors have similar trends, which are depicted in Figure 4.2-c as well.

4.3 Unsupervised Introspection of Containerized Application

Crafting a new machine learning technique for unsupervised introspection was not the main objective of this paper, instead, we leveraged the state-of-art techniques to investigate the feasibility of reliable introspection. The goal of the introspection was to accurately classify the benign and malicious (or suspect) behaviors using discrete system call traces. We believe that after a certain time of monitoring on the application (ensured to be secure during this period of time), an unsupervised classifier can be generated to automatically infer the boundary between benign and malicious. System call based detection is subject to be a collective anomaly detection problem. It is inappropriate to directly mark all the system calls with a fixed score or label. Therefore, the anomaly detection should not be performed in a single system call level

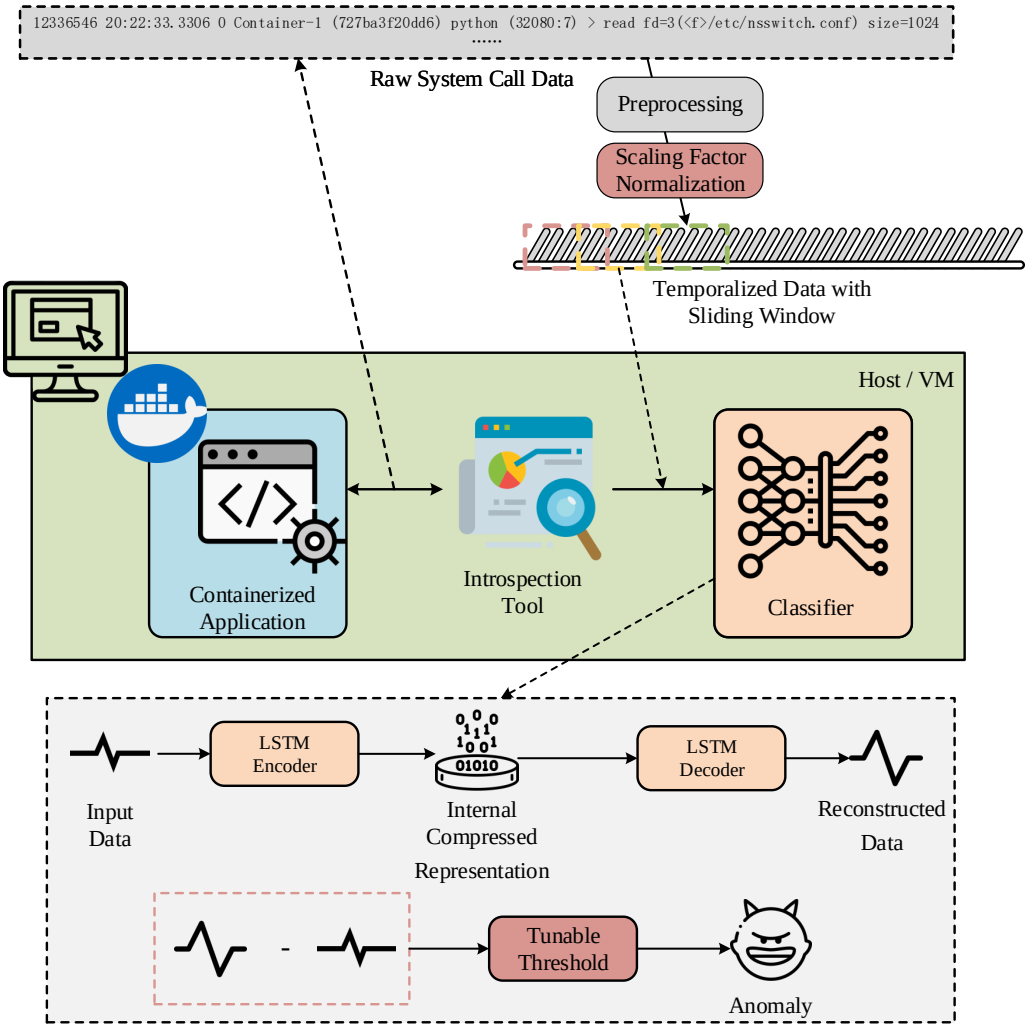


Figure 4.3: Proposed Unsupervised Introspection Framework

but achieved by classifying a batch of system calls. The overall design of the proposed framework is described in Figure 4.3, and the framework consists of a sliding window mechanism, a scaling normalization algorithm, an autoencoder-based classifier, and tunable thresholds.

4.3.1 Sliding Window

Normally, a sliding window is equivalent to a time-slot, with all the data captured in the time-slot aggregated as one data instance for further training or classification. The random behavior of applications generates an inconsistent number of system calls in a given time-slot. Although data padding may alleviate data streams of different lengths, it cannot solve the problem when the malicious data has an unknown number of system calls. This requires the original training data to be padded with a fixed larger size (to ensure all the incoming malicious data can be padded to the same size) or the incoming validation data to be compressed to the same size

of training data. This increases the overall size of the data and also violates the unsupervised motivation. We used the sliding window on a system call series, where a fixed number system calls are aggregated together as one window of data. The entire training data is temporalized with this fixed window size to achieve the sliding of the window.

4.3.2 Scaling Factor Normalization

The data distribution and extreme values of malicious data are unknown. In this unsupervised introspection procedure, if historical trained extreme values are used for the Min-Max normalization of new data, the normalized data may be negative or greater than one; if the Min-Max Normalization is carried out on new data itself, though the normalization returns the data with the range zero to one, the normalized data represents different actual values. Therefore, we adopt and modify the Scaling Factor Normalization (SFN) method [126] to enable the data normalization scaling on the new unknown data.

The traditional min-max normalization can be described as equation 1, and the SFN introduced in [126] can be described as equation 2.

$$x'_{ij} = \frac{x_{ij} - min(x_j)}{max(x_j) - min(x_j)} \tag{1}$$

$$x'_{ij} = \frac{x_{ij} - min(x_{ij} * (1 - a_j), x_j)}{max(x_{ij} * (1 + a_j), x_j) - min(x_{ij} * (1 - a_j), x_j)} \tag{2}$$

The value a_j is the scaling factor that influences the effectiveness of normalization with the range:

$$0 < a_j < \frac{max(x_j) - min(x_j)}{max(x_j) + min(x_j)} \tag{3}$$

This method works with the assumption that all the data values are positive [126]. If the minimum value of any feature is zero, this method cannot scale the normalization anymore. Unfortunately, the system call traces data may generate zero values as the arguments of the system call can be empty. Therefore, we modify the original SFN to the following form:

$$x'_{ij} = \frac{x_{ij} - min(x_{ij} * (1 - b_j), x_j^+)}{max(x_{ij} * (1 + b_j), x_j^+) - min(x_{ij} * (1 - b_j), x_j^+)} \tag{4}$$

The x_j^+ is all the positive values of x_j and b_j is $\frac{max(x_j)-min(x_j)}{2*(max(x_j)+min(x_j))}$. The value of b_j is equal to the half of a_j maximum value, this ensures the validity and certain effectiveness of the normalization scaling.

4.3.3 LSTM-based Classifiers

A variety of options can be used to accomplish the unsupervised classification task. Traditional clustering algorithms, self-organizing maps, one-class SVM, and autoencoders are all tested and evaluated by the other researchers. Since the data in our model is in the form of system call batches, we wanted the classifier to have the capability of memorizing and utilizing the knowledge of previous batches. We selected the classic LSTM autoencoder [127,128] as the baseline classifier. As illustrated in Figure 4.3, the LSTM autoencoder is formed by an encoder and a decoder. The encoder transforms the input data into the internal compressed representations, then the decoder reconstructs the data from the internal state. We simply adopt the classic LSTM architecture, more details can be found at [127,128]. We noticed that the classifier can be further optimized and the competitiveness of other classifiers is also well-known. However, this unsupervised prototype mainly aims to prove the feasibility of the proposed system call based introspection. A detailed and comprehensive comparison of classifiers will be performed as the future work, more discussions are elaborated in Section 4.5.

4.3.4 Tunable Threshold

The LSTM autoencoder is trained with only the benign data, thus the reconstructed data would be significantly heterogeneous whenever the malicious data is passed. The loss value between the real data and the reconstructed data can indicate the anomalies. The problem then becomes one of determining the loss threshold. Some of the solutions adopt user-selected values as thresholds [82–84], or add a semi-unsupervised or supervised classifier/algorithm for final inferences [78, 129]. However, these methods violate the unsupervised assumption. We employ a tunable threshold mechanism to provide unsupervised and tunable anomaly detection and it is described in Algorithm 1.

The underlying concept of the proposed tunable threshold is simple and straightforward. As it is not possible and appropriate to ask the users to decide the threshold value, then we make the classifier itself to be embedded with a tunable thresholds pool. So the user can adjust the sensitivity of the classifier in the future. The threshold pool is created after the training procedure. During the validation of the training, a set of confidence levels: [100%, 99.5%, 99%, 95%, 90%, 85%, etc.] is used to

Algorithm 1: Tunable Thresholds

Input : Classifier (*C*), Validation Dataset (*VD*), Confidence Levels (*L*)
Output: Tunable Thresholds (*T*)

Tunable Thresholds $T \leftarrow []$;
Reconstructed Data $(RD) \leftarrow C.predict(VD)$;
Loss Matrix $(M) \leftarrow |VD - RD|$;
Max_threshold $(t_m) \leftarrow M.max()$;
for *Confidence level (l) in L* **do**
    current_threshold $t_c \leftarrow t_m$;
    while $(M < t_c).count \,/\, M.count > l$ **do**
        $t_c \leftarrow t_c * 0.99$
    end
    $T.append(t_c)$
end
Output T;

generate the thresholds. The algorithm will automatically find the appropriate thresholds which make the classifier generate validation accuracy that equal to corresponding confidence levels. For example, 100% confidence makes the classifier has a 100% classification accuracy in validation by using the maximum loss value in the reconstructed data as the threshold. Note that, the higher the confidence level the classifier has, the less sensitive the classifier would be, because the high confidence level also leads to a high threshold value.

4.4 Evaluation

In this section, we perform the evaluation of the framework on the introduced dataset. We mainly aim to investigate the feasibility and performance with different configurations. The LSTM classifier is implemented in Keras and trained with an Nvidia Telsa K40c graphic card. Adam optimizer is adopted. Dropout and early stopping are enabled to avoid overfitting. We trained and evaluated the classifier with different features, sliding window sizes, and confidence levels. The detailed hyperparameters are listed in Table 4.3.

4.4.1 Evaluation Metric

We adopt three metrics in the evaluation procedure: negative predictive value (NPV), accuracy, and coverage (specificity). These three metrics are described in Equation 5, 6, and 7, respectively. NPV and Accuracy indicate the percentage of the windows are classified correctly, where the former provides the accuracy with only negative part and the later provides the overall accuracy. The Coverage metric can provide an overview of the amount of the negative values that are correctly classified. In order to have an overall measurement of the performance, a score metric is adopted as well, which is simply derived by averaging the three metrics.

	Hyperparameter	Value
Adam	γ	0.001
	β_1	0.9
	β_2	0.999
	ϵ	10^{-8}
Features	F	$[call, time]$, $[call, time, process, value]$, $[call, time, process, length, fields, value]$
Sliding Window Size	S	$[10, 20, 30, 40, 50, 60, 70]$
Confidence Level	C	$[1.0, 0.995, 0.99, 0.95, 0.90, 0.85]$
LSTM	units	$[128, 16, 16, 128]$
	activation function	relu
	dropout	0.2
	batch size	$[30, 300, 3000]$
	loss function	MAE

Table 4.3: Training Hyperparameters

$$NPV = \frac{TN}{TN + FN} \tag{5}$$

$$Accuracy(ACC) = \frac{TN + TP}{TN + FN + TP + FN} \tag{6}$$

$$Coverage(COV) = \frac{TN}{TN + FP} \tag{7}$$

$$Score = \frac{NPV + ACC + COV}{3} \tag{8}$$

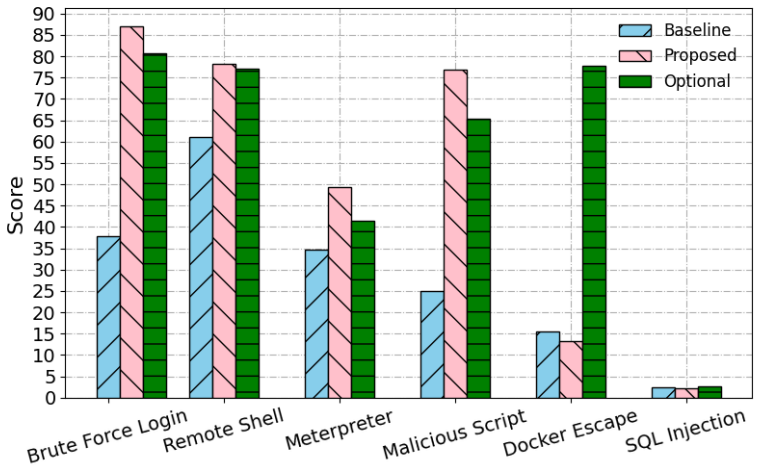


Figure 4.4: Evaluation Results with Different Features

4.4.2 Experiment Results

With using different configuration combinations, a total of 42 models are trained in this experiment (7 window sizes, 3 feature sets, and 2 normalization methods). In addition, 6 confidence levels are applied, and 7 attacks are evaluated (malware are not evaluated in this paper, more details can be found in Future Work section), which results in a total of 1764 entries. Here, for demonstration, we performed the discussion using confidence level 99% with window size 50. A detailed result of all the attacks with confidence level 99% and window size 50 is presented in Table 4.5. In addition, the best results of all the attacks are also summarized in Table 4.4, which proves the prototype system can provide an acceptable accurate detection for most of the attacks.

Attack	**Results**				
	Model (norm-feature-window size-confidence)	*NPV.*	*ACC.*	*COV.*	*Max Model Loss*
Brute Force Login	SFN-prop.-70-99%	74.51%	95.82%	99.77%	0.096
Remote Shell	SFN-opt.-50-100%	84.71%	93.81%	92.01%	0.089
Meterpreter	SFN-prop.-70-99%	24.39%	95.24%	54.85%	0.096
Malicious Script	SFN-opt.-70-99%	84.88%	90.64%	83.68%	0.084
Docker Escape	SFN-prop.-60-99%	89.35%	76.27%	84.75%	0.101

Table 4.4: Sample Best Result on Selected Attacks

By leveraging the advantage of the additional features such as caller process name and system call parameter values, the proposed framework (*proposed version*, with SFN enabled) can significantly increase the detection accuracy compares to traditional methods that only use system call name and timestamps (*baseline version*, with no SFN). This is depicted in Figure 4.4. However, more is not always better. With including length and fields of system call parameters into the training (as (*optional version*), the results do not always outperform the *proposed version*. The Figure 4.4 also describes this situation.

The SFN always increases the accuracy of the classification regardless of what other configurations are set except for the Docker Escape attack, the detailed results can be found in Table 4.5. The window size defines the amount of the system calls the classier can memorize and process, which significantly determines the accuracy and performance. However, the relationship between the window size and classification accuracy is still attack related, but a fixed window size can also provide an acceptable detection accuracy. Note that, we only evaluated the window size with maximum size 70, as the training cost of higher window size is not trivial. Meanwhile, the tunable threshold can also determine the classification sensitivity and accuracy. When the lower thresholds are applied, the higher coverage

Model	Brute Force Login			Remote Shell		
	NPV.	*ACC.*	*COV.*	*NPV.*	*ACC.*	*COV.*
MinMax-base	16.77%	86.55%	10.54%	**64.88**%	83.95%	34.33%
SFN-base	14.52%	85.42%	13.85%	63.86%	**84.54**%	37.93%
MinMax-prop	0.05%	84.47%	0.09%	62.5%	84.27%	31.34%
SFN-prop	67.19%	**94.12**%	**99.57**%	49.97%	84.35%	**99.98**%
MinMax-opt	1.95%	80.96%	2.08%	57.30%	84.39%	46.47%
SFN-opt	**67.85**%	92.09%	81.95%	49.37%	82.49%	99.36%

Model	Meterpreter			Malicious Script		
	NPV.	*ACC.*	*COV.*	*NPV.*	*ACC.*	*COV.*
MinMax-base	1.19%	**95.88**%	7.14%	0%	75.19%	0%
SFN-base	3.21%	91.27%	31.83%	0.11%	74.57%	0.12%
MinMax-prop	**14.35**%	96.2%	16.81%	36.73%	75.45%	4.25%
SFN-prop	13.73%	91.95%	**42.04**%	**77.22**%	**84.73**%	**68.36**%
MinMax-opt	12.91%	94.57%	15.81%	56.72%	77.14%	25.44%
SFN-opt	10.01%	95.39%	18.62%	67.62%	79.95%	48.41%

Model	Docker Escape			SQL Injection		
	NPV.	*ACC.*	*COV.*	*NPV.*	*ACC.*	*COV.*
MinMax-base	5.18%	41.28%	0.39%	6.66%	0.44%	0%
SFN-base	7.48%	42.02%	0.5%	5.55%	0.34%	0%
MinMax-prop	**86.13**%	71.45%	75.32%	8.14%	0.87%	0%
SFN-prop	6.06%	33.39%	0.6%	5.24%	1.16%	0%
MinMax-opt	77.6%	52.58%	24.31%	7.7%	1.01%	0%
SFN-opt	81.87%	**75.91**%	**75.67**%	7.00%	0.93%	0%

Table 4.5: Evaluation Results with 99% Confidence and Window Size 50

would be achieved but the price is the accuracy. We observed this situation in the evaluation results as we expected.

We notice that the classifier fails to detect SQL injection and SQL misbehavior attacks. These two attacks generate almost the same operation flows in the system call traces except the keywords passed in the parameters are different. Since all the benign behavioral data of the containerized application used for training are randomly generated, a simple change on a keyword cannot lead to a significant variation or make this operation to be an outlier. It may be possible to detect such variation if all the other parameters of all the SQL behaviors are fixed while these attacks use different keywords. However, these violate the motivation of real-world application simulation. This also indicates the vulnerabilities in the functionality level should be better regulated and concerned at the application level.

4.5 Discussion and Future Work

The malware embedded container and data are available in the dataset and repository, however, it is non-trivial to present a detailed analysis on a large number of malware samples with this limited space. It would be meaningful to extend the investigation on the detection of different malware behaviors in the future. This work provides a proof-of-concept on the unsupervised introspection of the containerized application, and we noticed that the proposed framework is only evaluated with an offline dataset with a

single application. For the evaluation perspective, we aim to provide a comprehensive online evaluation as our future work. On the other hand, we plan to scale the proposed dataset with different applications to provide more generalized coverage of application behaviors.

In addition, data of other application architectures will be collected and included as well, such as multi-container applications where a service is formed by a collection of containers. An introspection on this type of service requires the monitoring system to have an overview of all the containers and meanwhile having the capability to understand and interpret the relationships between the containers. We can investigate the feasibility and efficiency of introspection in such a clustering environment with our updated dataset in the future.

Chapter 5

Perturbing Smart Contract Execution through the Underlying Runtime

The virtualization and containerization have been adopted in many other applications besides cloud computing. Smart contract is one of the most promising techniques that also relies on the virtualization and containerization. Smart contract is a piece of code publicly stored in the blockchain and can be triggered by different nodes in blockchain network, these nodes can be different architecture-based and operating system-based. Ethereum and Hyperledger use different methods to ensure the smart contract can be run on all nodes and generate the same result. However, the core concept is the same: via virtualization. Ethereum adopts a virtual machine mechanism similar to the Java Virtual Machine, named Ethereum Virtual Machine (EVM). EVM is a stack machine that executes bytecode [130] transformed from a high-level smart contract programming language (Solidity and Vyper [131]). EVM is an embedded component of an Ethereum node client, which automatically runs in memory. In contrast, Hyperledger uses a Docker container to execute the smart contract. The smart contract in Hyperledger can be written in Go, Java, or NodeJS. The code is packaged and instantiated as a Docker container in the Hyperledger node's system. Each smart contract runs as a container, more details are described in Section 5.1.

In this chapter, we aim to attack the smart contract system beyond any one smart contract programming platform. Although the mainstream handles the smart contract security at programming level, we propose to review the smart contract security in a different view. We take the perspective of an adversary that does not care how the smart contract is programmed, as long as we can interfere or manipulate the output of smart contract execution and thus bypass any verification and protection frameworks that might be in place. We attempt to illustrate the flaws of the underlying smart contract life-cycle or virtualization mechanism (runtime) namely, the Ethereum Virtual Machine in Ethereum and the Docker container environment in Hyperledger Fabric. Since the smart contract installation and execution in Ethereum and

Fabric are different, we mainly focus to investigate the potential security risks in the Hyperledger Fabric system.

The remainder of this chapter is organized as follows. We perform a case study on Hyperledger Fabric in Section 5.1. The discussion of migrating this attack to Ethereum has been performed in Section 5.2. The findings, limitations, and countermeasures are demonstrated in Section 5.3.

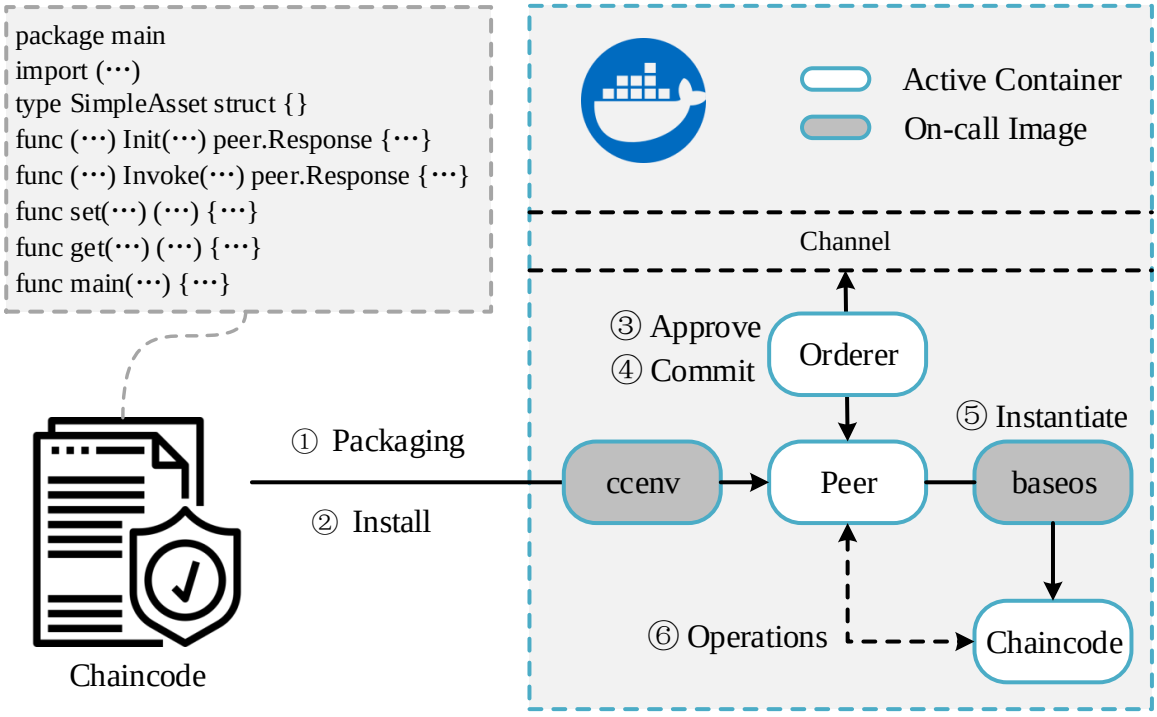


Figure 5.1: Chaincode Life Cycle in Hyperledger

5.1 Case Study: Attack on Hyperledger Fabric

In this section, we perform a case study on the Hyperledger Fabric platform to investigate the feasibility of perturbing the smart contract execution via runtime vulnerabilities. The entire case study is based on the official Hyperledger network test example, *Byfn* network. We demonstrate the chaincode (the alias for the smart contract in Fabric) life-cycle and Docker environment of Hyperledger Fabric system before introducing the vulnerability itself.

5.1.1 Chaincode Life Cycle

Figure 5.1 illustrates the life-cycle of chaincode in the Docker containers that Hyperledger uses as its runtime environment. The following steps transpire when a Hyperledger blockchain peer tries to launch and test a piece of chaincode (starting in the top left corner of Figure 5.1):

1. Packaging: The *Peer* first packages the chaincode into a tar format file.
2. Install: The compressed package is delivered to all the peers that need to run/endorse the chaincode. These peers build, compile, and install the chaincode locally.
3. Definition Approve: Corresponding channel members vote on and approve the definition of the chaincode, which includes such information as name, version, and endorsement policy (i.e., who can execute and validate).
4. Commit: Upon a success approval, a commit transaction proposal is submitted to the *Orderer*, which then commits the chaincode definition to the channel.
5. Instantiate: The chaincode is initialized as active containers for all the peers.
6. Operations: Chaincode operations are carried out by the communication between the peer container and chaincode container.

Throughout this procedure, the *Orderer* and *Peer* container are active all the time. The *ccenv* container, which is the chaincode environment container provides the functionalities of installation and instantiation. The *ccenv* is an offline docker image that only becomes an active container when there is a chaincode that needs to be processed. The complied chaincode is added into a base image to create the real instance of chaincode container. The *baseos* image is always offline; the chaincode image becomes active after the compiled binary is loaded into *baseos* image.

5.1.2 Threat Model

In this case study, we assume the potential attacks originate from the Docker perspective. We therefore assume the adversary has access to the Docker network, images, and containers. The adversary may not have root privilege on the host machine, but he/she can access the host file system and alter the user space data via a pre-planted backdoor and/or remote control. The communication can be eavesdropped,

intercepted, and modified by the adversary. The adversary can be an unrelated third party or an insider. The prerequisites of the these attacks may enable the adversary to damage the system in a more severe and obvious way, but the major motivation of the adversary is to bias, perturb, and stop the service of the chaincode.

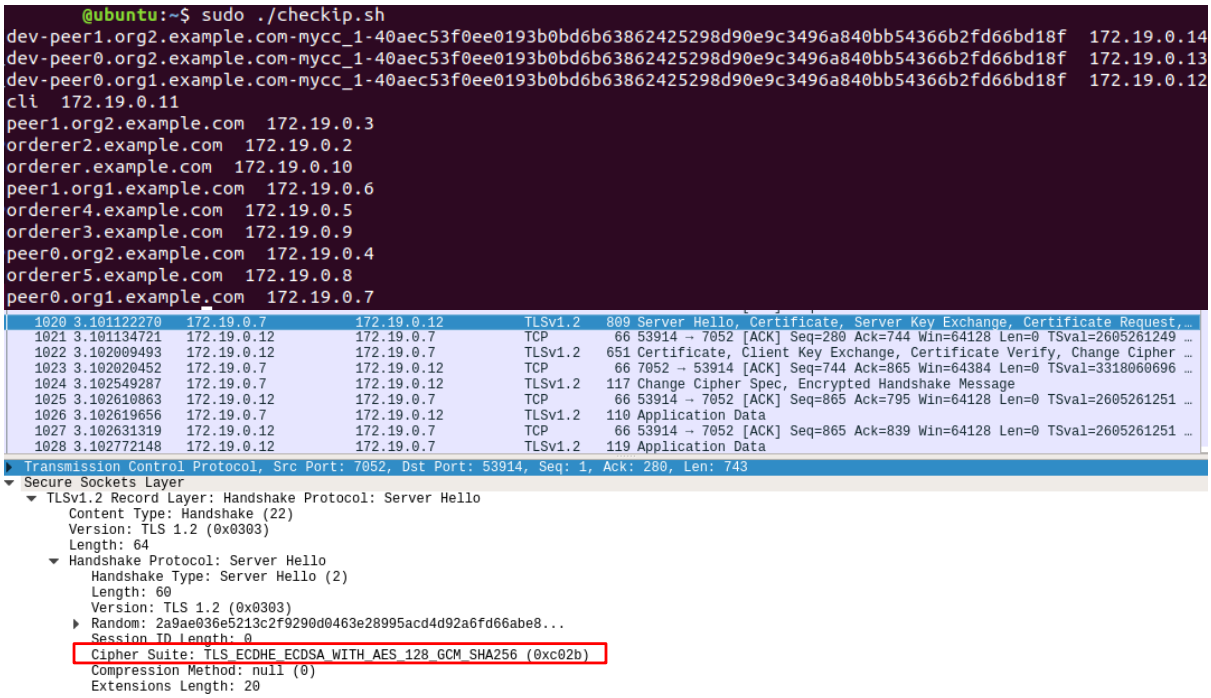


Figure 5.2: Communication and Handshake between the Peer and Chaincode Containers

5.1.3 Insecure Communication

The operations to an instantiated chaincode in Hyperledger are based on the communication between the *Peer* container and *Chaincode* container. However, this additional communication was not as secure as we expected. We sniffed the communication traffic between these two containers while a query operation was taking place. As shown in Figure 5.2, the communication was TLS v1.2 enabled, the encryption of the communication was based on ECDH Key Exchange, and the authentication was provided by mutual certificate verification. Normally, the story would just end here and presume the communication is secure and reliable. However, the vulnerability came from a permissioned blockchain and Docker container.

The permissioned blockchain requires the network to be a designated group of organizations and entities, and the enforcement of the network regulation relies on authentication. In other words, all the entities need to have corresponding certificates and keys for further verification. All these cryptographic

materials are pre-generated and shared in the entire network *by loading them into the container at the point the container is created*. This creates a problem: all the keys and certificates are stored in the user space of both host and container. Figure 5.3 shows the accessible keys and certificates stored in host and container, which are also stored in an unencrypted manner.

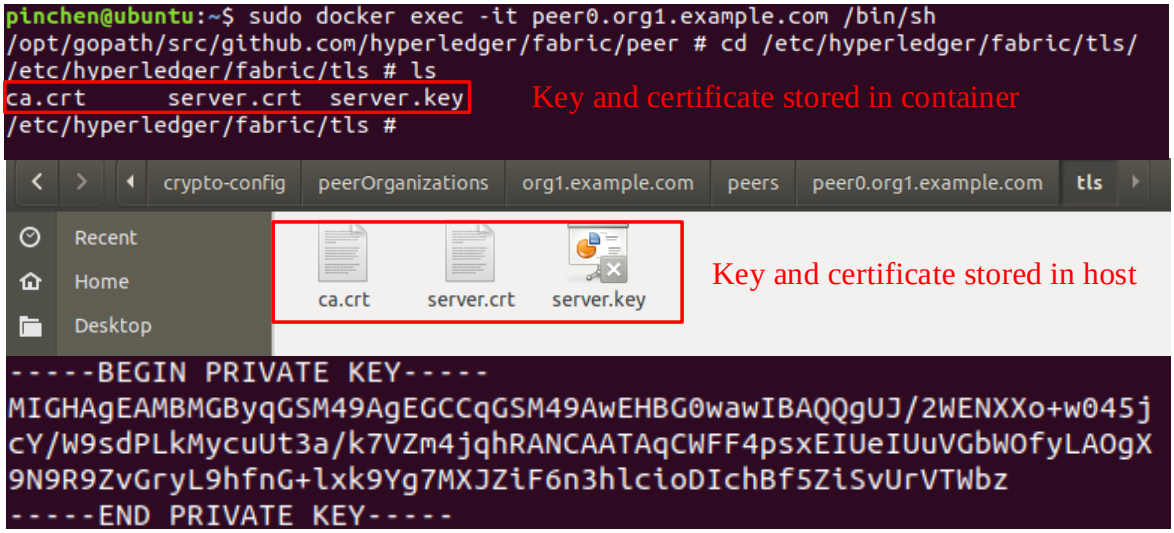


Figure 5.3: Unencrypted Key and Certificates stored in Host and Containers

The Hyperledger project provides a *Cryptogen* binary to help users tailor their cryptographic materials. Users can change the location of the keystore, change the key format and length, and create encrypted keys and certificates, but such information has to be loaded and stored into containers so that the containers can communicate with the blockchain network. Hyperledger containers come with root privileges, which provides access to keys. **As long as an adversary has access to any container, he/she has access to this material.**

Because ECDH Key Exchange is known to be vulnerable to Man-in-the-Middle (MITM) attack [132–134], thus mutual verification is needed. Traffic redirection tools (e.g. iptables and ARP spoofing) and TLS interception tools (e.g., SSLProxy *) make it relatively easy for an adversary to redirect the duo-direction communication between the *Peer* and *Chaincode* containers to a MITM agent. **All the operations to the chaincode can be then manipulated by the adversary**. An example attack scheme is described in Figure 5.4. Note that, besides the *Peer-Chaincode* communication, there is another vulnerable point. **The communication between the *Client* (user command line tool/ container that used to send the requests to Peer) and *Peer* can be the target of MITM attack as well.**

*https://github.com/sonertari/SSLproxy

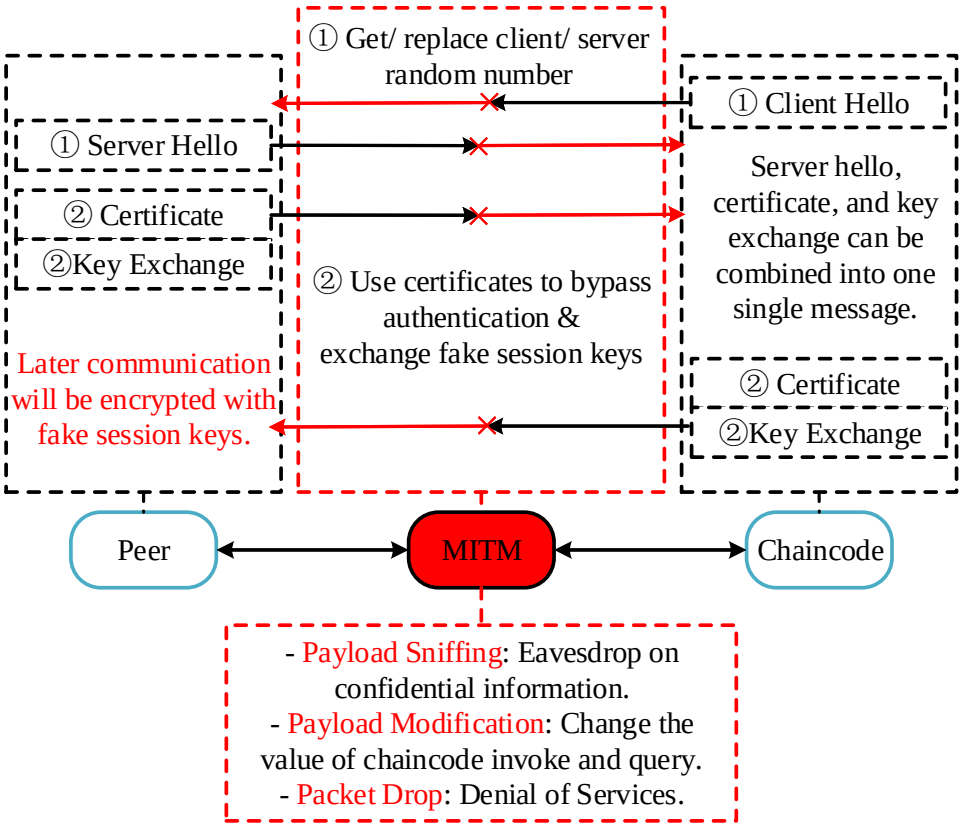


Figure 5.4: Demonstration of MITM attack on Chaincode Communication

After this stage, the attack becomes an engineering task of forging all the malicious packets. This is obviously nontrivial, but feasible.

5.1.4 Loose Image Management

As shown in Figure 5.1, the chaincode container is created by loading chaincode binary file into the *baseos* image. One potential strategy to circumventing the integrity of the chaincode would be to poison the base image, thus ensuring subsequent chaincode containers would be vulnerable if the *baseos* image were modified or replaced. The *baseos* image, along with other Hyperledger container images, are ill protected. Containers are called via tags instead of hashes. This means that a benign image can be modified or replaced and still appear to be valid if it has a tag that corresponds with the original.

Figure 5.5 illustrates a simple example of such an image alteration. The original on Hyperledger v2.1 *baseos* image was Linux Alpine based, with an image size of 6.94 MB. We replaced all the versions of *baseos* image to a Ubuntu-based image of 73.9 MB. In addition, we modified the clean Ubuntu image with Python and some other libraries installed, which also has been committed to *baseos* image

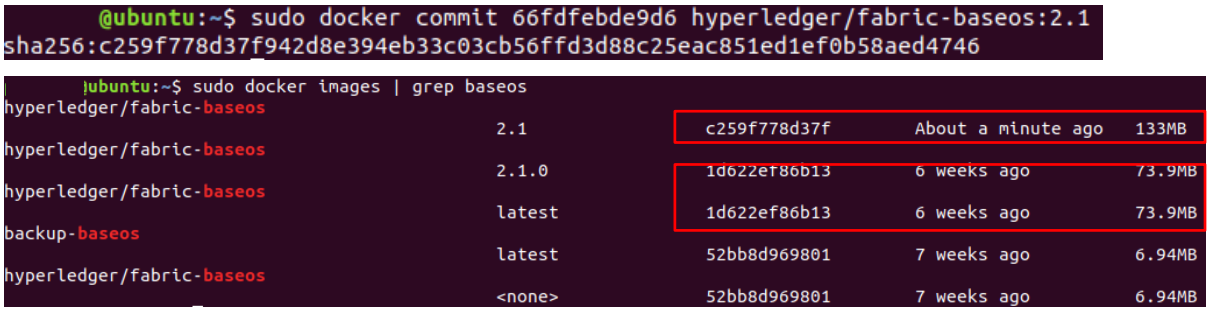


Figure 5.5: Replace the Hyperledger Baseos Image

version 2.1. **We observed that all the chaincode containers created after this image alteration were installed with Python and additional libraries**.

In this case, if the adversary can either redirect the user to download a malicious image or somehow modify/ replace the image, the entire system can be corrupted. The adversary can load customized code, autorun rootkit, revert channel backdoor, and cryptocurrency miner program into the malicious image. These malicious images can lead to denial of services, abuse of resources, unauthorized access, and information leaking. One can also use the malicious image to launch the aforementioned MITM attacks.

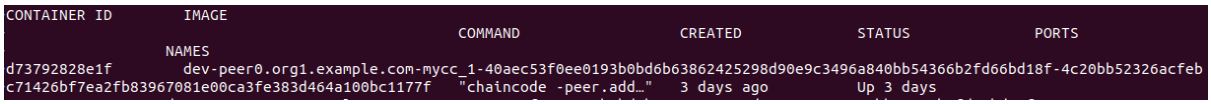


Figure 5.6: Dev-Chaincode Containers On-the-Fly

This naive attack can work because the Hyperledger container life-cycle environment calls all the images with tags. Although the hashes are inherently provided by Docker engine, it is apparently more convenient, albeit incautious, to use the human-readable tags. Moreover, besides the *baseos* image, all the other Hyperledger images are vulnerable as well. The orderer, CA, and peer images are all free for modification and replacement. Each installed and instantiated chaincode will be mapped with a container image (shown in Figure 5.6). These images that generated on-the-fly can be also modified and replaced.

5.1.5 Risks Behind Docker

Hyperledger outsources Docker as the chaincode runtime, and its security is then bounded with Docker container security, which basically doubles the attack surface. The adversary can utilize the vulnerability of Docker to perform much more severe attacks. For example, one can combine a malicious image with reverse shell and Docker escape attack to gain the root access of the host system [122, 135]. This can

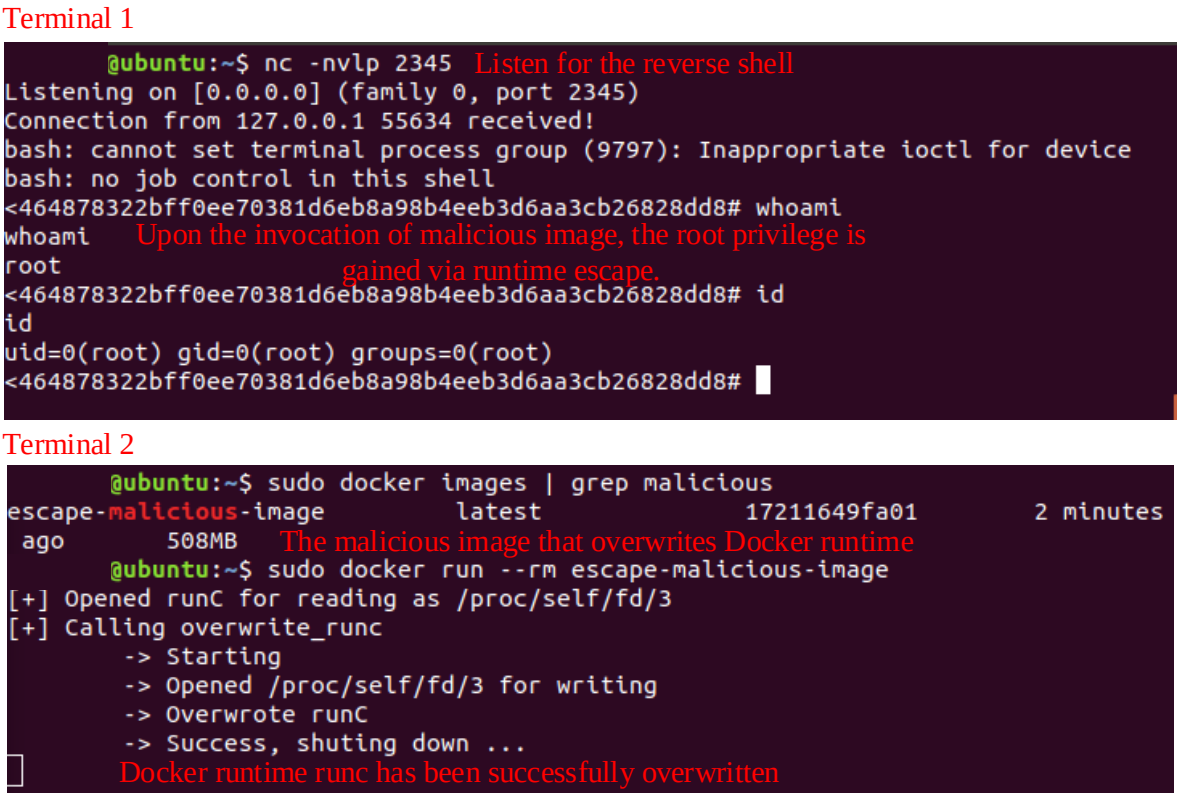


Figure 5.7: Gain Root Access on Host via Malicious Image: This attack allows the adversary to inject any code in Docker runtime (runc), which will be executed on host with root privilege. This example simply injects "$bash - i > \&/dev/tcp/0.0.0.0/23450 > \&1\&$" into runc, and a reverse shell with root access on victim's host will be created.

lead the attacker to have the full control of the entire system. An example set up is shown and described in Figure 5.7, where upon the invocation of the malicious image, a reverse shell establishes and returns with the root access of the victim's host. If the adversary replaces the *baseos* image with this malicious one, the entire chaincode system is disrupted. Note that the malicious docker images problem has been of concern for some time. By uploading malicious images into Docker Hub, an unscrupulous actor can generate $90,000 dollars from the million downloads and deployments of these malicious images in 10 months [136].

Complicating things further, Docker and Docker Compose are normally configured to be user-space applications. Since it is not practical, efficient, and secure in a production environment to ask all the persistent container operations for root privileges, the Docker environment can be significantly manipulated even without root access on the host system. Issuing Docker commands on the host can be another threat to the chaincode system. One can stop and re-run a chaincode container to break the established TLS connection with *Peer* container, thus, the sniffing and MITM can be launched at any

time. One can run the containers with privileged mode, so the *iptables* and all the other kernel-related system calls are enabled in the container for further malicious objectives (e.g., setup network forwarding rules and divert channels, and load malicious kernel modules).

5.2 What About Ethereum?

Ethereum achieves the cross-platform smart contract execution via EVM but several problems have been found in the underlying EVM mechanism, one notable vulnerability being the stack size limit [105,106]. Ethereum's smart contract can invoke another contract (including itself), and each invocation increases the size of the stack in EVM by one frame. It was possible to exceed the limit and cause an internal exception. [105, 106] demonstrated that an attacker could not have to pay coins by intentionally failing the payment fallback function by overflowing the EVM stack. This vulnerability was been fixed by a hard fork of Ethereum in 2016.

[137] observed that running particular smart contracts on different implementations of EVMs resulted in inconsistent gas (i.e., fees) and opcode sequences. This led to an EVM testing tool, EVM-Fuzzer, which can be used to detect potential vulnerabilities of EVM [138]. The tool revealed that Py-EVM version v0.2.0-alpha.33 allows attackers to make a malicious call to inject illegal values in stack (CVE-2018-18920 [139]). CVE-2018-19183 [140] and CVE-2018-19184 [141] documented two EVM vulnerabilities that enable the adversary to cause a denial of services. Two additional vulnerabilities CVE-2018-19330 [142] and CVE-2019-7710 [143] remain confidential to the public until they are repaired.

Generally, it may seem that Ethereum is more secure than Hyperledger since its runtime, EVM, is a customized in-memory stack machine. We focus on Hyperledger is not only because it is less discussed or arguably less secure, but also due to fact that EVM has already been attacked in a similar form. Both EVM stack overflow attacks [105,106] and CVE-2018-18920 [139] attacks perturb the normal execution of smart contract based on the vulnerabilities in EVM implementation. These two attacks can infinitely trigger the smart contract functions without corresponding gas and payments, which also do not rely on any programming faults in the smart contract. Although these two vulnerabilities have been already fixed, the concept of our proposed attack is verified.

A research question is whether it is possible to attack the EVM without a zero-day vulnerability. Because EVM is a program running in the local system, it can be perturbed if the system owner (or

the adversary with the same privilege) decides to do so. Indeed, any program and applications can be attacked in this manner as well, this type of attack is beyond the scope of "attack on the runtime". A smart contract is different from the other application scenarios. Since the contracts are immutable and reliable hardcoded programs in the blockchain, altering the execution or the result of smart contracts even in a more general manner would still be interesting. One potential direction is to locate the EVM stack and memory locations in physical memory space and use the *process_vm_writev()* system call to transfer and inject data into that memory location, thus altering execution.

5.3 Lessons Learned

5.3.1 Limitation and Impact of the Proposed Attack

Altering smart contract execution requires access to the runtime environment and/or certain vulnerabilities present at runtime. Access to the runtime environment may lead to other security concerns and make the perturbation unnecessary. For example, the adversary can simply remove the Docker engine from the victim's system, thus achieving DoS. Similarly, one can block the EVM implementation and Ethereum client from normal functioning by setting up certain network rules or checkpoints (via debugger). We notice that the assumption of having partial access to the system is a strong assumption, but note that all the aforementioned attacks in Hyperledger can still be performed without access to the system, as long as a malicious image is delivered.

Besides the prerequisites of the attacks, the impact of the attack is another question. The attack impact can be categorized into two types based on the smart contract function types: either read-only (*R*) or read and write (*R/W*). The *R* type functions will not assign miners for further smart contract execution, and the results will be locally read (from the current state database of blockchain), generated, and returned to the user. If this part has been perturbed, the query to the blockchain is altered or stopped. From the user's point of view, it is hard to know the real value stored in blockchain unless the transaction histories have been examined. On the other hand, *R/W* type functions create additional data in the blockchain, then miners (or endorsers in Hyperledger) have to be involved. From the Peer-Chaincode communication side, it may be hard for the adversary to manipulate the data stored in the blockchain, since it requires to break the consensus on chaincode execution result. However, if the adversary MITM the Client-Peer communication, then he/she can change the original parameters sent to the chaincode

and blockchain. Therefore, the final data upload into blockchain can be manipulated, though it will be in an obvious way – the user can easily find out that the client sent a different value.

5.3.2 Countermeasure to the Proposed Attack

If the Hyperledger community were to develop a standalone runtime instead of adopting Docker, the problems of runtime can be eliminated. Meanwhile, these problems can be solved in the following way:

- Malicious Image: It is not practical to forbid the user from changing the tag of images, however, the Hyperledger system should regulate all the invocation of containers to be bound with hashes instead of tags. Each time a container image is called, the hash needs to be compared, including the on-the-fly generated chaincode images.

- Access Control: If it is not necessary, all the Hyperledger containers should be run in root-less mode. This may need additional libraries installed on the host and further supports from Docker community [144]. This can limit the adversary from performing harmful operations that need root privileges in containers, for example, access the private keys.

- Communication Security: The problem of communication security is twofold, the confidential cryptographic material loaded into containers, and the use of containers adds an additional communication layer between the peer and installed chaincode. For the first issue, if one malicious image is loaded during any execution of chaincode in a blockchain, the adversary can obtain the access to the private keys for further communication manipulation. This problem is a side-effect of malicious image and loose access control, and the problem can be fixed only if the previous two are properly handled. However, communication between Peer and chaincode can be different. Note that, the chaincode is installed locally per Peer, which means the *Chaincode* container and *Peer* container are running in the same machine. The communication between any peer and orderers may be remote network connections, but for the local system data exchanging between *Chaincode* container and *Peer* container, the network-based communication is not the only option. Docker supports shared memory (inter-process communication (IPC) namespace) for inter-container data exchange. Instead of using network communication, using IPC between *Chaincode* container and *Peer* container can eliminate the risk in TLS based communication.

5.3.3 Threat Model of Smart Contract Systems

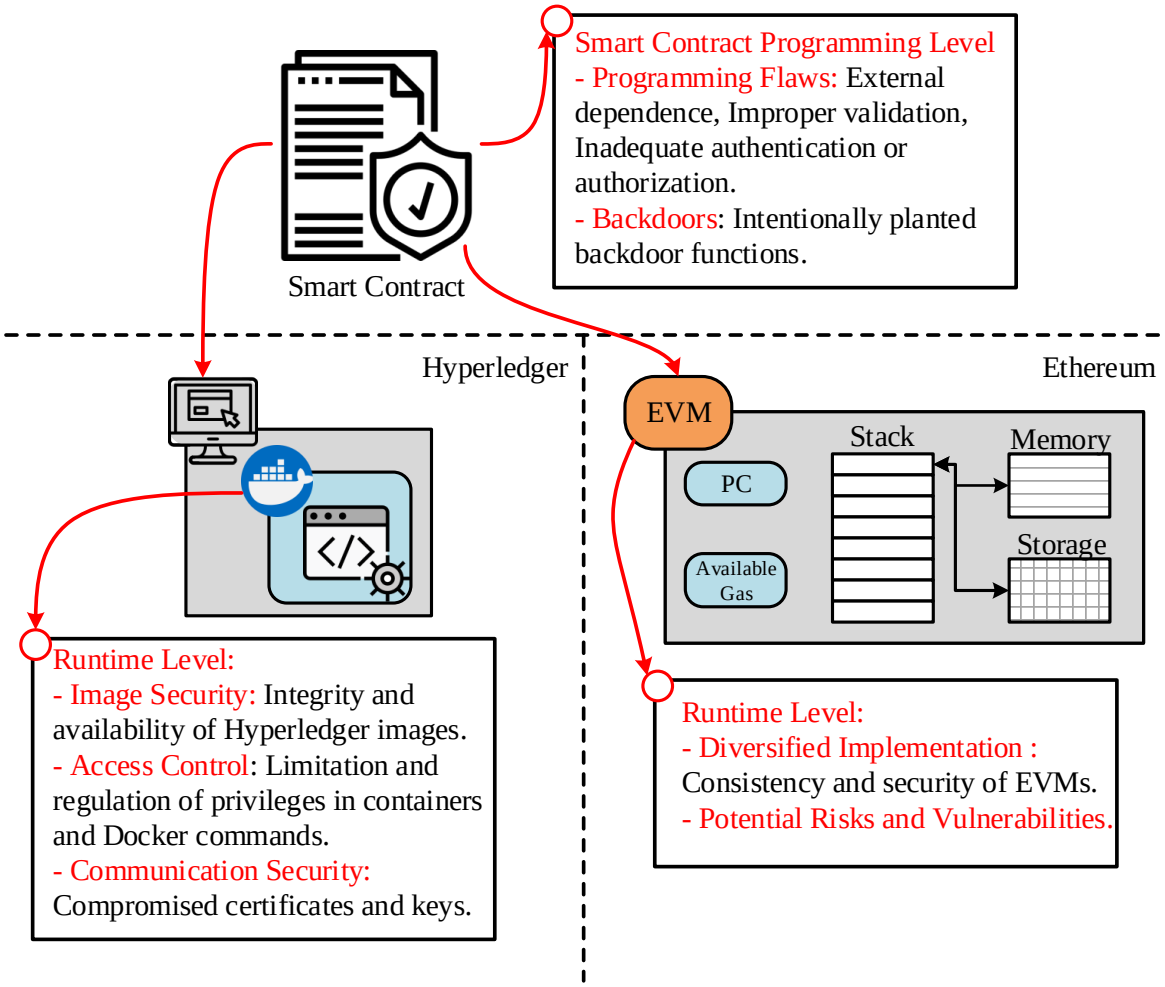


Figure 5.8: Threat Model of Smart Contract System

Based on the findings in this paper, a new threat model of smart contract system is established and shown in Figure 5.8. Generally, the threats can be divided into two levels, smart contract programming level [106], and runtime level. The threats in smart contract programming level are either caused by programming flaws or backdoors. The programming flaws including external dependence (e.g., reentrancy, delegatecall injection), improper validations (e.g., integer overflow and underflow), and inadequate authentication or authorization (e.g., erroneous visibility, unprotected suicide). The backdoors are intentionally planted malicious functions, they may not violate any programming rules or fall in any programming flaw definitions. However, they can be used to trigger blockchain operations that can potentially prejudice the interests of others.

The runtime level threats in Hyperledger-based blockchain are bound to Docker container security. The integrity and availability of Hyperledger images significantly determine the security of the chaincode system. In addition, if the access control in Docker environment is not appropriately configured, the confidential information, such as, keys and certificates can be accessed without authorization. The Docker commands that can be issued as a non-root user can also harm the chaincode system. For example, an evil insider can easily stop the containers to perturb and even DoS the blockchain system (endorsement fail, ordering service fail). Nevertheless, the communication between Peer and chaincode container leaves an open window for the adversary. The chaincode system can be manipulated if the keys and certificates are compromised and the communication under the control of the MITM.

On the other hand, the Ethereum-based smart contract system suffers from the diversified EVM implementations. Hyperledger adopts a single standardized runtime Docker as the universal chaincode runtime, whereas Ethereum provides a different implementation of EVMs. As stated in [137], the gas and opcode consistency problem have been already found in different EVM implementations. Moreover, maintaining and ensuring the security of all the EVM implementations are non-trivial and challenge task. The attack in CVE-2018-18920 just utilizes the flaws in the Python version of EVM implementation. Some of the other potential risks are also indicated in the work [138]. As the development and maintenance of EVM continue, the security of Ethereum runtime needs more concern.

Chapter 6

Conclusion

Containerization is one of the emerging techniques with increasing popularity in many application domains due to the customizable, manageable, and lightweight features. However, the dark side of containerization is the arguable weak security. The DevSecOps of containerization environment significantly determines the overall security of cloud computing systems and some of the blockchains. A malicious container in the cloud system can be used as the attack launching point for DDoS attack, and the underlying containerization vulnerabilities such as Docker escape can lead to more severe system damages. Meanwhile, the loose management of Docker containers in Hyperledger blockchains make the system more vulnerable.

We present a comprehensive literature review on virtualization and containerization techniques. The anomaly detection and secure monitoring of virtualization and containerization environment has been discussed and analyzed as well. In order to provide secure and noninvasive monitoring of containerized applications in the cloud environment, we propose to use unsupervised machine learning based monitoring with containerization introspection tools. A proof of concept (PoC) system is implemented as the baseline to verify the proposed concept. A detailed design and implementation of the monitoring framework is established based on the success of PoC. The implementation includes a novel containerization-oriented and system call traces based dataset with multiple different attacks involved. Our proposed framework provides reliable monitoring and anomaly detection capability for the containerized application.

In addition, we elaborate the background of blockchain and smart contract systems, and we also illustrate the basic concepts and life cycles of containerization based smart contract systems. We conduct a case study on Hyperledger Farbic blockchain to reveal the potential vulnerabilities in the smart contract runtime level, whereas the mainstream only considers the security of smart contract at the programming

level. We proved that the underlying Docker environment of Hyperledger system may be vulnerable to certain attacks. The additional communication between containers and the loaded cryptographic materials in containers leave a hole for the adversary to launch MITM attacks. The loose management of Docker images can further interfere the entire smart contract system as well. Therefore, we also demonstrate the countermeasures and create a new threat model of smart contract systems to thwart the threats.

As the future work, we will extend the secure monitoring framework with more features and evaluate the framework in online mode. We aim to evaluate the contribution of each features in the security monitoring goal, which can help the framework to automatically determine what to monitor in an unsupervised fashion. We also plan to scale the dataset with different application architectures such as multi-containers applications where a service is formed by a collection of containers. Introspection on this type of services require the monitoring system to have an overview of all the containers and meanwhile having the capability to understand and interpret the relationships between the containers. In addition, distributed monitoring in Docker Swarm with different monitoring tools and features is interested as well.

Acknowledgment

This research was supported in part by Progeny Systems, Inc. (Grant G0001080, Progeny-Psc-0342 Navy-N00253-16-C-0007 COTS Approach To Information Security).

Bibliography

[1] A. Lenk, M. Klems, J. Nimis, S. Tai, and T. Sandholm, "What's inside the cloud? an architectural map of the cloud landscape," in *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, ser. CLOUD '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 23–31. [Online]. Available: https://doi.org/10.1109/CLOUD.2009.5071529

[2] M. Pearce, S. Zeadally, and R. Hunt, "Virtualization: Issues, security threats, and solutions," *ACM Comput. Surv.*, vol. 45, no. 2, pp. 17:1–17:39, Mar. 2013. [Online]. Available: http://doi.acm.org/10.1145/2431211.2431216

[3] "I/o architectures for virtualization - vmware." [Online]. Available: http://download3.vmware.com/vmworld/2006/tac0080.pdf

[4] C. Pahl, "Containerization and the paas cloud," *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, May 2015.

[5] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Comput. Surv.*, vol. 41, no. 3, pp. 15:1–15:58, Jul. 2009.

[6] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *2011 IEEE Symposium on Security and Privacy*, May 2011, pp. 297–312.

[7] P. Mell and T. Grance, "The nist definition of cloud computing: Recommendations of the national institute of standards and technology," *NIST Special Publication*, 2011. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf

[8] Gartner Research, "Gartner forecasts worldwide public cloud services revenue to reach $260 billion in 2017," 2017, https://www.gartner.com/newsroom/id/3815165.

[9] A. Singh and K. Chatterjee, "Cloud security issues and challenges: A survey," *Journal of Network and Computer Applications*, vol. 79, pp. 88 – 115, 2017.

[10] Z. Xiao and Y. Xiao, "Security and privacy in cloud computing," *IEEE Communications Surveys Tutorials*, vol. 15, no. 2, pp. 843–859, Second 2013.

[11] R. Cogranne, G. Doyen, N. Ghadban, and B. Hammi, "Detecting botclouds at large scale: A decentralized and robust detection method for multi-tenant virtualized environments," *IEEE Transactions on Network and Service Management*, vol. 15, no. 1, pp. 68–82, March 2018.

[12] Microsoft, "The identity security and protection team has seen a 300 percent increase in user accounts attacked over the past year," 2017, https://www.microsoft.com/en-us/security/Intelligence-report.

[13] B. Tak, C. Isci, S. Duri, N. Bila, S. Nadgowda, and J. Doran, "Understanding security implications of using containers in the cloud," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 313–319.

[14] "Container market will reach 2.7 billion in 2020." [Online]. Available: https://451research.com/images/Marketing/press_releases/Application-container-market-will-reach-2-7bn-in-2020_final_graphic.pdf

[15] "Application container market." [Online]. Available: https://www.marketsandmarkets.com/Market-Reports/application-container-market-182079587.html#:~:text=Application%20containers%20usually%20work%20on,32.9%25%20during%20the%20forecast%20period

[16] T. Bui, "Analysis of docker security," *CoRR*, vol. abs/1501.02967, 2015.

[17] T. Combe, A. Martin, and R. D. Pietro, "To docker or not to docker: A security perspective," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 54–62, Sept 2016.

[18] V. V. Sarkale, P. Rad, and W. Lee, "Secure cloud container: Runtime behavior monitoring using most privileged container (mpc)," in *2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud)*, June 2017, pp. 351–356.

[19] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *In Proc. Network and Distributed Systems Security Symposium*, 2003, pp. 191–206.

[20] “Kdd cup 1999 data,” http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html.

[21] “1999 darpa intrusion detection evaluation dataset,” https://www.ll.mit.edu/r-d/datasets/1999-darpa-intrusion-detection-evaluation-dataset.

[22] “Unm sequence-based intrusion detection dataset,” https://www.cs.unm.edu/~immsec/systemcalls.htm.

[23] “Intrusion detection evaluation dataset (cicids2017),” https://www.unb.ca/cic/datasets/ids-2017.html.

[24] “Intrusion detection evaluation dataset (iscxids2012),” https://www.unb.ca/cic/datasets/ids.html.

[25] “Cse-cic-ids2018 on aws,” https://www.unb.ca/cic/datasets/ids-2018.html.

[26] “Intelligence and security informatics data sets,” https://www.azsecure-data.org/other-data.html.

[27] “Adfa-ids-datasets,” https://www.unsw.adfa.edu.au/unsw-canberra-cyber/cybersecurity/ADFA-IDS-Datasets/.

[28] J. Sahoo, S. Mohapatra, and R. Lath, “Virtualization: A survey on concepts, taxonomy and associated security issues,” in *2010 Second International Conference on Computer and Network Technology*, April 2010, pp. 222–226.

[29] “Kvm: Kernel virtual machine.” [Online]. Available: https://www.linux-kvm.org/page/FAQ

[30] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the art of virtualization,” *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 164–177, Oct. 2003. [Online]. Available: http://doi.acm.org/10.1145/1165389.945462

[31] “Vmware vsphere 4 - esx and vcenter server documentation center.” [Online]. Available: https://pubs.vmware.com/vsphere-4-esx-vcenter/index.jsp

[32] “Intel virtualization technology processor virtualization extensions and intel trusted execution technology.” [Online]. Available: https://software.intel.com/sites/default/files/m/0/2/1/b/b/1024-Virtualization.pdf

[33] “Amd secure virtual machine architecture reference manual.” [Online]. Available: https://www.mimuw.edu.pl/~vincent/lecture6/sources/amd-pacifica-specification.pdf

[34] J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[35] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs containerization to support paas," in *2014 IEEE International Conference on Cloud Engineering*, March 2014.

[36] R. Morabito, J. Kjllman, and M. Komu, "Hypervisors vs. lightweight virtualization: A performance comparison," in *2015 IEEE International Conference on Cloud Engineering*, March 2015, pp. 386–393.

[37] J. Aikat, A. Akella, J. S. Chase, A. Juels, M. Reiter, T. Ristenpart, V. Sekar, and M. Swift, "Rethinking security in the era of cloud computing," *IEEE Security Privacy*, pp. 1–1, 2017.

[38] D. Puthal, B. P. S. Sahoo, S. Mishra, and S. Swain, "Cloud computing features, issues, and challenges: A big picture," in *2015 International Conference on Computational Intelligence and Networks*, Jan 2015, pp. 116–123.

[39] Google Doc, https://www.google.com/docs/about/.

[40] Saleforce CRM, https://www.salesforce.com/crm/.

[41] Google App Engine, https://cloud.google.com/appengine/.

[42] Microsoft Azure, https://azure.microsoft.com/en-us/.

[43] Amazon EC2, https://aws.amazon.com/ec2/.

[44] G. Kim, S. Lee, and S. Kim, "A novel hybrid intrusion detection method integrating anomaly detection with misuse detection," *Expert Systems with Applications*, vol. 41, no. 4, Part 2, pp. 1690 – 1700, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0957417413006878

[45] A. Sari, "A review of anomaly detection systems in cloud networks and survey of cloud security measures in cloud storage applications," pp. 142 – 154, 2015.

[46] E. Bauman, G. Ayoade, and Z. Lin, "A survey on hypervisor-based monitoring: Approaches, applications, and evolutions," *ACM Comput. Surv.*, vol. 48, no. 1, pp. 10:1–10:33, Aug. 2015.

[47] N. L. Petroni, Jr. and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 103–115. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315260

[48] K. Nance, M. Bishop, and B. Hay, "Virtual machine introspection: Observation or interference?" *IEEE Security Privacy*, vol. 6, no. 5, pp. 32–37, Sept 2008.

[49] H.-J. Liao, C.-H. R. Lin, Y.-C. Lin, and K.-Y. Tung, "Intrusion detection system: A comprehensive review," *Journal of Network and Computer Applications*, vol. 36, no. 1, pp. 16 – 24, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1084804512001944

[50] P. Mishra, E. S. Pilli, V. Varadharajan, and U. Tupakula, "Intrusion detection techniques in cloud environment: A survey," *Journal of Network and Computer Applications*, vol. 77, pp. 18 – 47, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1084804516302417

[51] B. Dolan-Gavitt, T. Leek, J. Hodosh, and W. Lee, "Tappan zee (north) bridge: mining memory accesses for introspection," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 839–850. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516697

[52] "Libvmi," http://libvmi.com/.

[53] C. Ramaswamy, "Security assurance requirements for linux application container deployments," 2017. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/ir/2017/NIST.IR.8176.pdf

[54] SELinux, http://www.selinuxproject.org/.

[55] AppAmor, https://help.ubuntu.com/lts/serverguide/apparmor.html.en.

[56] "Mesos." [Online]. Available: http://mesos.apache.org/documentation/latest/monitoring/

[57] "Ceilometer." [Online]. Available: http://docs.openstack.org/developer/ceilometer/.

[58] F. Moradi, C. Flinta, A. Johnsson, and C. Meirosu, "Conmon: An automated container based network performance monitoring system," in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, May 2017, pp. 54–62.

[59] Y. Zhu, J. Ma, B. An, and D. Cao, "Monitoring and billing of a lightweight cloud system based on linux container," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, June 2017, pp. 325–329.

[60] "cadvisor." [Online]. Available: https://github.com/google/cadvisor

[61] "Sysdig," https://github.com/draios/sysdig/.

[62] N. M. Nasrabadi, "Pattern recognition and machine learning," *Journal of electronic imaging*, vol. 16, no. 4, p. 049901, 2007.

[63] D. F. Specht, "A general regression neural network," *IEEE transactions on neural networks*, vol. 2, no. 6, pp. 568–576, 1991.

[64] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[65] J. A. Suykens and J. Vandewalle, "Least squares support vector machine classifiers," *Neural processing letters*, vol. 9, no. 3, pp. 293–300, 1999.

[66] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE transactions on information theory*, vol. 13, no. 1, pp. 21–27, 1967.

[67] I. Rish *et al.*, "An empirical study of the naive bayes classifier," in *IJCAI 2001 workshop on empirical methods in artificial intelligence*, vol. 3, no. 22. IBM New York, 2001, pp. 41–46.

[68] S. Mika, G. Ratsch, J. Weston, B. Scholkopf, and K.-R. Mullers, "Fisher discriminant analysis with kernels," in *Neural networks for signal processing IX, 1999. Proceedings of the 1999 IEEE signal processing society workshop.* Ieee, 1999, pp. 41–48.

[69] E. Alpaydin, *Introduction to machine learning*. MIT press, 2009.

[70] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[71] "Ai winter." [Online]. Available: https://en.wikipedia.org/wiki/AI_winter

[72] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[73] C.-W. Hsu and C.-J. Lin, "A comparison of methods for multiclass support vector machines," *IEEE transactions on Neural Networks*, vol. 13, no. 2, pp. 415–425, 2002.

[74] L. E. Peterson, "K-nearest neighbor," *Scholarpedia*, vol. 4, no. 2, p. 1883, 2009.

[75] H. Anton, *Elementary Linear Algebra*. John Wiley & Sons, 2010. [Online]. Available: https://books.google.com/books?id=YmcQJoFyZ5gC

[76] D. F. Specht, "A general regression neural network," *IEEE transactions on neural networks*, vol. 2, no. 6, pp. 568–576, 1991.

[77] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "Lstm: A search space odyssey," *IEEE transactions on neural networks and learning systems*, vol. 28, no. 10, pp. 2222–2232, 2017.

[78] P. Casas, J. Mazel, and P. Owezarski, "Unsupervised network intrusion detection systems: Detecting the unknown without knowledge," *Computer Communications*, vol. 35, no. 7, pp. 772–783, 2012.

[79] M. Amer, M. Goldstein, and S. Abdennadher, "Enhancing one-class support vector machines for unsupervised anomaly detection," in *Proceedings of the ACM SIGKDD Workshop on Outlier Detection and Description*, 2013, pp. 8–15.

[80] H. Xu, W. Chen, N. Zhao, Z. Li, J. Bu, Z. Li, Y. Liu, Y. Zhao, D. Pei, Y. Feng *et al.*, "Unsupervised anomaly detection via variational auto-encoder for seasonal kpis in web applications," in *Proceedings of the 2018 World Wide Web Conference*, 2018, pp. 187–196.

[81] T. Kohonen, "The self-organizing map," *Proceedings of the IEEE*, vol. 78, no. 9, pp. 1464–1480, 1990.

[82] D. J. Dean, H. Nguyen, and X. Gu, "Ubl: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems," in *Proceedings of the 9th international conference on Autonomic computing*, 2012, pp. 191–200.

[83] J. Dromard, G. Roudière, and P. Owezarski, "Online and scalable unsupervised network anomaly detection method," *IEEE Transactions on Network and Service Management*, vol. 14, no. 1, pp. 34–47, 2016.

[84] S. Ahmad, A. Lavin, S. Purdy, and Z. Agha, "Unsupervised real-time anomaly detection for streaming data," *Neurocomputing*, vol. 262, pp. 134–147, 2017.

[85] B. Dolan-Gavitt, T. Leek, J. Hodosh, and W. Lee, "Tappan zee (north) bridge: mining memory accesses for introspection," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 839–850.

[86] Y.-H. Li, Y.-R. Tzeng, and F. Yu, "Viso: Characterizing malicious behaviors of virtual machines with unsupervised clustering," in *2015 iEEE 7th international conference on cloud computing technology and science (cloudCom)*. IEEE, 2015, pp. 34–41.

[87] M. R. Memarian, M. Conti, and V. Leppänen, "Eyecloud: A botcloud detection system," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1. IEEE, 2015, pp. 1067–1072.

[88] A. S. Abed, T. C. Clancy, and D. S. Levy, "Applying bag of system calls for anomalous behavior detection of applications in linux containers," in *2015 IEEE Globecom Workshops*. IEEE, 2015.

[89] Q. Du, T. Xie, and Y. He, "Anomaly detection and diagnosis for container-based microservices with performance monitoring," in *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 2018, pp. 560–572.

[90] A. Samir and C. Pahl, "Anomaly detection and analysis for clustered cloud computing reliability," *CLOUD COMPUTING 2019*, p. 120, 2019.

[91] T. Watts, R. Benton, W. Glisson, and J. Shropshire, "Insight from a docker container introspection," in *Proceedings of the 52nd Hawaii International Conference on System Sciences*, 2019.

[92] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system, http://bitcoin.org/bitcoin.pdf," 2008.

[93] Anonymous. White paper: Next-generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper.

[94] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. D. Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolic, S. W. Cocco, and J. Yellick, "Hyperledger fabric: A distributed operating system for permissioned blockchains," 2018.

[95] "Filecoin." [Online]. Available: https://filecoin.io/

[96] "Storj." [Online]. Available: https://storj.io/

[97] R. Neisse, G. Steri, and I. Nai-Fovino, "A blockchain-based approach for data accountability and provenance tracking," in *Proceedings of the 12th International Conference on Availability, Reliability and Security*. ACM, 2017, p. 14.

[98] Z. Zheng, S. Xie, H.-N. Dai, and H. Wang, "Blockchain challenges and opportunities: A survey," *Work Pap.–2016*, 2016.

[99] I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse, "Bitcoin-ng: A scalable blockchain protocol," in *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, ser. NSDI'16, 2016.

[100] X. Li, P. Jiang, T. Chen, X. Luo, and Q. Wen, "A survey on the security of blockchain systems," *Future Generation Computer Systems*, 2017.

[101] "Bitcoin confirmation." [Online]. Available: https://en.bitcoin.it/wiki/Confirmation

[102] L. Lamport, R. E. Shostak, and M. C. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, 1982.

[103] M. O. T. de Castro, "Practical byzantine fault tolerance," in *OSDI*, 1999.

[104] G. Wood. Yellow paper: Ethereum: A secure decentralised generalised transaction ledger. https://github.com/ethereum/yellowpaper.

[105] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *Principles of Security and Trust*, M. Maffei and M. Ryan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 164–186.

[106] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A survey on ethereum systems security: Vulnerabilities, attacks and defenses," *ACM Comput. Surv.*, vol. 0, no. ja, 2020. [Online]. Available: https://doi.org/10.1145/3391195

[107] "The dao attacked: Code issue leads to $60 million ether theft," https://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft/.

[108] "King of the ether throne refund issue," http://www.kingoftheether.com/postmortem.html.

[109] "Backdoor flaw sees australian firm lose $6.6 million in cryptocurrency." [Online]. Available: {https://finance.yahoo.com/news/backdoor-flaw-sees-australian-firm-115323212.html}

[110] "Bancor unchained: All your token are belong to us." [Online]. Available: {https://medium.com/unchained-reports/bancor-unchained-all-your-token-are-belong-to-us-d6bb00871e86}

[111] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 67–82.

[112] D. Park, Y. Zhang, M. Saxena, P. Daian, and G. Roşu, "A formal verification tool for ethereum vm bytecode," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 912–915.

[113] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu *et al.*, "Kevm: A complete formal semantics of the ethereum virtual machine," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 2018, pp. 204–217.

[114] E. Zhou, S. Hua, B. Pi, J. Sun, Y. Nomura, K. Yamashita, and H. Kurihara, "Security assurance for smart contract," in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE, 2018, pp. 1–5.

[115] M. Wohrer and U. Zdun, "Smart contracts: security patterns in the ethereum ecosystem and solidity," in *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2018, pp. 2–8.

[116] D. Perez and B. Livshits, "Smart contract vulnerabilities: Does anyone care?" *arXiv preprint arXiv:1902.06710*, 2019.

[117] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," 2018.

[118] A. R. Manu, J. K. Patel, S. Akhtar, V. K. Agrawal, and K. N. B. S. Murthy, "Docker container security via heuristics-based multilateral security-conceptual and pragmatic study," in *2016 International Conference on Circuit, Power and Computing Technologies (ICCPCT)*, March 2016, pp. 1–14.

[119] S. Sultan, I. Ahmad, and T. Dimitriou, "Container security: Issues, challenges, and the road ahead," *IEEE Access*, vol. 7, pp. 52 976–52 996, 2019.

[120] M. Liu, Z. Xue, X. Xu, C. Zhong, and J. Chen, "Host-based intrusion detection system with system calls: Review and future trends," *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, pp. 1–36, 2018.

[121] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014.

[122] "Cve-2019-5736," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-5736.

[123] "Cve-2019-5736-poc," https://github.com/Frichetten/CVE-2019-5736-PoC.

[124] "Virusshare," https://virusshare.com/.

[125] "Virustotal," https://www.virustotal.com/.

[126] X. Zhang, Y. Zou, S. Li, and S. Xu, "A weighted auto regressive lstm based approach for chemical processes modeling," *Neurocomputing*, vol. 367, pp. 64–74, 2019.

[127] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, 1997.

[128] A. H. Mirza and S. Cosan, "Computer network intrusion detection using sequential lstm neural networks autoencoders," in *2018 26th Signal Processing and Communications Applications Conference (SIU)*. IEEE, 2018, pp. 1–4.

[129] D. Liu, Y. Zhao, H. Xu, Y. Sun, D. Pei, J. Luo, X. Jing, and M. Feng, "Opprentice: Towards practical and automatic anomaly detection through machine learning," in *Proceedings of the 2015 Internet Measurement Conference*, 2015, pp. 211–224.

[130] "Ethereum virtual machine opcodes." [Online]. Available: https://ethervm.io/

[131] "Ethereum smart contract languages." [Online]. Available: https://ethereum.org/developers/

[132] A. P. Sarr, P. Elbaz-Vincent, and J.-C. Bajard, "A secure and efficient authenticated diffie–hellman protocol," in *European Public Key Infrastructure Workshop*. Springer, 2009, pp. 83–98.

[133] N. Li, "Research on diffie-hellman key exchange protocol," in *2010 2nd International Conference on Computer Engineering and Technology*, vol. 4. IEEE, 2010, pp. V4–634.

[134] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta *et al.*, "Imperfect forward secrecy: How diffie-hellman fails in practice," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 5–17.

[135] "Cve-2019-5736-poc-malicious-image." [Online]. Available: https://github.com/twistlock/RunC-CVE-2019-5736

[136] "Backdoored images downloaded 5 million times." [Online]. Available: https://arstechnica.com/information-technology/2018/06/backdoored-images-downloaded-5-million-times-finally-removed-from-docker-hub/

[137] Y. Fu, M. Ren, F. Ma, Y. Jiang, H. Shi, and J. Sun, "Evmfuzz: Differential fuzz testing of ethereum virtual machine," 2019.

[138] Y. Fu, M. Ren, F. Ma, H. Shi, X. Yang, Y. Jiang, H. Li, and X. Shi, "Evmfuzzer: detect evm vulnerabilities via fuzz testing," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 1110–1114.

[139] "Cve-2018-18920." [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-18920

[140] “Cve-2018-19183.” [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-19183

[141] “Cve-2018-19184.” [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-19184

[142] “Cve-2018-19330.” [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-19330

[143] “Cve-2019-7710.” [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-7710

[144] “Run the docker daemon as a non-root user (rootless mode).” [Online]. Available: https://docs.docker.com/engine/security/rootless/