IEEE Access
Multidisciplinary ⁝ Rapid Review ⁝ Open Access Journal

Received February 18, 2019, accepted April 10, 2019, date of publication April 17, 2019, date of current version May 1, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2911732

Container Security: Issues, Challenges, and the Road Ahead

SARI SULTAN, (Student Member, IEEE), IMTIAZ AHMAD, AND TASSOS DIMITRIOU, (Senior Member, IEEE)
Department of Computer Engineering, Kuwait University, Safat 13060, Kuwait

Corresponding author: Imtiaz Ahmad (imtiaz.ahmad@ku.edu.kw)

ABSTRACT Containers emerged as a lightweight alternative to virtual machines (VMs) that offer better microservice architecture support. The value of the container market is expected to reach $2.7 billion in 2020 as compared to $762 million in 2016. Although they are considered the standardized method for microservices deployment, playing an important role in cloud computing emerging fields such as service meshes, market surveys show that container security is the main concern and adoption barrier for many companies. In this paper, we survey the literature on container security and solutions. We have derived four generalized use cases that should cover security requirements within the host-container threat landscape. The use cases include: (I) protecting a container from applications inside it, (II) inter-container protection, (III) protecting the host from containers, and (IV) protecting containers from a malicious or semi-honest host. We found that the first three use cases utilize a software-based solutions that mainly rely on Linux kernel features (e.g., namespaces, CGroups, capabilities, and seccomp) and Linux security modules (e.g., AppArmor). The last use case relies on hardware-based solutions such as trusted platform modules (TPMs) and trusted platform support (e.g., Intel SGX). We hope that our analysis will help researchers understand container security requirements and obtain a clearer picture of possible vulnerabilities and attacks. Finally, we highlight open research problems and future research directions that may spawn further research in this area.

INDEX TERMS Containers, Docker, Linux containers, OS level virtualization, lightweight virtualization, security, survey.

I. INTRODUCTION

Virtual machines (VMs) provide excellent security. However, their security isolation creates a bottleneck for the total number of VMs that can run on a server because each VM should have its own copy of the operating system (OS), libraries, dedicated resources, and applications. This has a detrimental effect on performance (e.g., long startup time) and storage size. The advent of DevOps software development practice [1], [2] and microservices underscored the need for a faster solution than VMs as it is not efficient to run each microservice on a separate VM due to its long startup time and increased resource usage.

Container-based virtualization emerged as a lightweight alternative to VMs. Many containers can share the same OS kernel instead of having a dedicated copy for each one as in VMs. This greatly reduces startup time and the required resources for each image. For example, a container can start in 50 milliseconds while a VM might take as long as 30–40 seconds to start [3]. Many container technologies are available such as LXC, OpenVZ, Linux-Vserver, with Docker being the predominant one. Figure 1a shows a simple container's architecture while Figure 1b shows a simple VM's architecture. Containers are a more plausible option for microservices than VMs due to numerous benefits such as being lightweight, fast, easier to deploy, and allowing for better resource utilization and version control. Containers are being used for different applications such as Internet of Things (IoT) services, smart cars, fog computing, service meshes, and so on [3]–[8].

The introduction of microservice architectures helped increase software agility, wherein software parts became independent units of development, versioning, deployment, and scaling [8]. Microservices are used by numerous organizations such as Amazon, Spotify, Netflix, and Twitter to deliver their software [9]. Containers are considered the standard to deploy microservices and applications to the cloud [10]. Containers are also important for the future of cloud computing and their market value is expected to reach $2.7 billion by 2020 (was $762 million in 2016) [11], [12].

The associate editor coordinating the review of this manuscript and approving it for publication was Kashif Saleem.

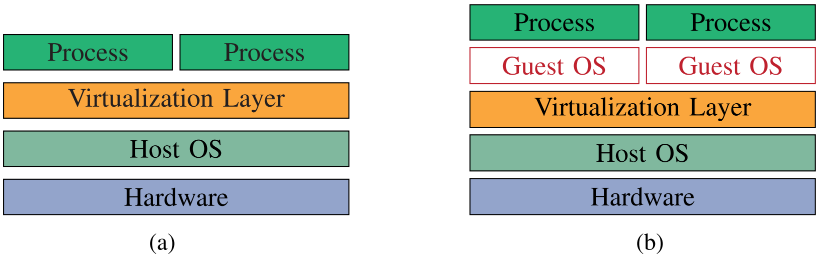

FIGURE 1. Comparison between container and hypervisor. (a) Container. (b) Virtual machine.

However, containers are less secure than VMs [13]–[15]. Hence, security is the main barrier to widespread container adoption [16].

Although there are several surveys that address VMs, they do not focus on container security issues [17]–[19]. As containers are based on sharing the OS kernel among them while each VM has its own kernel, container security is different from its VM counterpart since it is based on different architectural support. Hence, understanding container security threats and solutions is very important due to the lack of systematic reviews about them in the literature. This is problematic because each of the solutions presented pertains to a very specific use case. For example, trusted platform support (e.g., Intel Software Guard Extensions (SGX)) is used primarily to allow the running of containers on an untrusted host, thus tracking those different use cases can be frustrating for the reader.

In this work, we provide four general use cases that should cover most use cases for the host-container level. This should help readers better understand security issues about containers and the available mechanisms to secure them. The use cases are: (I) protecting a container from applications inside it, (II) inter-container protection, (III) protecting the host from containers, and (IV) protecting containers from a malicious or semi-honest host. We discuss the available software-based solutions that are typically used for the first three use cases and hardware-based solutions that are used for the last one. Our threat model for the four use cases could be used by researchers to enhance their understanding of possible vulnerabilities and attacks and to clearly illustrate what their solutions provide. Finally, we highlight open problems and future research directions in order to motivate further work in this exciting area.

The rest of this paper is organized as follows. Section II discusses background material, relevant resources, and selection criteria. In Section III, we present our threat model and the proposed use cases. In Section IV, we present the software and hardware protection mechanisms used to secure containers. In particular, Section IV-A presents software-based mechanisms while Section IV-B presents hardware-based ones. A discussion on container vulnerabilities, exploits, discovery tools, and relevant standards is presented in Section V. In Section VI, we discuss future research directions and open issues. Finally, Section VII concludes this paper.

II. BACKGROUND

In this section, we present background material on containers as well as monolithic and microservice architectures. Section II-A provides background details on containers while Section II-B considers monolithic and microservice architectures. Section II-C presents our selected resources and selection criteria.

A. CONTAINERS

Different names are used to refer to containers in the literature including OS level virtualization and lightweight virtualization. Docker, LXC, and RKT are examples of container managers. Many studies focus on Docker because it is the predominant container runtime environment. Hardware virtualization refers to traditional VMs and hypervisors.

Containers improve two main downsides of VMs [14]: first, they share the same OS kernel and can share resources while each VM needs its own copy. Second, containers can be started and stopped almost instantly while VMs need considerable time to start [3]. Containers have also proven to be more efficient than VMs for some applications such as microservices because they are lightweight and do not require a full OS copy for each image. However, containers still need a fully functional kernel that is shared among different containers. Additionally, microservice design underscores the importance of ephemeral state containers, wherein any data persistence goes to another data store or service. Containers are considered the standard way to deploy microservices to the cloud [10].

Container as a Service (CaaS or CoaaS) creates a new delivery model for cloud computing [3]. Many companies offer container services which allow a wide variety of containerized applications for several marketplaces [20]. Although OS level virtualization is a promising technology with many benefits, it faces a large number of challenges. For example, host OS kernel sharing introduces many security issues, which make them less secure than VMs [21].

Figure 1a shows a simple container's architecture. The bird's eye view of a container's stack is necessary because deployment relies on various parts. Figure 2 shows container stack components and realization technologies; it is based on the architecture from [22] which was modified to merge realization technologies and stack components together. This figure shows that a container is a building block

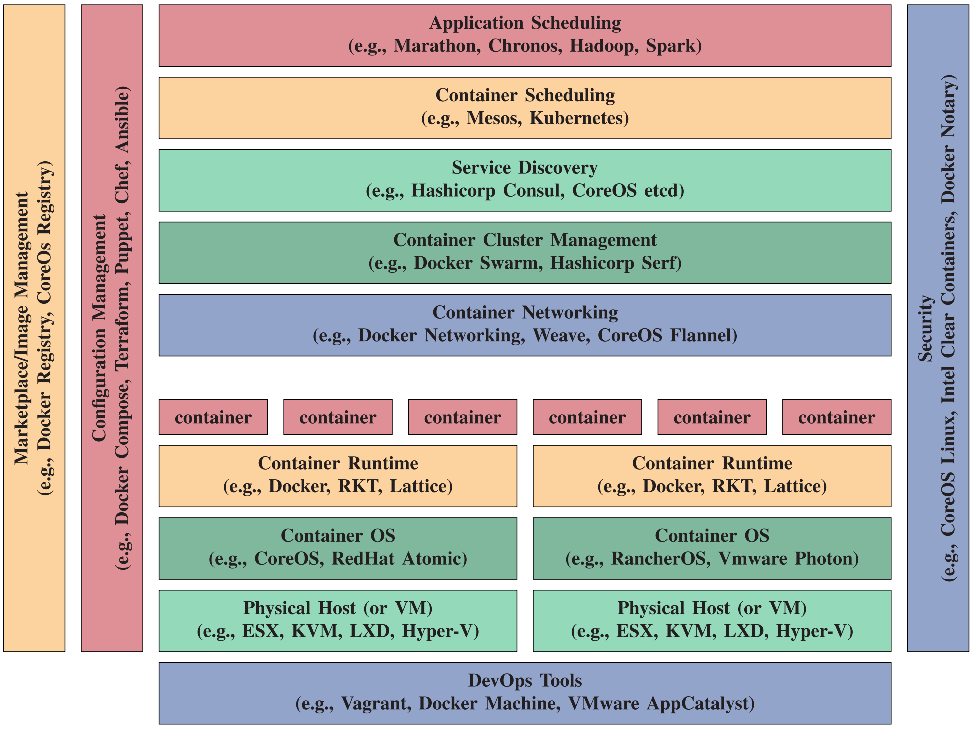
Marketplace/Image Management
(e.g., Docker Registry, CoreOs Registry)
Configuration Management
(e.g., Docker Compose, Terraform, Puppet, Chef, Ansible)
Application Scheduling
(e.g., Marathon, Chronos, Hadoop, Spark)
Container Scheduling
(e.g., Mesos, Kubernetes)
Service Discovery
(e.g., Hashicorp Consul, CoreOS etcd)
Container Cluster Management
(e.g., Docker Swarm, Hashicorp Serf)
Container Networking
(e.g., Docker Networking, Weave, CoreOS Flannel)
container
container
container
container
container
container
Container Runtime
(e.g., Docker, RKT, Lattice)
Container Runtime
(e.g., Docker, RKT, Lattice)
Container OS
(e.g., CoreOS, RedHat Atomic)
Container OS
(e.g., RancherOS, Vmware Photon)
Physical Host (or VM)
(e.g., ESX, KVM, LXD, Hyper-V)
Physical Host (or VM)
(e.g., ESX, KVM, LXD, Hyper-V)
DevOps Tools
(e.g., Vagrant, Docker Machine, VMware AppCatalyst)
Security
(e.g., CoreOS Linux, Intel Clear Containers, Docker Notary)

FIGURE 2. Container stack and realization technologies.

for a larger technology stack that can be used to facilitate microservices deployment. Peinl *et al.* [23] presented a survey on the tools available for container management. They classified different solutions in both the academic and industry literature as well as mapping them to requirements based on a case study they provided. Additionally, they identified gaps in these tools and integration requirements and proposed their own tools in order to overcome these deficiencies.

B. MONOLITHIC AND MICROSERVICES ARCHITECTURES

A monolithic application refers to software whose parts are strongly coupled and cannot be executed independently [24]. Although monolithic applications can run inside a container, it is highly recommended to use microservice architecture when using containers [25]. Before the advent of Service Oriented Architectures (SOAs), and specifically microservices, most applications used to be monolithic. On the other hand, microservices help build applications that consist of loosely coupled parts that can be operated independently.

Microservice architectures have revolutionized how applications are built nowadays. They allow developers to be more innovative and open to new technologies. For example, if a company wants to experiment with a new programming language, they can build a single microservice with that language, which will have minimal effect on the whole application, unlike using a monolithic architecture, which requires rewriting all parts. Microservices and containers are closely related subjects where containers are considered the standardized deployment option for microservices [10]. Running each microservice in a separate VM is not efficient because VMs are heavy compared to containers [25]. Containers are important alternatives to VMs and they have a number of benefits over them, especially in performance and size. The advent of containers highlighted the importance of microservice architectures over older monolithic architectures. However, containers are afflicted with numerous security issues that are the main barriers for their adoption by companies. Dragoni *et al.* [24] presented a recent study about the emergence of microservices and how their development improved monolithic architecture drawbacks.

C. LITERATURE REVIEW ON CONTAINER SECURITY

In this work, we have included papers from the proceedings of top academic research venues, journals, and books. Occasionally we relied on research that is unpublished or has been published in non-commercial forms such as reports, policy statements, etc. Such articles have been included because research on containers is an inherently practical field that is dominated by the industry and is published about in different online sources. Our selection criteria focused on the following areas related to containers: security features, solutions, threats, vulnerabilities, exploits, tools, standards, evaluation methodologies, applications, and container alternatives. We relied primarily on Google Scholar and mainly used the following search keywords: ‘‘Containers security’’, ‘‘Docker security’’, ‘‘Linux containers’’. Then we removed sources that were generic to containers and did not discuss

TABLE 1. Threat model specifications for apps, containers, and host for the studied use cases. *'Semi'* refers to semi-honest. Apps in semi-honest/malicious containers can be semi-honest or malicious too.

	Apps can be			Containers can be			Host can be		
Use Case	honest	semi	malicious	honest	semi	malicious	honest	semi	malicious
(I) Protect container from applications	✓	✓	✓	✓	-	-	✓	-	-
(II) Inter-container protection	✓	-	-	✓	✓	✓	✓	-	-
(III) Protect host from containers	✓	✓	✓	✓	✓	✓	✓	-	-
(IV) Protect containers from host	✓	-	-	✓	-	-	✓	✓	✓

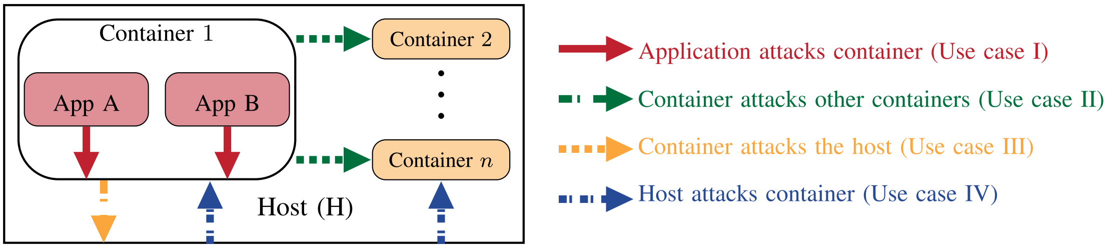
Container 1
App A
App B
Container 2
Container n
Host (H)
Application attacks container (Use case I)
Container attacks other containers (Use case II)
Container attacks the host (Use case III)
Host attacks container (Use case IV)

FIGURE 3. Overview of security protection requirements in containers.

security issues. Finally, we used backward citations from the selected resources to expand our search. Most of the references were published between 2014 and 2018, including 21 papers in 2016, 45 papers in 2017, and 25 papers in 2018. Container security research started gaining momentum in 2015.

III. THREAT MODEL

Many use cases can be derived to illustrate Linux container security issues and solutions. We believe it would be ineffective to create an exhaustive list of all such use cases as it would be frustrating for the reader to keep track of them. Hence, we provided a new taxonomy for use cases for the host-container level to better identify protection requirements and possible solutions. Registry and orchestration are not the primary focus of our research whose aim is to shed light on the host and container levels that represent the primary focus for container technologies.

We consider a host H that has $|\mathcal{C}|$ ($|.|$ denotes the cardinality of a set) number of containers $\mathcal{C} = \{c_1, c_2, \ldots, c_n\}$. Each container $c_i \in \mathcal{C}$ can run at least one application from a set of applications $\mathcal{A}^{c_i} = \{a_1, a_2, \cdots, a_m\}$. Additionally, we assume that the host H and each container ($c_i \in \mathcal{C}$) have limited resources which makes them vulnerable to availability-targeted attacks. We address four use cases, each of which has its own adversarial model and security goals. The threat model specification for applications, containers, and hosts is shown in Table 1. A *semi-honest* adversary is a passive adversary that could cooperate on information gathering but will not deviate from the protocol execution. A *malicious* adversary is an active adversary that might deviate from the protocol specification in order to gather information, cheat, or disrupt the system and target other system's components. The four cases are shown in Figure 3 and are described in the following sections.

1) USE CASE I: PROTECTING A CONTAINER FROM APPLICATIONS INSIDE IT

In this use case, each application within a running container $c_i \in \mathcal{C}$ can be honest, semi-honest, or malicious. We assume that applications cannot break access control policies if set. Additionally, we assume that some applications might require root privileges (or parts of the full root access). We believe this is a very important case because if an application could gain control over the container manager it might be able to target the host system and other containers within the system. However, an attack application might take control of a specific vulnerable container. Our goal is to minimize intra-container attacks. Table 2 shows a list of possible attacks and solutions for this use case.

2) USE CASE II: INTER-CONTAINER PROTECTION

In this use case, we assume that one or more of the containers are semi-honest or malicious, i.e., $\exists \mathcal{CM} \subseteq \mathcal{C}, |\mathcal{CM}| \geq 1$ where $\mathcal{CM}$ denotes a set of semi-honest or malicious containers. These containers can be inside the host H or on different hosts. Being on different hosts is important for the emerging field of service meshes [8]. We assume that applications inside $c_i \in \mathcal{CM}$ can be malicious or semi-honest, too. Although an attacker application might take control of other applications within a container, we assume that applications in honest containers remain honest for this use case because protecting applications from each other is not a container-specific problem.

Following are some attacks that could be executed. A semi-honest container could be able to access confidential data of other containers, learn resource usage patterns, and target the integrity of application information. Furthermore, a malicious container can perform similar attacks to a semi-honest container and can target another container's availability. For example, a malicious container can consume most for the

TABLE 2. Possible attack scenarios for use case (I) (protecting a container from applications in it) and suggested solutions. (CVE stands for Common Vulnerabilities and Exposure).

Threat	Possible Attack	Possible Scenario	Possible Solution
Image vulnerabilities	Remote code execution	Remote code execution using ShellShock vulnerability that affected most Linux systems (CVE-2014-6271).	Periodic vulnerability scanning for images and applications.
Image configuration defects	Unauthorized access	Running an application with unnecessary root privileges will allow the application to have full control over the container.	Running application on the least privilege needed. Linux Capabilities are also used to provide a specific functionality of the root privileges (not all functionalities).
	Network-based intrusions	Enabling remote access (e.g., SSH) would give attackers an opportunity to gain access over the container if not configured properly.	Management on containers should be done through container runtime APIs. SSH and other remote access tools should not be enabled [31].
Embedded malware	Virus, Worm, Trojan, Ransomeware, etc.	Ransomeware gaining access to a container.	Monitoring the processes with anti-malware software. Keep this software up-to-date. Running trusted apps only.
Embedded clear-text secrets	Information disclosure, tampering, privacy issues	Database connection parameters that are stored in clear text in the container can be used by a compromised application to access/edit the database.	Secrets should be stored outside the image. Orchestrator (e.g., Kubernetes) has native management of secrets.
Use of untrusted images	Many attacks since the image is untrusted and can contain various threats such as backdoors	Untrusted images might have a preinstalled backdoor inside them.	Use only trusted images/registries, and verify the images.
Vulnerabilities within container runtime	Privilege escalation and container runtime escape attack	CVE-2017-5123 mentions a vulnerability in Docker runtime that allows applications to modify the capabilities.	Container runtime should be monitored for vulnerabilities and updated periodically.
Application vulnerabilities	Denial-of-Service (DoS)	A vulnerable application (e.g., Apache server with Slowloris vulnerability) can be targeted with different types of attacks one of which is DoS which would target the availability of the container.	Running applications with least privilege possible to reduce the possible damage when compromised. The root filesystem should be read-only. Scanning applications periodically for vulnerabilities.

dedicated host resources to containers in which case it renders other containers useless. Our goal here is to protect containers from each other, a perfect case would be that containers do not know anything about other containers (like VMs) unless required (e.g., network communication). Protection requirements for this are not specific to containers.

One of the high-risk attacks that affect containers is Meltdown. In [26], the authors successfully mounted an attack on Docker, LXC, and OpenVZ. This attack allowed the adversary to leak kernel information from the host OS and all other containers running on the same system. This had a detrimental effect on cloud service providers, wherein a malicious user could access information from all other containers hosted on the system. Spectre [27] is another serious threat to containers, where it tricks other applications into accessing arbitrary locations in their memory. Both Spectre and Meltdown attacks pose a serious threat to containers [26], [27]. Table 3 shows a list of possible attacks and solutions for this use case.

3) USE CASE III: PROTECTING THE HOST (AND THE APPLICATIONS INSIDE IT) FROM CONTAINERS

In this use case, we assume that at least one container is semi-honest or malicious within the host H ($\mathcal{CM} \subseteq \mathcal{C}$, $|\mathcal{CM}| \geq 1$, where $\mathcal{CM}$ denotes a set of semi-honest or malicious containers). We assume that applications that are inside $c_i \in \mathcal{CM}$, as well as applications in honest containers, can be semi-honest or malicious. A semi-honest container can have access to confidential host information or even target its integrity. A malicious container can target the host's availability by consuming its resources. Our goal here is to eliminate any container's ability to target the host components' confidentiality, integrity, and availability. A perfect scenario would be to make containers act as VMs. (Recently, Intel merged its clear containers project with kata containers and claims that kata containers have similar isolation to VMs [28]). Table 4 shows a list of possible attacks and solutions for this use case.

One of the promising applications of containers is to use them to limit resource drainage attacks of software

TABLE 3. Possible attack scenarios for use case (II) (inter-container protection) and suggested solutions.

Threat	Possible Attack	Possible Scenario	Possible Solution
Untrusted images	Attack all containers within the host	Untrusted images pose a serious threat to containers in the host. An untrusted image might have pre-installed vulnerability scanner which can scan all images in the network to find vulnerabilities and later exploit them.	Use only trusted images/registries, and verify the images using signatures. Vulnerability scanning.
Application vulnerabilities	DoS on other containers	A vulnerable application might cause DoS attack on neighboring containers (e.g., syn flood, ICMP flood) or using large amount of resources from the host, in which it affects other containers' operation.	Periodic vulnerability scanning for images and applications.
Poorly separated inter-container traffic	ARP spoofing (man-in-the-middle attack) and MAC flooding	If traffic among containers is poorly separated this might allow a compromised container to perform man-in-the-middle attack on other containers [32].	Containers should not be able to communicate unless necessary. Network should be configured as least-necessary privilege.
	DoS	A compromised container might flood the network, which renders other containers useless or highly degrade the performance.	Containers should not be able to communicate unless necessary. Chelladhurai et al. [33] proposed an algorithm to prevent DoS attacks in Docker systems via limiting container resources.
Unbounded network access from containers	Port/vulnerability scanning	If a compromised container has access to a network of other containers from various sensitivity levels it can easily perform port/vulnerability scanning and discover vulnerable containers to be targeted.	Containers should be separated into virtual networks based on sensitivity level [31]. Network monitoring for anomalies (port scanning) should be implemented too.
Insecure container runtime	Remote code execution, DoS	CVE-2014-9357, CVE-2014-6407 are some examples of remote code execution vulnerabilities in Docker. An attacker could use a vulnerability or configuration error in the runtime to target other containers (e.g., stop other containers).	Container runtime should be monitored for vulnerabilities and updated periodically.

services. As an extension to their earlier work in [29], Catuogno *et al.* [30] proposed a methodology to measure the effectiveness of containers against resource drainage attacks in which a host's resources might be drained by a suspicious service. Hence, they proposed running each service in a container, wherein the constraints used on the containers help shape their allowed resources. The authors used Docker to protect against power drainage attacks which proved to be a simple and effective technique.

4) USE CASE IV: PROTECTING CONTAINERS FROM THE HOST

Running containers on untrusted hosts should be avoided, especially with the advent of CaaS where containers can be rented from a CaaS provider. In this use case, we assume that containers are honest but the host is either semi-honest or malicious. A semi-honest host can learn about confidential container information because it controls network devices, memory, storage, and processors. A malicious host can also target the integrity of the container and its application(s). Numerous passive and active attacks can be launched from semi-honest and malicious hosts against containers in them. Examples of passive attacks include profiling in-container application activities and unauthorized access for container data. Active attacks can be more harmful since a malicious host can change the application's behavior.

IV. SOFTWARE AND HARDWARE PROTECTION MECHANISMS

Figure 4 shows the use cases based on our taxonomy and possible solutions. We used solution-based categorization since the first three use cases (discussed in Section III) depend on software-based solutions and the last one can be solved with hardware-based solutions. Software-based solutions are discussed in Section IV-A and hardware-based solutions are discussed in Section IV-B.

A. SOFTWARE-BASED PROTECTION MECHANISMS

In this section, we discuss the available software-based solutions that are used for container security (listed in Figure 4). Container technology relies heavily on software-based solutions that are either Linux Security Features (LSFs) or Linux Security Modules (LSMs). In Section IV-A.1, we discuss LSFs because most containers are based on Linux and our scope is focused on Linux containers. First, we present *namespaces* in Section IV-A.1.a and provide some

TABLE 4. Possible attack scenarios for use case (III) (protecting the host from containers) and suggested solutions.

Threat	Possible Attack	Possible Scenario	Possible Solution
Host OS attack surface	Attacks on unnecessary services	Print spooler service and other services are unnecessary on almost all servers. Hence, keeping them running will make the host inherit their vulnerabilities. Print spooler for example can be exploited (example vulnerabilities are MS16-087 and CVE-2010-2729) which enables remote code execution and privilege escalation.	Use container specific OS (e.g., CoreOS, RancherOS). If this is not possible, then follow NIST's guidelines for securing servers [34]. Stop all unnecessary services.
Containers share host OS kernel	Container escape attack	CVE-2017-5123, shocker exploit, CVE-2016-5195 (Dirty Cow) are examples of recent vulnerabilities that allow containers to escape the container runtime and target the host OS. This could allow remote code execution, which eventually leads to compromised host.	Periodic vulnerability scanning for images, applications, and container runtime. Namespaces, capabilities, and LSMs are important to control such issues.
Resources accountability	DOS	A container might consume most of the host resources causing DoS on the host and other containers in it.	Capabilities are used to control how many resources are allowed to be used by each container.
Host filesystems	Tampering	Mounting a local file system on the host allows a container with sufficient permission to cause data tampering, this is especially dangerous for sensitive directories with configuration data [31].	Containers should run with minimal set of permissions, and file changes of containers should persist to specific storage volumes for this purpose [31].

examples of their usage. Second, *CGroups* are investigated in Section IV-A.1.b. Third, *capabilities* are addressed in Section IV-A.1.c. Finally, *secure computation mode (seccomp)* is addressed in Section IV-A.1.d. In Section IV-A.2, we discuss LSMs and provide a list of most common LSMs.

1) LINUX KERNEL FEATURES

a: NAMESPACES

Namespaces perform the job of isolation and virtualization of system resources for a collection of processes. Namespaces operate as a divider of the identifier tables and other structures linked with kernel global resources into isolated instances. They partition the file systems, processes, users, hostnames, and other components. Hence, each file system namespace will have its private mount table and root directory. For every container, a unique view of the resources can be seen. The constrained view of resources for a process within a container can be extended also to a child process [35]. Namespaces are crucial building components to control what resources the container can see.

Namespaces ensure the isolation of processes that are running in a container to blind them from seeing other processes running in a different container [36]. However, one issue with namespaces is that some resources are still not namespace-aware such as devices [35]. There are numerous namespaces available, each of which is responsible for specific resource isolation. Table 5 shows a list of available namespaces and what resource it is used to isolate [37].

Namespaces example: PID namespace

The PID namespace assures that a process will only see processes that are within its own PID namespace (e.g., a container will only see its processes). Figure 5 shows an example of PID isolation for parent and child processes. In the host machine, we notice that the *init* process has PID 1. However, when the child process for a container starts, PID namespace allows it to start numbering the PIDs from 1 inside the container. This number will be mapped to a different PID in the host machine. In this example, the PID 6 in the host will be mapped to PID 1 in the machine. Similarly, PIDs 7, 8, and 9 will be mapped to 2, 3, and 4 respectively.

Inter-container Protection using Namespaces (Use case II)

As discussed earlier, namespaces are powerful Linux features to isolate resources among different containers. This helps prevent containers from accessing each other's resources, which increases their security. For instance, without namespaces, a container can easily see another container's processes and interact with them (assuming it has sufficient privileges). However, by using the PID namespaces a container will only see its own processes. Other namespaces can be applied to containers to isolate different resources.

Protecting the Host from Containers using Namespaces (Use case III)

Similar to what we discussed in the previous section, if a container can see the host's processes then this can pose a serious security risk. The same example of the PID namespace can be applied here as well. A compromised container that managed to escalate to a root privilege can disrupt the operation of the host's processes. However, with PID namespaces, isolating the container vision to its own processes helps mitigate such risks.

Solutions that use namespaces

Jian and Chen [38] studied the container escape attack for Docker. They presented a defense technique that tries to solve the escape attack based on namespace status

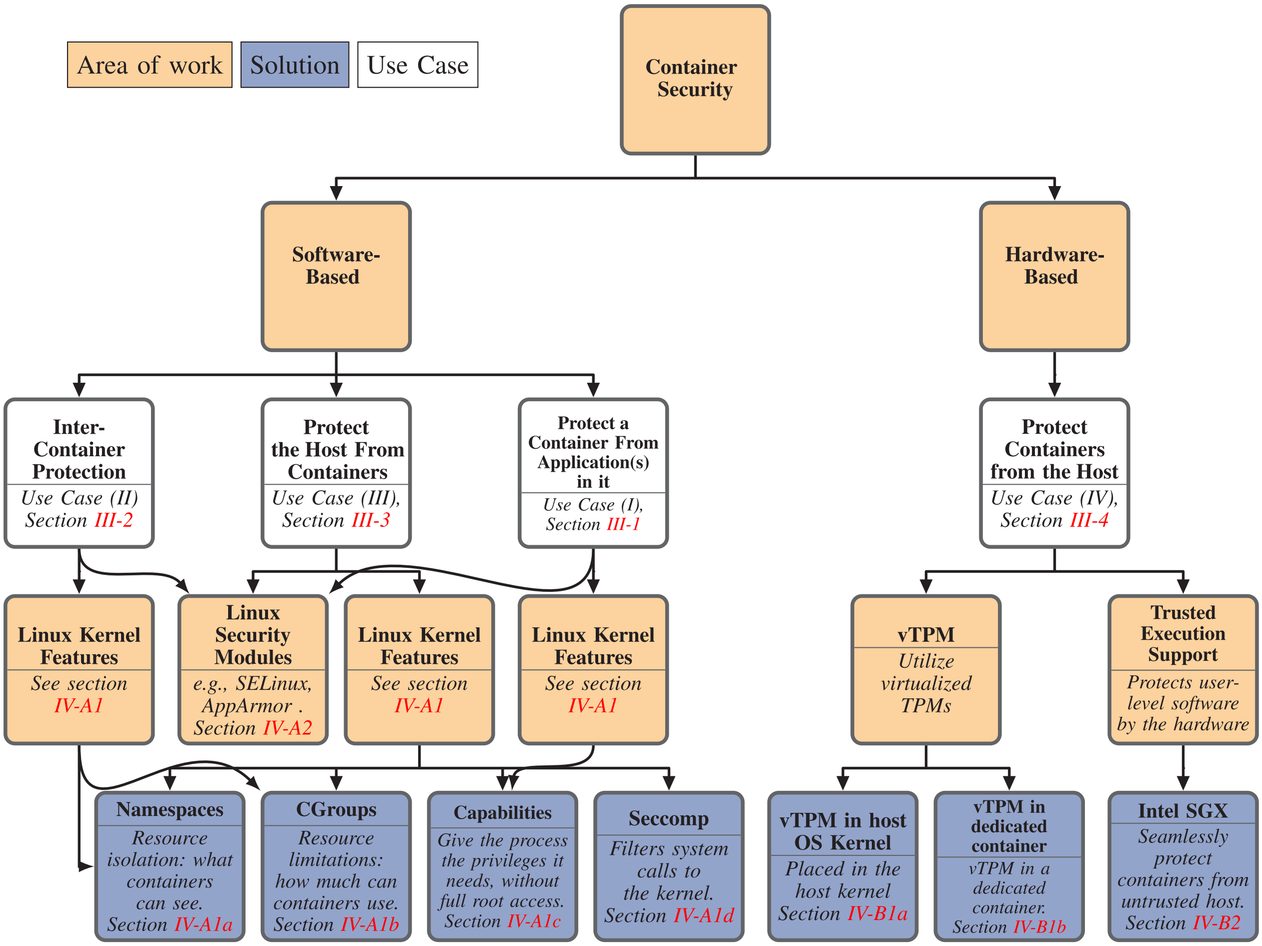
Area of work
Solution
Use Case
Container Security
Software-Based
Hardware-Based
Inter-Container Protection
Use Case (II) Section III-2
Protect the Host From Containers
Use Case (III), Section III-3
Protect a Container From Application(s) in it
Use Case (I), Section III-1
Protect Containers from the Host
Use Case (IV), Section III-4
Linux Kernel Features
See section IV-A1
Linux Security Modules
e.g., SELinux, AppArmor . Section IV-A2
Linux Kernel Features
See section IV-A1
Linux Kernel Features
See section IV-A1
vTPM
Utilize virtualized TPMs
Trusted Execution Support
Protects user-level software by the hardware
Namespaces
Resource isolation: what containers can see. Section IV-A1a
CGroups
Resource limitations: how much can containers use. Section IV-A1b
Capabilities
Give the process the privileges it needs, without full root access. Section IV-A1c
Seccomp
Filters system calls to the kernel. Section IV-A1d
vTPM in host OS Kernel
Placed in the host kernel Section IV-B1a
vTPM in dedicated container
vTPM in a dedicated container. Section IV-B1b
Intel SGX
Seamlessly protect containers from untrusted host. Section IV-B2

FIGURE 4. Protection requirements, use cases, and solutions.

TABLE 5. Linux provided namespaces.

Namespace	Constant	Isolates
IPC	CLONE_NEWIPC	System V IPC, POSIX message queues
Network	CLONE_NEWNET	Network devices, stacks, ports, etc.
Mount	CLONE_NEWNS	Mount points
PID	CLONE_NEWPID	Process IDs
User	CLONE_NEWUSER	User and group IDs
UTS	CLONE_NEWUTS	Hostname and NIS domain name

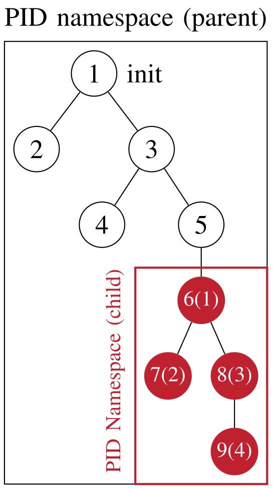

FIGURE 5. PID namespace isolation.

inspection during process execution. This should help in detecting anomalies and further prevent escape attacks. The primary motivation of their work is that if an adversary program gained root access, it will try to change the namespaces. The proposed solution detects namespace status and flags any changes that could indicate a compromised container.

Gao *et al.* [39] studied information leakage channels within containers (this study was expanded later in [40]). They address channels that could leak host information and allow adversaries to launch advanced attacks against the cloud service provider. The authors used Docker and LXC containers as a testbed and verified the discovered information leakage channels on five major cloud service providers. They showed that those channels can increase the attack's effect and reduce their cost. One of the important attacks that they analyzed is power attacks, wherein an adversary can launch power-intensive workloads when the system is already in high usage based on the leaked information, which could negatively affect the system's performance. In addition to discussing the root causes of information leakage in containers, the authors proposed a two-stage defense mechanism. They implemented a power-based namespace to provide fine-grained consumption on the container level. Their evaluation showed that this mechanism is effective at neutralizing power attacks. However, Yu *et al.* [41] show that the solution presented in [39] is not very efficient as it makes containers heavy, similar to VMs.

b: CONTROL GROUPS (CGROUPS)

CGroups are Linux features that control the accountability and limitation of resource usage such as central processing unit (CPU) runtime, system memory, input/output (I/O), and network bandwidth. In contrast to namespaces, CGroups limit how many resources can be used while namespaces control what resources a container can see (i.e., isolation). Additionally, CGroups prevent containers from using all the available resources and starving other processes. A CGroup is arranged as a slice for each resource. Then a task set can be attached to each specific CGroup. Thus, task groups are forced to use their own part of the resources [35], [42].

Inter-container protection using CGroups (Use case II)

CGroups are vital for inter-container protection requirements. As discussed, CGroups circumscribe the allowed resource usage, hence, a container cannot use more resources than what is designated for it. This helps protect other containers from numerous attacks such as Denial-of-Service (DoS) attacks. For example, a container may use so much of the host's available RAM that other containers cannot operate properly.

Protecting the host from containers using CGroups (Use case III)

Similar to inter-container protection, CGroups force resource usage limits on containers. This helps prevent the container from performing a DoS attack on the host itself. Additionally, CGroups give the host the power to not only limit resource usage but also account for how much of the resource is used. This could help in implementing usage quotas which can be a very important factor in the emerging cloud computing delivery model CaaS.

Solutions that use CGroups

Chen *et al.* [43] presented a real-time protection mechanism against DoS attacks called ContainerDrone. First, CPU DoS protection utilizes CGroups to assign a set of cores to each task. It also uses Docker features to restrict a process from raising its priority. Then, for memory DoS protection, the authors demonstrated experimentally that CGroups are not efficient because although they can restrict the allocated memory bandwidth, malicious applications could still use intensive access for that amount of memory. Hence, to protect memory from DoS attacks they used a MemGuard kernel module to prevent each CPU core from exceeding memory access.

c: CAPABILITIES

Linux systems implement the binary option of root and non-root dichotomy. In the context of containers, those binary options can be troublesome. For example, a web server (e.g., Apache) needs to bind on a specific port (e.g., TCP port 80). Without using capabilities, the web server process should have root access to perform its task. This poses a great danger because if it gets compromised, an attacker will be able to control the entire system. Capabilities turn the root and non-root dichotomy into fine-grained access control [36]. Hence, containers (usually not the daemon or container manager) will not need to have full root privilege (assuming there is an available capability for the required tasks). There are thirty-eight capabilities that cover a wide variety of tasks [44].

Capabilities are very important to protect a container from running applications inside it (Use case I). For our previous example about web servers, a container can assign the process the CAP_NET_BIND_SERVICE capability. This allows the container to run a version of that web server without requiring full root access. Assuming the web server contains a vulnerability that has been exploited by an adversary, having this capability in place will circumscribe the adversary to a single root operation (i.e., binding ports). On the other hand, if the capability was not set, the compromised or malicious application can perform full root operations on the container.

In addition, capabilities are important for protecting the host from containers (Use case III). In the previous example, we saw that a container can limit its applications but what happens if the container itself is malicious? This causes a direct risk to the host. Thus, a set of capabilities can be assigned to the container which could reduce the container's root operational threats.

d: SECURE COMPUTATION MODE (SECCOMP)

Seccomp is a Linux kernel feature that filters system calls to the kernel. Secomp is more fine-grained than capabilities [45] since different seccomp profiles can be applied to different filters. This helps to decrease the number of system calls coming from containers, which could further reduce possible threats since most attacks leverage kernel exploits through system calls.

Solutions that use Seccomp

Lei *et al.* [46] presented Split-Phase Execution of Application Containers (SPEAKER) aiming to differentiate between necessary and unnecessary system calls made by containers. SPEAKER is based on Linux seccomp. The authors observed that system calls are used in the container's short-term booting phase, hence, they can be safely removed from the long-term operation of containers. SPEAKER reduces the number of system calls made by containers through differentiating between such short- and long-term phases. They extended Linux seccomp to automatically and dynamically update available system calls for applications when they transit from the short-term booting phase to the long-term running phase. They also evaluated SPEAKER on Docker hub images for popular web servers and datastore containers and found a reduction of system calls by more than 35% with an unimportant performance overhead.

Wan *et al.* [47] presented a solution to mine sandboxes for containers based on automatic testing. During the testing phase, the proposed solution extracts the used system calls by the container. Using Seccomp, the solution creates a profile for each application based on the seen system calls during the testing phase and denies all other calls. There are two primary issues with this solution. First, it takes a relatively long time to create the profile (i.e., about 11 minutes). Second, a vulnerable or compromised container could access the needed exploitable system calls during the testing phase. However, assuming the container was safe during testing then the created profile should reduce the attack surface considerably.

e: NAMESPACES, CGROUPS, SECCOMP, AND CAPABILITIES USE IN DOCKER

Docker automatically generates a set of container namespaces and control groups when the container is started with Docker run [36]. Docker uses namespaces and CGroups to create a safe virtual environment for its containers. For example, to offer a separate network for each container, the network namespace isolates some network resources like Internet protocol (IP) addresses [48].

Docker depends on CGroups to group processes running in the container. CGroups reveal metrics about CPU and I/O block usage along with managing the resources of Docker such as CPU and memory. Docker resource configurations are either with hard or soft limits. Hard limits are used to specify a specific amount of resources to the container. Soft limits give the container the necessary resources in the machine [49].

The issue with capabilities as expressed by the Docker team in [36] is that the default set of capabilities might not provide complete security isolation. Docker default settings use the capabilities listed in Table 6 [36], [50]. Docker adds support for adding and removing capabilities. Additionally, users can set their own profile. We should note here that as the number of capabilities added to the container increases there will be a higher security risk. This is because the container will be able to perform more root privileged tasks.

The default seccomp profile of Docker blocks 44 out of 300 available system calls [45]. However, Lei *et al.* [46] claim that Docker cannot customize system calls for a specific application.

2) LINUX SECURITY MODULES (LSMs)

Morris *et al.* [51] claimed that enhanced access control mechanisms are not accepted widely to maintain OSs. This is due to the fact that there is no consensus on the right solution within the security community. LSMs allow a wide variety of security models to be implemented on Linux kernel as loadable modules [51]. This means that a user can select the preferred implementation rather than being forced to use the one that came with the OS. LSMs focus on providing the needs for implementing Mandatory Access Control (MAC) [52] with minimal changes to the Kernel itself.

TABLE 6. **Docker default capabilities.**

Capability	Description
CHOWN	Changes file owner and group.
DAC_ OVERRIDE	Discretionary Access Control (DAC), allows bypassing read, write, and execute permission checks.
FSETID	Does not clear the set-user-ID and set-group-ID bits on file modification.
FOWNER	Bypass permission checks on some operations, set ACL on files.
MKNOD	Creates a file system node.
NET_RAW	Use PACKET and RAW sockets, bind to any address.
SETGID	Arbitrarily manipulate process GIDs.
SETUID	Arbitrarily manipulate process UIDs.
SETFCAP	Setting file capabilities. Introduced in Linux 2.6.24.
SETPCAP	Has different behaviors based on CAP_SETFCAP.
NET_BIND_-SERVICE	Allows binding a socket to privileged port (i.e., < 1024).
SYS_CHROOT	Changing the root directory for a caller process.
KILL	Send any signal to any process or process group.
AUDIT_ WRITE	Enables writing records to kernel auditing log. Available since Linux 2.6.11.

LSMs date back to the 2001 Linux Kernel Summit when the U.S. National Security Agency (NSA) proposed to include Security-Enhanced Linux (SELinux) in Linux 2.5 Kernel [53]. After that, the LSM project started and many modules were developed to support various security models. In case of not specifically selecting an LSM then the default LSM will be the Linux capabilities system [54]. Currently, there are numerous LSMs available as shown in Table 7.

Typically, LSMs are used for the first three use cases as defined in Section III. Mattetti *et al.* [55] presented the LiCShield framework that aims to secure Linux containers and workloads using automatic rules constructions. LiCShield traces the image execution and generates profiles for LSMs (mainly AppArmor), which reflects on the image capabilities to increase host protection narrow container operations. The framework introduces a negligible performance overhead. A downside of LiCShield is that it does not perform vulnerability scanning for the images, which might enable dormant threats to persist. LiCShield can help in use cases I and III (protecting a container from applications in it and protecting the host from containers).

Loukidis-Andreou *et al.* [56] presented an automated system to enhance Docker container security that is based on AppArmor LSM called Docker-sec. Docker-sec includes

TABLE 7. Available Linux security modules (LSMs), with SELinux and AppArmor being the most used LSMs.

Name	Description
AppArmor	Task centered policies. The profiles are created/loaded from user space.
LoadPin	Ensures that all kernel-loaded files originate from the same file system, which allows the systems with verified unchangeable file system to override restrictions on loading modules and firmware without signing the files individually.
SELinux	Fine-grained access control features. For example, instead of having read, write, and execute permissions you can specify unlink, append only, move a file and other permissions.
Smack	Simplified Mandatory Access Control Kernel (SMACK) focuses on simplicity as a primary design goal. It provides similar functionalities to SELinux.
TOMOYO	Name based MAC. It allows processes to declare the resources and behaviors required to achieve its goals and restricts the process to the declared parameters.
YAMA	Collects the un-handled security protections that are not handled by the kernel itself.

two primary mechanisms. First, when an image is created, Docker-sec creates a static set of access rules according to the image creation parameters. Second, during runtime, the initial set is enhanced to further restrict the image's threats. According to the authors, Docker-sec can automatically protect against zero-day vulnerabilities while having minimal performance overhead. MP *et al.* [57] studied the effect of different LSMs and LSFs on Docker containers. They considered SELinux, AppArmor, and TOMOYO and provided a proof of concept that those are effective in mitigating several risks such as malicious code and bugs in namespaces code.

Bacis *et al.* [58] proposed a solution to bind SELinux policies with Docker container images to enhance their security. They claim that a major threat to containers comes from malicious (or compromised) containers that target other containers on the same host. A downside of their work is that they can only prevent essential kernel vulnerabilities. However, this can still be of help in use cases I, II, and III (protecting a container from applications in it, inter-container protection, and protecting the host from containers).

One of the primary issues with LSMs is that they are shared by all containers, wherein a single container cannot use a specific LSM [59]. To ameliorate this issue, Sun *et al.* [60] presented a novel approach to enable LSMs for containers. They presented *security namespaces* to enable containers to have autonomous control over their security and allow each container to have its own security profile. They also presented a routing mechanism to make sure that decisions made by a specific container cannot affect the host or other containers. To test their method, they developed a namespace abstraction for AppArmor LSM. Their results show that security namespaces protect against several security issues within containers with $< 0.7\%$ increased latency.

B. HARDWARE-BASED PROTECTION MECHANISMS

This section addresses solutions to protect containers from a semi-honest or malicious host as well as other containers. These solutions target use case IV, protecting containers from the host (as defined in Section III). We address two available mechanisms: The use of Virtual Trusted Platform Modules (vTPMs) and the use of Intel SGX as a trusted platform support mechanism.

1) VIRTUAL TRUSTED PLATFORM MODULES (vTPM)

One commonly used technique in trusted computing is Trusted Platform Modules (TPMs) which is hardware that is intended to be used as a cryptographic processor. TPMs provide hardware support for attestation, sealing data, secure boot, and algorithm acceleration [61], [62]. Martin [63] presented detailed information about trusted computing and TPMs. With the advent of cloud computing and virtualization, researchers started to look for alternatives to hardware TPMs in order to be more suitable for the hypervisor. This is because the hypervisor needs to make the TPM available, at the same time, to a plethora of VMs [64]. Software TPMs, also known as Virtual TPMs (vTPMs) were created to match these needs [64], [65].

vTPMs have been studied by many researchers in hardware virtualization [65]–[68]. Wan *et al.* [69] examined trusted cloud computing by using vTPMs and analyzed existing solutions that use vTPMs to better understand the security of different implementations. We should note here that vTPMs (or software TPMs) are not as secure as hardware TPMs as they suffer from numerous vulnerabilities and they cannot support protection levels as the hardware TPMs do [70], [71].

Hosseinzadeh *et al.* [64] presented the first study that implements vTPMs for containers. The two proposed implementations are discussed in the following sections (Sections IV-B.1.a and IV-B.1.b). Souppaya *et al.* [31] recommended the following pattern for attestation: start with secured/measured boot; provide a verified system platform; build a chain of trust in the rooted hardware; extend the chain of trust to the boot loader, OS kernel, system images, container runtime, and finally container images.

a: vTPM IN HOST OS KERNEL

The first implementation proposed by Hosseinzadeh *et al.* [64] was to place the vTPM in the host OS Kernel. This allows the TPM to be available to different containers. To assign a vTPM to a new container, the container manager asks the host OS kernel to create a new vTPM and then assigns it to the new container. Figure 6 shows the architecture of this type. However, the level of protection is still less than that of the full hypervisor approach [64]. Two possible scenarios are available for security assurance requirements [35]:

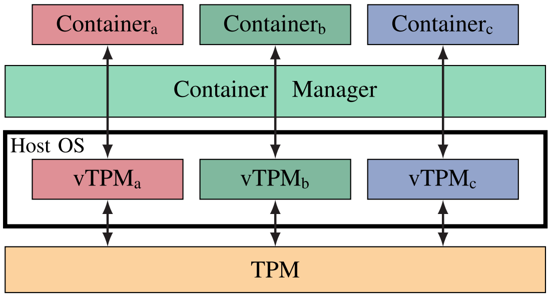

FIGURE 6. vTPM implementation in kernel.

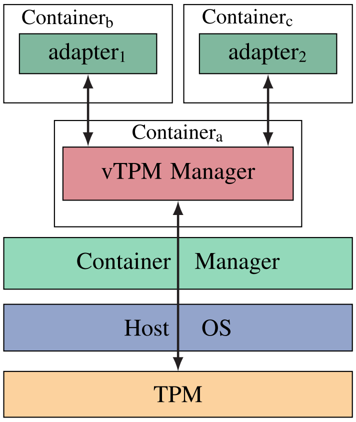

FIGURE 7. vTPM implementation in a dedicated container.

1) Fully Trusted Host OS: Trust can be established by extending the root of trust using TPM. Thus, the host OS will be considered trusted and any generated vTPM can be trusted as well. Hence, containers can attest their state by utilizing the hash extend feature implemented in the vTPM.
2) Semi-trusted Host OS: This requires placing trust in the vTPM that is generated by the host. In this case, the TPM provides an endorsement key that will be extended by giving vTPMs its own instance and deploying protocols for signing the endorsement keys of vTPMs using TPM.

b: vTPM IN A DEDICATED CONTAINER

Figure 7 shows the architecture of this implementation. The vTPM allows accessing the physical TPM differently. A single interface is used for managing the vTPM, where each container in this interface will need a software *adapter*. The container that hosts the vTPM will process incoming requests from other containers through a specific communication channel (e.g., inter-process communication). This alleviates the burden of multiple kernel modules as in the former implementation (Section IV-B.1.a).

2) INTEL SGX

In 2015, Intel created SGX which is compatible with their CPUs. Intel SGX is a set of extensions that allows Intel architecture hardware to provide confidentiality and integrity guarantees when the underlying privileged software (e.g., hypervisor, kernel) is potentially malicious [72]. Intel SGX seamlessly protects containers from underlying layers (e.g., cloud provider, or host machine). It supports secure enclaves [73] which help shield application data from other applications including higher-privileged software.

Arnautov *et al.* [74] presented Secure Linux Containers with Intel SGX (SCONE) to protect containers from attacks. The design of SCONE provides a smaller trust base, low performance overhead, and supports asynchronous system calls and user-level threading. As we discussed earlier, typically, container protections employ software mechanisms to protect containers from unauthorized access. This might protect containers from other containers and outside attacks. However, containers also need to be protected from a privileged system such as the OS kernel or hypervisor [74]. This is because adversaries usually target vulnerabilities not in the container itself but in the virtualized privileged system/ administration [74], [75].

Hunt *et al.* [76] presented Ryoan which utilizes Intel SGX to provide a distributed sandbox. It uses enclaves to protect sandbox instances from malicious computing platforms and allows it to process secret data while preventing leaking secret data. The authors evaluated Ryoan on some problems such as email filtering, health analysis, image processing, and machine translation. Hardware limitations of SGX should be taken into consideration, such as not being able to run unmodified applications. Recently, researchers started exploring solutions to allow SGX to run unmodified applications [77]. Running unmodified applications is not efficient and introduces a considerable overhead. Modifying applications, on the other hand, might not be and easy task in most cases, which will limit the overall use of SGX among developers.

Vaucher *et al.* [10] presented another solution to help developers run their containers on an untrusted cloud service provider. They proposed integrating SGX inside Kubernetes orchestrator. This is very important for CaaS providers. They tested their proposed solution on a private cluster using Google Borg traces. The authors found that the challenge is primarily in scheduling containers to SGX machines in priority. They observed that performance degrades when the memory capacity of SGX enclaves is exhausted. The authors claim that no orchestrators currently offer native support for information usage statistics about resources utilized by containers that use SGX. They extended the Linux SGX driver to gather information about the SGX runtime and direct them to the orchestrator. More studies on the practicality of SGX and/or similar technologies (e.g., ARM Trust Zone, AMD SEV, IBM SecureBlue++) are needed.

SGX is used widely to protect containers and other applications running on untrusted hosts. However, there are several attacks that target SGX that may affect container solutions using this technology. Examples of the most prominent attacks against SGX are: Controlled-channel attacks [78], stealthy page-table-based attacks [79], branch shadowing [80], BranchScope [81], last-level cache

TABLE 8. Summary of studies on Docker hub image vulnerabilities. These studies focus on image and registry risks, see Table 9 for risk types.

Ref.	Findings	Strengths	Weaknesses
[88], 2015	More than 30% of official Docker Hub repositories contain images that are highly vulnerable.	• Built open source tools to automate the process Banyan Collector and Banyan Insights, published on GitHub. • Used standardized metrics such as CVE and CVSS. • Tested 960 unique images. • Performed detailed analysis to address main packages that caused these vulnerabilities.	• CVSS vulnerability score is referred as NVD which is the National Vulnerability Database and not the score standard itself, this might create confusion for the reader. • Failed to mention mitigation to these vulnerabilities. We believe most of the them could be eliminated by image's update, which might be a standardized procedure when deploying containers. • Image scanning process is not clear. False positives for example can highly alter the results.
[89], 2017	70% of Docker Hub images have high severity issues and 54% have critical severity issues based on 1000 images scanned using an in-house tool and a commercial scanner.	• Used standardized metrics such as CVE and CVSS. • Scanned images without running them (i.e., static scanning).	• Scanning is performed using commercial scanner (Outpost). The reliability and method of this scanner was not studied. • The method for selecting the 1000 images was not clear. Are they unique images as in [88]?
[90], 2017	Studied 356,218 Docker images and found that a Docker image has on average 180 vulnerabilities, many images are outdated, and vulnerabilities usually propagate from parent to child images. Also showed that 90% of the images suffer from high severity vulnerabilities.	• Covered both community and official images. • Analyzed very large number of images. • Used open source vulnerability analyzer called CoreOS Clair. • Used standardized metrics such as CVE, CVSS. • Detailed analysis of vulnerabilities types and severity.	• The study used CoreOS Clair vulnerability scanner for Docker Hub images. Clair was mainly designed to scan CoreOS images deployed Quay.io registry. Another tool was available for Docker Hub which is Docker Security Scanner. The scope of the study was limited to Docker Hub registry, and it is not justified why to use CoreOS scanner instead of Docker Scanner at least for official images. We believe this would reduce false positives since Docker Scanner was designed for Docker Hub.

attacks [82], L1 cache attacks without hyperthreading [83], L1 cache attacks with hyperthreading [84], and translation lookaside buffer (TLB)-based attacks [85]. Intel Transaction Synchronization Extensions SGX (T-SGX) [86] provide better defense against traditional SGX [87]. However, the effectiveness of such approaches is not yet verified for container-specific applications.

V. OTHER ASPECTS OF CONTAINERS SECURITY

Previously we discussed software- and hardware-based solutions. There are two other important aspects to maintain container security: vulnerabilities management and standard guidelines. In Section V-A, we discuss studies about container vulnerabilities, exploits, and related tools. We did not use the proposed use cases for this aspect because vulnerabilities do not, in general, target a specific use case. For example, some vulnerabilities could allow inter-container attacks (e.g., meltdown), others could be used to enable hosts to attack containers (e.g., controlled-channel attacks). Then, in Section V-B, we consider the next important aspect in evaluating container security which deals with efforts towards evaluation methodologies and standardization methods for secure container deployment.

A. VULNERABILITIES, EXPLOITS, AND TOOLS

Some researchers show that a large number of container images suffer from security vulnerabilities. The number of vulnerabilities is increasing with time, which highlights an issue in remediation processes for container vulnerabilities. For example, in 2015, Gummaraju *et al.* [88] showed that 30+% of official Docker images contained high impact security vulnerabilities. Henriksson and Falk [89] scanned the top 1000 Docker images, in 2017, and showed that 70% of the images suffered from high severity issues and 54% had critical severity issues. This shows an increase in vulnerabilities for Docker images compared to the former study in 2015 [88]. Furthermore, Shu *et al.* [90] studied vulnerabilities that exist in Docker hub images and proposed the Docker Image Vulnerability Analysis (DIVA) framework to automate the process of analyzing Docker images. They scanned 356,218 images and concluded that community and official images contain 180 vulnerabilities on average, and showed that 90% of the images suffered from high severity vulnerabilities. Table 8 summarizes those studies along with their strengths and weaknesses. We believe DIVA can play an important role in discovering image vulnerabilities automatically, in which case it could help users to make sure that the downloaded image is not vulnerable before using it.

While the works above focused on discovering vulnerabilities, other researchers showed the actual effect of vulnerability exploitations. Martin *et al.* [91] presented a vulnerability analysis for the Docker ecosystem. The authors detailed some real-world scenarios for the vulnerabilities and how they could be exploited and proposed possible fixes. Lin *et al.* [92] created a dataset of 223 exploits that are effective on container platforms. Then, they evaluated the security of different Linux containers using a subset of those exploits. The authors discovered that 56.8% of these exploits are successful against the default container configurations. Similarly, Kabbe [93] presented a security analysis of Docker containers. The author focused on showing how vulnerabilities can be exploited in a production environment against exploits such as DirtyCow (CVE-2016-5195), Shellshock (CVE-2014-6271), Heartbleed (CVE-2014-0160), and Fork-bomb.

Luo *et al.* [94] identified possible covert channels that target Docker. These side channels can leak information among containers or between a container and its host, reaching a capacity larger than 700b/s. Mohallel *et al.* [95] assessed the attack surface difference between a web server installed in a container and another one installed on a base OS. They installed Apache, Nginx, and MySQL on three different containers provided on Docker hub, using a Debian server for the base OS. They assessed the security of each web server using a local network vulnerability scanner. The results concluded that Docker containers increased the attack surface due to Docker image vulnerabilities. Lu and Chen [96] highlighted the importance of penetration testing to assure container security. They first analyzed one of the top container managers (i.e., Docker) and compared it against traditional virtualization. Then penetration testing was further evaluated for specific attacks such as DoS, container escape, and side channels.

All these studies show that vulnerabilities constitute a serious threat to container security. To mitigate this threat, Barlev *et al.* [97] presented Starlight, a centralized system protection tool which intercepts and analyzes events to determine if the system administrator needs to be alerted. Starlight's downsides might include lacking the ability to discover dormant threats because it does not perform scanning nor does it have quarantine feature for compromised containers. These two issues were addressed by Bila *et al.* [98] who proposed an automated threat mitigation technique (similar to [55]) using a server-less architecture based on OpenWhisk and Kubernetes. The main advantage of this tool is that it performs image vulnerability scanning and provides quarantine for compromised containers.

B. ON STANDARDS, EVALUATION METHODOLOGIES, AND RECOMMENDATIONS

In this section, we discuss efforts towards container security standardization, evaluation methodologies, and implementation guidelines. Several researchers ([99], [100]) agree that there are no clear evaluation strategies and no systematic ways to define, study, and verify container security. Abbott's [14] findings are similar to the previous two works as the claim is made that there are no evaluation methodologies or standards for container security.

Towards this end Abbott [14] proposed a new methodology for assessing the security of container images, focusing on actions to be carried out by the user to assess the security of a container image. The author tested four popular container images: nginx, redis, google/cadvisor, mbabineau/cfn-bootstrap. The results showed security issues in each of the tested images and provided ways to resolve each issue. Reference [101] discussed, compared, and evaluated different container security features. Additionally, the author discussed the threats, attack surfaces, and hardening tips to enhance container security. Goyal [102] presented a comprehensive benchmark that provides guidelines for securing Docker (using Docker CE 17.06 or later) through a step-by-step checklist.

Efforts towards container security standardization have been realized since late 2017. In September 2017, the National Institute of Standards and Technology (NIST) presented the first special publication about container application security guidelines [31]. It discussed security threats, recommendations, and countermeasures. To implement these generalized recommendations and countermeasures one (or more) solutions are required. We summarized the NIST standard in Table 9, considered some other threats that were not covered in the standard and added studies that addressed each threat type. Efforts towards standardization have been realized by the Linux Foundation, which led to the birth of the OCI in June 2015, with its main goal to create industry standards for container format and runtime. In September 2017, the OCI published v.1.0.0 for runtime specifications of container image format. In October 2017, NIST presented another special publication ([35]) focusing on assurance requirements for container deployments and provided deployment solutions for the recommendations made in the previous special publication ([31]).

NIST did not address container security from a research point-of-view as it is more targeted towards industrial uses. Additionally, they did not address the body of research behind container security which might limit the benefits to researchers. Furthermore, they did not provide open research issues and future research directions. Our work answers the aforementioned concerns and highlights other risks that are not addressed, such as the Meltdown attack, that puts all containers at risk. We believe that standards are important to provide guidelines to overcome security issues in containers. However, as we saw in this survey, several security needs and issues are not addressed or well understood (e.g., container specific namespaces, container specific LSMs, and Meltdown). Hence, we believe earlier works on standardization require more investigation.

Table 10 shows a summary of selected studies addressed herein and the use cases that can be applied to them.

VI. OPEN ISSUES AND FUTURE RESEARCH DIRECTIONS

Research on container technologies is still in a formative stage and needs more experimental evaluation [113]. This section provides several future research directions based on our review and other researchers' recommendations.

A. MELTDOWN AND SPECTRE ATTACKS

Meltdown exploits the out-of-order execution in modern processors to extract information about the OS and other containers. Containers are based on the concept of

TABLE 9. Summary of container threats and studies addressed in each category.

	Risk	Countermeasure(s)	Studies
Image Risks	Image Vulnerabilities	Use vulnerability management tools and processes.	[55, 89, 88, 90, 93, 96, 103]
	Image Configuration Defects	Use tools and processes to comply with secure configuration best practices. Disable SSH and remote administration tools to prevent network-based attacks.	
	Embedded Malware	Monitor images for embedded malware by using monitoring processes that use malware signatures and behavioral detection.	
	Embedded Cleartext Secrets	Secrets should not be stored on images. Most orchestrators (e.g., Kubernetes) include support for secrets. Secrets should only be provided to images when needed.	
	Use of Untrusted Images	Should keep a group of trusted registries and images and not use any untrusted images.	
Registry Risks	Insecure connection to registries	Configure Development tools, container runtime, and orchestrators to connect to registries over encrypted channels.	[89, 88, 90, 93, 103, 104, 105]
	Stale images in registries	Remove any unsafe and vulnerable images from registry. Use version control on images to specifically access a specific version.	
	Insufficient Authentication	Access to registry resources should require secure authentication and authorization.	
Orchestrator Risks	Unbounded Administrative Access	Test teams only have access to test containers with limited access to containers in production environment. In addition, orchestrators must use the least privilege access model.	[106, 55, 107, 104, 105]
	Unauthorized Access	Access to administrative accounts should be controlled (e.g., multi-factor authentication).	
	Poorly Separated Inter-container Traffic	Separate network traffic into virtual network based on the sensitivity level of the network.	
	Mixing of Workload Sensitivity Levels	Isolate deployments to specific group of hosts by its sensitivity levels. Prevent placing high sensitivity workload with lower sensitivity ones.	
	Orchestrator Node Trust	Orchestration platform should provide features to create secure environment for applications that it runs. Ensure that all applications are introduced safely to the cluster.	
Container Risks	Vulnerabilities Within Runtime	Monitor container runtime for vulnerabilities because vulnerable runtime exposes all images inside it.	[55, 94, 108, 38, 93, 93, 39, 103, 109]
	Unbounded Network Access from Containers	Control outbound network traffic from containers.	
	Insecure Container Runtime Configurations	Comply with container runtime configuration standards (e.g., Center for Internet Security Docker Benchmark [102]).	
	Application Vulnerabilities	Container-aware vulnerability and instruction detection tools.	
	Rogue Containers	Strict dichotomy among development, test, and production environments. Each should come with the necessary controls for logging and access control.	
Host OS Risks	Large Attack Surface	Container-specific OS has lower threats that other OS's because it is specifically designed to host containers. Organizations that cannot use container specific OS can follow Guides to General Server Security [34].	[55, 94, 94, 108, 39, 109]
	Shared Kernel	Group container workloads on hosts based on their sensitivity levels. Do not mix containerized and non-containerized workload on the same host.	
	Host OS Component Vulnerabilities	Implement management tools to validate versioning of components provided for OS.	
	Improper User Access Rights	Authentications to the OS must be audited. Login anomalies should be monitored.	
	Host File System Tampering	Containers should run with minimal file system permissions.	
Other	Side Channel and Information Leakage Attacks	Eliminate the relationship between the leaked information and the secret data. Meltdown [26] and Spectre [27] are high risk vulnerabilities that affected Docker, LXC, and OpenVZ. Meltdown overcomes memory isolation mechanisms on any virtualization technology and OSs. It is based on the out-of-order execution in today's processors. One countermeasure is to convert to full virtualization. Spectre tries to trick other applications to access arbitrary memory location in their memory.	[26, 27, 110, 40, 94, 39, 96, 82, 111, 78, 79, 80, 81, 83, 84, 85, 86, 87]
	IAGO Attacks [112]	Narrow the interface between the trusted components and the untrusted OS.	[74]

TABLE 10. Summary of selected studies and applicable use cases.

Ref	Year	Summary	Use case (I)	Use case (II)	Use case (III)	Use case (IV)	Mechanisms
[99]	2014	Studied different container security and protection mechanisms. Showed that all containers rely on similar separation features to prevent various attacks.	✓	✓	✓	✗	LSFs
[16]	2015	Studied different protection mechanisms for Docker containers.	✓	✓	✓	✗	LSFs
[55]	2015	Proposed a framework to automate the process of rules construction for LSMs.	✓	✗	✓	✗	LSFs and LSMs
[58]	2015	Proposed a solution to enhance Docker container image security through binding SELinux LSM policies to the image.	✓	✓	✓	✗	LSMs
[103]	2015	Studied Docker container security and the available protection mechanisms to enhance their security.	✓	✓	✓	✗	LSFs and LSMs
[33]	2016	Addressed security mechanisms to protect Docker containers from DoS attacks.	✓	✓	✓	✗	LSFs and LSMs
[57]	2016	Studied different hardening techniques to enhance the security of Docker containers.	✓	✓	✓	✗	LSMs and LSFs
[64]	2016	Studied vTPMs to enhance containers security for cloud computing.	✗	✗	✗	✓	vTMPs
[74]	2016	Used Intel SGX to enhance container security on untrusted hosts.	✗	✗	✗	✓	Intel SGX
[94]	2016	Identified covert channels for Docker, and studied capabilities configuration effect on Docker container security.	✓	✓	✓	✗	LSFs and LSMs
[95]	2016	Conducted experiments to study the effect of Docker containers on the host's attack surface.	✗	✗	✓	✗	Vulnerability Scanning
[111]	2016	Presented a container management framework called MIGRATE that obfuscates the execution behavior of a container to defend against side-channel attacks from neighboring tenants.	✗	✓	✗	✗	LSFs
[15]	2016	Studied security issues and solutions for Docker containers using practical use cases.	✓	✓	✓	✗	LSFs and LSMs
[97]	2016	Proposed a tool called Starlight that analyzes a container's system calls to detect malicious operations.	✓	-	✓	✗	System calls analysis
[101]	2016	Discussed container security and protection mechanisms available.	✓	✓	✓	✗	LSFs and LSMs
[38]	2017	Presented a solution to Docker escape attacks based on namespaces status inspection.	✓	✓	✓	✗	LSFs
[46]	2017	Presented a solution called SPEAKER that is based on seccomp and analyzed system calls. Unnecessary system calls are blocked.	✓	✗	✓	✗	LSFs, system calls analysis
[47]	2017	Presented a solution to automate the creation of seccomp profiles for application based on a training period. Unseen system calls during that period will be blocked.	✓	✗	✓	✗	LSFs, system calls analysis
[102]	2017	Presented guidelines to secure Docker containers.	✓	✓	✓	✗	LSFs and LSMs
[31]	2017	NIST guidelines for securing containers. [35] provides deployment solutions for those guidelines.	✓	✓	✓	✓	LSFs, LSMs, TPMs, vulnerability scanning, and SGX
[27]	2018	Presented Spectre attacks that violate several isolation mechanisms used by containers.	✓	✓	✓	✓	Speculative execution attacks
[30]	2018	Extension for [29]. Measured the effectiveness of containers against resource drainage attacks (e.g., power drainage).	✗	✓	✓	✗	LSFs, Third party tools (e.g., Wondershaper)
[40]	2018	Extension for [39]. Studied information leakage attacks on five major cloud service providers. They proposed a solution for the power-based namespace to protect against power attacks.	✓	✓	✓	✗	LSFs, LSMs
[43]	2018	Presented a technique called ContainerDrone to protect against DoS attack using CGroups to protect CPU usage and MemGuard kernel module to protect memory access.	✓	✓	✓	✗	LSFs, MemGuard
[10]	2018	Presented a solution that incorporates SGX with Kubernetes orchestrator to make it easier for developers to deploy containers on untrusted cloud service providers.	✗	✗	✗	✓	SGX
[26]	2018	Presented Meltdown attack that can allow a malicious container to leak information about other containers on the same host. The attack worked on Docker, LXC, and OpenVZ without any restrictions.	✗	✓	✗	✗	Exploit out-of-order execution
[56]	2018	Presented a solution called Docker-sec to automate AppArmor rules creation and updates for Docker images.	✓	✓	✓	✗	LSMs
[60]	2018	One of the issues with namespaces is that they are shared among different containers on the same host. This work presents a solution to provide fine-grained namespaces to allow each container to use its own security profile.	✗	-	✓	✗	LSFs
[87]	2018	SGX is an important tool used in protecting containers from the host. This study shows that SGX suffers from numerous side-channel attacks and shows that T-SGX [86] provides better defense against such attacks.	✗	✗	✗	✓	SGX

sharing the same kernel. Recently, Lipp *et al.* [26] showed that a malicious container can leak information about other containers on the same host. This poses a serious threat to all cloud service providers that provide CaaS. Spectre [27] is another serious threat to containers, as it tricks other applications into accessing arbitrary locations in their memory. Both attacks target use case II (as defined in Section III). Those issues need to be addressed because they pose a serious threat to all containerization systems and can cause big damage to cloud providers.

B. STANDARDS FOR CONTAINER SECURITY

Further investigation for standardization of container deployment, communication protocols, and assessment methodologies is required. Similar recommendations have been proposed by many researchers [14], [100], [108]. The proposed unified models and standards should take into consideration different application requirements and platforms, usability aspects, practicality, automation, simplicity, and ease of adoption.

C. DIGITAL FORENSICS FOR CONTAINERS

Digital forensics is used to analyze security incidents. As the world is moving towards microservices that use containers, digital forensic techniques should be able to analyze security incidents related to them. According to Dewald *et al.* [114], Docker's container forensics have not yet been addressed (as of 2018). Their study can motivate further work in this area as they provided details on the new evidence introduced by using containers and how current investigation methods could be changed to commensurate containers.

Digital forensics research should answer whether the current investigation methods are sufficient for containers and what new methods are possibly required in this domain.

D. USABILITY OF VULNERABILITIES ASSESSMENT TOOLS

Several researchers showed that Docker images contain many high-risk vulnerabilities that range from 30% to 90% [88]–[90], indicating a real issue with such images. As many vulnerability assessment tools are available for Docker images, the question we are raising here is about the usability of such tools in the production environment and if their use could hinder the deployment process or not. We believe that more work is needed to study and contrast available container vulnerability scanners with respect to performance, usability, automation, and integration with current deployment and orchestration tools. This would provide tremendous help for developers and companies needing to know the risks they are facing before, during, and after deploying a specific image. We also recommend further examination of the feasibility of automatically mitigating the discovered container vulnerabilities by applying the required solutions automatically.

E. CONTAINER ALTERNATIVES

De Lucia [115] presented a review, in May 2017, on security isolation of VMs, containers, and unikernels. The study shows that VMs and unikernels provide good isolation compared to containers. However, VMs are inefficient because they are large, unikernels are not optimal because they lack privilege levels (which help separate kernel code from application code), and containers do not provide as good isolation as VMs or unikernels. The study concludes that there is no optimal solution that has been developed as of yet. It further states that an optimal solution should combine the good characteristics of VMs, containers, and unikernels.

In December 2017, clear containers merged with Kata containers which Intel claims have the same security level as VMs [28]. Further analysis is required on VMs, containers, unikernels, and hybrids. As suggested by De Lucia [115], studies on hybrid combination's feasibility, benefits, and security are necessary. This would help identify the best technology for different scenarios.

F. CONTAINER SECURITY AND PRIVACY FOR IoT APPLICATIONS

Symantec's recent report (published March 2018) shows an overall increase of 600% in attacks targeting IoT for 2017. Recently, containers are used in IoT applications because they are lighter than VMs [3]–[7]. Celesti *et al.* [5] underscored the importance of containers to enhance IoT cloud service provisioning. The authors studied performance issues and possible advantages when containers are used for IoT cloud services. Results showed that containers introduced acceptable performance overhead in real scenarios. Morabito *et al.* [6] further showed that security issues arise when using containers for IoT because of resource sharing.

Morabito *et al.* [116] analyzed containers and unikernels scalability, privacy, security, and performance for numerous IoT applications. The authors underscored some open issues for integrating containers and unikernels with IoT applications such as orchestrations and monitoring, security and privacy, standards and regulations, management frameworks and applications, portability, data storage, and telecom industry readiness and perspectives. For security and privacy, the authors highlighted the challenge of certification of applications and the need for validation mechanisms to identify tampered images. Morabito [117] encouraged the development of security mechanisms that are related to IoT applications such as [118]. Haritha and Lavanya [119] presented a survey on IoT security issues and provided a set of open issues and challenges. Hence, we re-emphasize the importance of studying container security, privacy, and standardization technologies for different IoT applications such as smart grids, smart vehicles, augmented reality, smart sensor networks, E-Health, and network function virtualization (NFV).

G. BLOCKCHAIN FOR CONTAINER VERIFICATION

As we have seen earlier, many of the security issues in containers arise from using unverified images. For example, Docker default installation does not check for image authenticity [104]. Towards this end, Notary can be used to verify Docker images' authenticity, however, is a centralized solution. A better solution is to use a decentralized verification that could use a blockchain. Xu *et al.* [104] proposed a solution for Docker images verification using blockchain called Decentralized Docker Trust (DDT). We believe further investigation is required for the applicability and effectiveness of decentralized attestation for container images.

H. CONTAINER SPECIFIC LSMs

One issue with LSMs is that they are shared by all containers, in which a single container cannot use a specific LSM [59]. As we discussed earlier in Section IV-A.2, LSMs play a pivotal role in providing security for Linux systems in general. However, sharing LSMs among different containers can be crippling as different containers might have different protection requirements. Hence, we recommend further research towards container specific LSMs to enhance and facilitate container security. Johansen and Schaufer's [120] work focuses on making LSMs available to containers and as they mention ''To date containers access to the LSM has been limited but there has been work to change the situation''. The way to achieve this according to the authors is by virtualizing the LSMs. Recently, Sun *et al.* [60] presented a study where they provided container specific LSMs (for AppArmor). Further investigation is required on this topic, especially addressing profile creation for container images and their applicability using current orchestration tools.

VII. CONCLUSIONS

Containers are important for the future of cloud computing. Microservices and containers are closely related, where containers are considered the standardized way for microservice deployment. Containers are important for the emerging field of service meshes that relies on microservices, too. However, one of the primary adoption barriers to container widespread deployment is the security issues they face. To the best of our knowledge, there are no comprehensive surveys on container security; our work attempted to fill this gap by looking at the literature and identifying the main threats which are due to image, registry, orchestration, container, side channels, and host OS risks. We thus proposed four use cases for the host-container level to elucidate how current solutions can be used to enhance container security. The use cases are: (I) protecting a container from applications inside it, (II) inter-container protection, (III) protecting the from containers, and (IV) protecting containers from a malicious or semi-honest host. Available solutions for the four use cases can be either (i) software solutions such as Linux namespaces, CGroups, capabilities, seccomp, and LSMs, or (ii) hardware solutions such as using vTPMs and utilizing trusted platform support (e.g., Intel SGX).

We further identified some open challenges and research directions for containers. The directions are focused on the importance of enhanced vulnerability management, digital investigation, container alternatives, and container-specific LSMs. We hope our work can shed light on the power and limitations of container technologies, stimulate interactions between practitioners, and spawn further research in the area.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for their helpful and constructive comments that greatly contributed to improving the final version of the paper.

REFERENCES

[1] M. Httermann, *DevOps for Developers*. New York, NY, USA: Apress, 2012.

[2] C. Tozzi. (Jan. 2017). *What Do Containers have to Do With DevOps, Anyway?*. [Online]. Available: https://containerjournal.com/2017/01/11/containers-devops-anyway/

[3] K. Kaur, T. Dhand, N. Kumar, and S. Zeadally, ''Container-as-a-service at the edge: Trade-off between energy efficiency and service availability at fog nano data centers,'' *IEEE Wireless Commun.*, vol. 24, no. 3, pp. 48–56, Jun. 2017.

[4] H. Khazaei, H. Bannazadeh, and A. Leon-Garcia, ''SAVI-IoT: A self-managing containerized IoT platform,'' in *Proc. IEEE 5th Int. Conf. Future Internet Things Cloud (FiCloud)*, Aug. 2017, pp. 227–234.

[5] A. Celesti, D. Mulfari, M. Fazio, M. Villari, and A. Puliafito, ''Exploring container virtualization in IoT clouds,'' in *Proc. IEEE Int. Conf. Smart Comput. (SMARTCOMP)*, May 2016, pp. 1–6.

[6] R. Morabito, I. Farris, A. Iera, and T. Taleb, ''Evaluating performance of containerized IoT services for clustered devices at the network edge,'' *IEEE Internet Things J.*, vol. 4, no. 4, pp. 1019–1030, Aug. 2017.

[7] R. Morabito, R. Petrolo, V. Loscrì, N. Mitton, G. Ruggeri, and A. Molinaro, ''Lightweight virtualization as enabling technology for future smart cars,'' in *Proc. IFIP/IEEE Symp. Integr. Netw. Service Manage. (IM)*, May 2017, pp. 1238–1245.

[8] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, ''Microservices: The journey so far and challenges ahead,'' *IEEE Softw.*, vol. 35, no. 3, pp. 24–35, May/Jun. 2018.

[9] J. Soldani, D. A. Tamburri, and W.-J. Van Den Heuvel, ''The pains and gains of microservices: A systematic grey literature review,'' *J. Syst. Softw.*, vol. 146, pp. 215–232, Dec. 2018.

[10] S. Vaucher, R. Pires, P. Felber, M. Pasin, V. Schiavoni, and C. Fetzer, ''SGX-Aware container orchestration for heterogeneous clusters,'' in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2018, pp. 730–741.

[11] R. Buyya *et al.* (2017). ''A manifesto for future generation cloud computing: Research directions for the next decade.'' [Online]. Available: https://arxiv.org/abs/1711.09123

[12] 451Research. (2017). *451 Research Says Application Container Market to Reach 2.7 Billion by 2020*. Accessed: Oct. 22, 2017 [Online]. Available: https://www.enterpriseai.news/2017/01/10/451-research-says-application-container-market-reach-2-7-billion-2020/

[13] D. Walsh. (2014). *Are Docker Containers Really Secure?*. Accessed: Dec. 27, 2017. [Online]. Available: https://opensource.com/business/14/7/docker-security-selinux

[14] B. M. Abbott, ''A security evaluation methodology for container images,'' M.S. thesis, Brigham Young Univ., Provo, UT, USA, 2017.

[15] T. Combe, A. Martin, and R. Di Pietro, ''To docker or not to docker: A security perspective,'' *IEEE Cloud Comput.*, vol. 3, no. 5, pp. 54–62, Sep./Oct. 2016.

[16] A. Bettini. (2015). Vulnerability exploitation in Docker container environments. FlawCheck, Black Hat Europe, Netherlands. [Online]. Available: https://www.blackhat.com/docs/eu-15/materials/eu-15-Bettini-Vulnerability-Exploitation-In-Docker-Container-Environments.pdf

[17] M. Pearce, S. Zeadally, and R. Hunt, ''Virtualization: Issues, security threats, and solutions,'' *ACM Comput. Surv.*, vol. 45, no. 2, Feb. 2013, Art. no. 17.

[18] G. Pék, L. Buttyán, and B. Bencsáth, "A survey of security issues in hardware virtualization," *ACM Comput. Surv.*, vol. 45, no. 3, Jun. 2013, Art. no. 40.

[19] F. Zhang, G. Liu, X. Fu, and R. Yahyapour, "A survey on virtual machine migration: Challenges, techniques, and open issues," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 2, pp. 1206–1243, 2nd Quart., 2018.

[20] W. Gentzsch and B. Yenier, "Novel software containers for engineering and scientific simulations in the cloud," *Int. J. Grid High Perform. Comput. (IJGHPC)*, vol. 8, no. 1, pp. 38–49, Jan. 2016.

[21] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *Proc. 21st Euromicro Int. Conf. Parallel, Distrib., Netw.-Based Process.*, Feb./Mar. 2013, pp. 233–240.

[22] M. Stuart. (2015). *Evolving Container Architectures*. Accessed: Nov. 13, 2017. [Online]. Available: https://wikibon.com/evolving-container-architectures/

[23] R. Peinl, F. Holzschuher, and F. Pfitzer, "Docker cluster management for the cloud-survey results and own solution," *J. Grid Comput.*, vol. 14, no. 2, pp. 265–282, Jun. 2016.

[24] N. Dragoni *et al.*, "Microservices: Yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*. Cham, Switzerland: Springer, 2017, pp. 195–216.

[25] B. Golden. (2016). *3 Reasons Why You Should Always Run Microservices Apps in Containers*. Accessed: Nov. 11, 2017. [Online]. Available: https://techbeacon.com/app-dev-testing/3-reasons-why-you-should-always-run-microservices-apps-containers

[26] M. Lipp *et al.*, "Meltdown: Reading Kernel memory from user space," in *Proc. 27th USENIX Secur. Symp. USENIX Secur.*, 2018, pp. 973–990.

[27] P. Kocher *et al.* (2018). "Spectre attacks: Exploiting speculative execution." [Online]. Available: https://arxiv.org/abs/1801.01203

[28] Intel. (2018). *Intel Clear Containers: Now Part of Kata Containers*. Accessed: Mar. 20, 2018. [Online]. Available: https://clearlinux.org/containers

[29] L. Catuogno, C. Galdi, and N. Pasquino, "Measuring the effectiveness of containerization to prevent power draining attacks," in *Proc. IEEE Int. Workshop Meas. Netw. (MN)*, Sep. 2017, pp. 1–6.

[30] L. Catuogno, C. Galdi, and N. Pasquino, "An effective methodology for measuring software resource usage," *IEEE Trans. Instrum. Meas.*, vol. 67, no. 10, pp. 2487–2494, Oct. 2018.

[31] M. Souppaya, J. Morello, and K. Scarfone, *Application Container Security Guide*, vol. 800. Gaithersburg, MD, USA: NIST, 2017, p. 190.

[32] P. Bogaerts. (Jan. 2017). *Arp Spoofing Docker Containers*. [Online]. Available: https://dockersec.blogspot.com/2017/01/arp-spoofing-docker-containers_26.html

[33] J. Chelladhurai, P. R. Chelliah, and S. A. Kumar, "Securing docker containers from denial of service (DOS) attacks," in *Proc. IEEE Int. Conf. Services Comput. (SCC)*, Jun./Jul. 2016, pp. 856–859.

[34] K. Scarfone, W. Jansen, and M. Tracy, *Guide to General Server Security*, vol. 800. Gaithersburg, MD, USA: NIST, 2008, p. 123.

[35] R. Chandramouli, "Security assurance requirements for linux application container deployments," Nat. Inst. Standards Technol., Gaithersburg, MD, USA, Tech. Rep. 8176, 2017.

[36] D. S. Team. (2017). *Docker Security*. Accessed: Nov. 13, 2017. [Online]. Available: https://docs.docker.com/engine/security/security/

[37] Man7.org. (2017). *Namespaces—Overview of Linux Namespaces*. Accessed: Nov. 13, 2017. [Online]. Available: http://man7.org/linux/man-pages/man7/namespaces.7.html

[38] Z. Jian and L. Chen, "A defense method against docker escape attack," in *Proc. Int. Conf. Cryptogr., Secur. Privacy*, Mar. 2017, pp. 142–146.

[39] X. Gao, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang, "ContainerLeaks: Emerging security threats of information leakages in container clouds," in *Proc. 47th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2017, pp. 237–248.

[40] X. Gao, B. Steenkamer, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang, "A study on the security implications of information leakages in container clouds," *IEEE Trans. Dependable Secure Comput.*, to be published.

[41] D. Yu, Y. Jin, Y. Zhang, and X. Zheng, "A survey on security issues in services communication of Microservices-enabled fog applications," in *Concurrency and Computation: Practice and Experience*. Hoboken, NJ, USA: Wiley, 2018, p. e4436. doi: 10.1002/cpe.4436.

[42] R. Inam, J. Slatman, M. Behnam, and M. Sjödin, and T. Nolte, "Towards implementing multi-resource server on multi-core linux platform," in *Proc. IEEE 18th Conf. Emerg. Technol. Factory Automat. (ETFA)*, Sep. 2013, pp. 1–4.

[43] J. Chen, Z. Feng, J.-Y. Wen, B. Liu, and L. Sha. (2018). "A container-based DoS attack-resilient control framework for real-time UAV systems." [Online]. Available: https://arxiv.org/abs/1812.02834

[44] Man7.org. (2017). *Linux Capabilities*. Accessed: Dec. 5, 2017. [Online]. Available: http://man7.org/linux/man-pages/man7/capabilities.7.html

[45] RedHat. (May 2018). *Chapter 8. Linux Capabilities and Seccomp*. Accessed: May 16, 2018. [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_atomic_host/7/html/container_security_guide/linux_capabilities_and_seccomp

[46] L. Lei *et al.*, "SPEAKER: Split-phase execution of application containers," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*. Cham, Switzerland: Springer, Jun. 2017, pp. 230–251.

[47] Z. Wan, D. Lo, X. Xia, L. Cai, and S. Li, "Mining sandboxes for linux containers," in *Proc. IEEE Int. Conf. Softw. Test., Verification Validation (ICST)*, Mar. 2017, pp. 92–102.

[48] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014, Art. no. 2. [Online]. Available: https://dl.acm.org/citation.cfm?id=2600241

[49] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Autonomic vertical elasticity of docker containers with ELASTICDOCKER," in *Proc. IEEE 10th Int.Conf. Cloud Comput. (CLOUD)*, Jun. 2017, pp. 472–479.

[50] Docker. (2017). *Docker Default Capabilities*. Accessed: Dec. 1, 2017. [Online]. Available: https://github.com/moby/moby/blob/master/oci/defaults.go

[51] J. Morris, S. Smalley, and G. Kroah-Hartman, "Linux security modules: General security support for the linux kernel," in *Proc. USENIX Secur. Symp.*, Aug. 2002, pp. 17–31.

[52] H. Lindqvist, "Mandatory access control," M.S. thesis, Dept. Comput. Sci., Umea Univ., Umeå, Sweden, 2006, vol. 87.

[53] S. Smalley, T. Fraser, and C. Vance, "Linux security modules: General security hooks for Linux," 2001. [Online]. Available: https://www.kernel.org/doc/htmldocs/lsm/index.html

[54] Kernel.org. (2017). *Linux Security Module Usage*. Accessed: Dec. 7, 2017. [Online]. Available: https://www.kernel.org/doc/html/v4.13/admin-guide/LSM/index.html

[55] M. Mattetti, A. Shulman-Peleg, Y. Allouche, A. Corradi, S. Dolev, and L. Foschini, "Securing the infrastructure and the workloads of linux containers," in *Proc. IEEE Conf. Commun. Netw. Secur. (CNS)*, Sep. 2015, pp. 559–567.

[56] F. Loukidis-Andreou, I. Giannakopoulos, K. Doka, and N. Koziris, "Docker-Sec: A fully automated container security enhancement mechanism," in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2018, pp. 1561–1564.

[57] A. R. MP, A. Kumar, S. J. Pai, and A. Gopal, "Enhancing security of docker using linux hardening techniques," in *Proc. 2nd Int. Conf. Appl. Theor. Comput. Commun. Technol. (iCATccT)*, Jul. 2016, pp. 94–99.

[58] E. Bacis, S. Mutti, S. Capelli, and S. Paraboschi, "Dockerpolicymodules: Mandatory access control for docker containers," in *Proc. IEEE Conf. Commun. Netw. Secur. (CNS)*, Sep. 2015, pp. 749–750.

[59] J. Johansen. (2018). *Making Linux Security Modules Available to Containers*. Accessed: Mar. 20, 2018. [Online]. Available: https://archive.fosdem.org/2018/schedule/event/containers_lsm/

[60] Y. Sun, D. Safford, M. Zohar, D. Pendarakis, Z. Gu, and T. Jaeger, "Security namespace: making linux security frameworks available to containers," in *Proc. 27th USENIX Secur. Symp. USENIX Secur.*, 2018, pp. 1423–1439.

[61] T. Morris, "Trusted platform module," in *Encyclopedia of Cryptography and Security*. Boston, MA, USA: Springer, 2011, pp. 1332–1335. doi: 10.1007/978-1-4419-5906-5_796.

[62] E. W. Felten, "Understanding trusted computing: Will its benefits outweigh its drawbacks?," *IEEE Security Privacy*, vol. 1, no. 3, pp. 60–62, May/Jun. 2003.

[63] A. Martin, "The ten-page introduction to trusted computing," Comput. Lab., Oxford Univ., 2008. [Online]. Available: https://www.cs.ox.ac.uk/files/1873/RR-08-11.PDF

[64] S. Hosseinzadeh and S. Laurén, and V. Leppänen, "Security in container-based virtualization through vTPM," in *Proc. IEEE/ACM 9th Int. Conf. Utility Cloud Comput. (UCC)*, Dec. 2016, pp. 214–219.

[65] R. Perez *et al.*, "vTPM: Virtualizing the trusted platform module," in *Proc. 15th Conf. USENIX Secur. Symp.*, 2006, pp. 305–320.

[66] P. England and J. Loeser, "Para-virtualized TPM sharing," in *Proc. 1st Int. Conf. Trusted Comput. Trust Inf. Technol., Trusted Comput.-Challenges Appl.*, Mar. 2008, pp. 119–132.

[67] B. Danev, R. J. Masti, G. O. Karame, and S. Capkun, "Enabling secure VM-vTPM migration in private clouds," in *Proc. 27th Annu. Comput. Secur. Appl. Conf.*, Dec. 2011, pp. 187–196.

[68] P. Fan, B. Zhao, Y. Shi, Z. Chen, and M. Ni, "An improved vTPM-VM live migration protocol," *Wuhan Univ. J. Natural Sci.*, vol. 20, no. 6, pp. 512–520, Dec. 2015.

[69] X. Wan, Z. Xiao, and Y. Ren, "Building trust into cloud computing using virtualization of TPM," in *Proc. 4th Int. Conf. Multimedia Inf. Netw. Secur.*, Nov. 2012, pp. 59–63.

[70] F. Stumpf and C. Eckert, "Enhancing trusted platform modules with hardware-based virtualization techniques," in *Proc. 2nd Int. Conf. Emerg. Secur. Inf., Syst.Technol.*, Aug. 2008, pp. 1–9.

[71] TrustedComputingGroup. (2016). *Trusted Platform Module (TPM) 2.0: A Brief Introduction*. Accessed: Dec. 9, 2017. [Online]. Available: https://www.trustedcomputinggroup.org/wp-content/uploads/TPM-2.0-A-Brief-Introduction.pdf

[72] V. Costan and S. Devadas, "Intel SGX explained," Cryptol. ePrint Arch., Tech. Rep. 2016/086, 2016. [Online]. Available: https://eprint.iacr.org/2016/086

[73] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using innovative instructions to create trustworthy software solutions," in *Proc. 2nd Int. Workshop Hardw. Architectural Support Secur. Privacy*, Jun. 2013, Art. no. 11.

[74] S. Arnautov *et al.*, "SCONE: Secure linux containers with intel SGX," in *Proc. OSDI*, 2016, pp. 689–703.

[75] K. Zetter, "NSA hacker chief explains how to keep him out of your system," Wired, 2016. [Online]. Available: https://www.wired.com/2016/01/nsa-hacker-chief-explains-how-to-keep-him-out-of-your-system/

[76] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," in *Proc. OSDI*, 2016, pp. 533–549.

[77] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A practical library OS for unmodified applications on SGX," in *Proc. USENIX Annu. Tech. Conf. USENIX ATC*, 2017, pp. 645–658.

[78] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 640–656.

[79] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution," in *Proc. 26th USENIX Secur. Symp. USENIX Secur.*, 2017, pp. 1041–1056.

[80] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," in *Proc. 26th USENIX Secur. Symp., USENIX Secur.*, 2017, pp. 557–574.

[81] D. Evtyushkin *et al.*, "BranchScope: A new side-channel attack on directional branch predictor," in *Proc. 23rd Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Mar. 2018, pp. 693–707.

[82] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using sgx to conceal cache attacks," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*. Cham, Switzerland: Springer, Jun. 2017, pp. 3–24.

[83] M. Hähnel, W. Cui, and M. Peinado, "High-resolution side channels for untrusted operating systems," in *Proc. USENIX Annu. Techn. Conf. (USENIX ATC)*, 2017, pp. 299–312.

[84] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. (2017). "Software grand exposure: SGX cache attacks are practical," [Online]. Available: https://arxiv.org/abs/1702.07521

[85] W. Wang *et al.*, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct./Nov. 2017, pp. 2421–2434.

[86] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: Eradicating controlled-channel attacks against enclave programs," in *Proc. 2017 Annu. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, San Diego, CA, USA, 2017, pp. 1–16.

[87] M.-W. Shih. (Jun. 2018).*TSX-Based Defenses Against SGX Side-Channel Attacks*. Accessed: Jan. 16, 2019. [Online]. Available: https://gts3.org/2018/tsgx-defense.html

[88] J. Gummaraju, T. Desikan, and Y. Turner, "Over 30% of official images in Docker Hub contain high priority security vulnerabilities," BanyanOps, 2015. [Online]. Available: https://web.archive.org/web/20150921104849/http://www.banyanops.com/pdf/BanyanOps-AnalyzingDockerHub-WhitePaper.pdf

[89] O. Henriksson and M. Falk, "Static vulnerability analysis of docker images," M.S. thesis, Blekinge Inst. Technol., Karlskrona, Sweden, 2017.

[90] R. Shu, X. Gu, and W. Enck, "A study of security vulnerabilities on docker hub," in *Proc. 7th ACM Conf. Data Appl. Secur. Privacy*, Mar. 2017, pp. 269–280.

[91] A. Martin, S. Raponi, T. Combe, and R. Di Pietro, "Docker ecosystem—Vulnerability analysis," *Comput. Commun.*, vol. 122, pp. 30–43, Jun. 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0140366417300956

[92] X. Lin, L. Lei, Y. Wang, J. Jing, K. Sun, and Q. Zhou, "A measurement study on linux container security: Attacks and countermeasures," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, Dec. 2018, pp. 418–429.

[93] J.-A. Kabbe, "Security analysis of Docker containers in a production environment," M.S. thesis, NTNU, Norway, 2017.

[94] Y. Luo, W. Luo, X. Sun, Q. Shen, A. Ruan, and Z. Wu, "Whispers between the containers: High-capacity covert channel attacks in docker," in *Proc. IEEE Trustcom/BigDataSE/ISPA*, Aug. 2016, pp. 630–637.

[95] A. A. Mohallel, J. M. Bass, and A. Dehghantaha, "Experimenting with docker: Linux container and base os attack surfaces," in *Proc. Int. Conf. Inf. Soc. (i-Society)*, Oct. 2016, pp. 17–21.

[96] T. Lu and J. Chen, "Research of penetration testing technology in docker environment," in *Proc. 5th Int. Conf. Mechatronics, Mater., Chem. Comput. Eng. (ICMMCCE)*, Sep. 2017, pp. 1354–1359.

[97] S. Barlev, Z. Basil, S. Kohanim, R. Peleg, S. Regev, and A. Shulman-Peleg, "Secure yet usable: Protecting servers and linux containers," *IBM J. Res. Develop.*, vol. 60, no. 4, pp. 1–12, Jul./Aug. 2016.

[98] N. Bila, P. Dettori, A. Kanso, Y. Watanabe, and A. Youssef, "Leveraging the serverless architecture for securing linux containers," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. Workshops (ICDCSW)*, Jun. 2017, pp. 401–404.

[99] E. Reshetova, J. Karhunen, T. Nyman, and N. Asokan, "Security of OS-level virtualization technologies," in *Proc. Nordic Conf. Secure IT Syst.* Cham, Switzerland: Springer, 2014, pp. 77–93.

[100] L. Catuogno and C. Galdi, "On the evaluation of security properties of containerized systems," in *Proc. 15th Int. Conf. Ubiquitous Comput. Commun. Int. Symp. Cyberspace Secur. (IUCC-CSS)*, Dec. 2016, pp. 69–76.

[101] A. Grattafiori, "Understanding and hardening linux containers," NCC Group, Manchester, U.K., White Paper Version 1.1, 2016. [Online]. Available: https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2016/april/ncc_group_understanding_hardening_linux_containers-1-1.pdf

[102] P. Goyal. (2017). *CIS Docker Community Edition Benchmark v1.1.0*. Accessed: Dec. 10, 2017. [Online]. Available: https://www.cisecurity.org/benchmark/docker/

[103] T. Bui. (2015). "Analysis of docker security." [Online]. Available: https://arxiv.org/abs/1501.02967

[104] Q. Xu, C. Jin, M. F. B. M. Rasid, B. Veeravalli, and K. M. M. Aung, "Blockchain-based decentralized content trust for docker images," *Multimedia Tools Appl.*, vol. 77, no. 14, pp. 18223–18248, Jun. 2018.

[105] A. Brinckman *et al.*, "A comparative evaluation of blockchain systems for application sharing using containers," in *Proc. IEEE 13th Int. Conf. e-Science (e-Science)*, Oct. 2017, pp. 490–497.

[106] D. Sæther, "Security in Docker swarm: Orchestration service for distributed software systems," M.S. thesis, Dept. Inform., Univ. Bergen, Bergen, Norway, 2018.

[107] N. Paladi, A. Michalas, and H.-V. Dang, "Towards secure cloud orchestration for multi-cloud deployments," in *Proc. 5th Workshop CrossCloud Infrastruct. Platforms*, Apr. 2018, Art. no. 4.

[108] M. A. Babar and B. Ramsey, "Understanding container isolation mechanisms for building security-sensitive private cloud," CREST, Univ. Adelaide, Adelaide, SA, Australia, Tech. Rep., 2017.

[109] Y. Li, B. Dolan-Gavitt, S. Weber, and J. Cappos, "Lock-in-Pop: Securing privileged operating system kernels by keeping on the beaten path," in *Proc. USENIX Annu. Tech. Conf. (USENIXATC)*, 2017, pp. 1–13.

[110] R. Sprabery, K. Evchenko, A. Raj, R. B. Bobba, S. Mohan, and R. Campbell, "Scheduling, isolation, and cache allocation: A side-channel defense," in *Proc. IEEE Int. Conf. Cloud Eng. (IC2E)*, Apr. 2018, pp. 34–40.

[111] M. Azab and M. Eltoweissy, "Migrate: Towards a lightweight moving-target defense against cloud side-channels," in *Proc. IEEE Secur. Privacy Workshops (SPW)*, May 2016, pp. 96–103.

[112] S. Checkoway and H. Shacham, "Iago attacks: Why the system call API is a bad untrusted rpc interface," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 253–264, Apr. 2013.

[113] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud container technologies: A state-of-the-art review," *IEEE Trans. Cloud Comput.*, to be published.

[114] A. Dewald, M. Luft, and J. Suleder. (2018). *Incident Analysis and Forensics in Docker Environments*. [Online]. Available: https://static.ernw.de/whitepaper/ERNW_Whitepaper64_IncidentForensicDocker_signed.pdf

[115] M. J. De Lucia, "A survey on security isolation of virtualization, containers, and unikernels," US Army Res. Lab. Aberdeen Proving Ground, MD, USA, Tech. Rep. AD1035194, 2017.

[116] R. Morabito, V. Cozzolino, A. Y. Ding, N. Beijar, and J. Ott, "Consolidate IoT edge computing with lightweight virtualization," *IEEE Netw.*, vol. 32, no. 1, pp. 102–111, Jan./Feb. 2018.

[117] R. Morabito, "Virtualization on internet of things edge devices with container technologies: A performance evaluation," *IEEE Access*, vol. 5, pp. 8835–8850, 2017.

[118] E. Brown. (2016). *The Future of IoT: Containers AIM to Solve Security Crisis*. Accessed: Dec. 28, 2017. [Online]. Available: https://www.linux.com/news/future-iot-containers-aim-solve-security-crisis

[119] A. Haritha and A. Lavanya, "Internet of things: Security issues," *International J. Eng. Sci. Invention*, vol. 6, no. 11, pp. 45–52, 2017.

[120] J. Johansen and C. Schaufler, "Namespacing and stacking the LSM," in *Proc. Linux Plumbers Conf.*, 2017, pp. 1–5.

SARI SULTAN received the B.Sc. degree in computer engineering from Hashemite University. He is currently pursuing the M.Sc. degree in computer engineering with Kuwait University, through an Excellence Scholarship. His research interests include security, privacy, digital forensics, and artificial intelligence. He is a Certified Ethical Hacker (CEH), a Computer Hacking Forensics Investigator (CHFI), and an ISO 27001 Information Security Management System Lead Auditor.

IMTIAZ AHMAD received the B.Sc. degree in electrical engineering from the University of Engineering and Technology at Lahore, Pakistan, the M.Sc. degree in electrical engineering from the King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia, and the Ph.D. degree in computer engineering from Syracuse University, Syracuse, NY, USA, in 1984, 1988, and 1992, respectively. Since 1992, he has been with the Department of Computer Engineering, Kuwait University, Kuwait, where he is currently a Professor. His research interests include the design automation of digital systems, high-level synthesis, distributed computing, and software-defined networks.

TASSOS DIMITRIOU is currently with the Department of Computer Engineering, Kuwait University, and the Research and Academic Computer Technology Institute, Greece. Prior to that, he was an Associate Professor with the Athens Information Technology, Greece, where he was leading the Algorithms and Security Group, and an Adjunct Professor with Carnegie Mellon University, USA, and Aalborg University, Denmark. He conducts research in areas spanning from the theoretical foundations of cryptography to the design and the implementation of leading edge efficient and secure communication protocols. Emphasis is given in authentication and privacy issues for various types of networks (adhoc, sensor nets, RFID, and smart grids), security architectures for wireless and telecommunication networks, and the development of secure applications for networking and electronic commerce. He is a Senior Member of IEEE, a Distinguished Lecturer of ACM, and a Fulbright Fellow. More information about him can be found at http://tassosdimitriou.com/