Securing the infrastructure and the workloads of linux containers

Massimiliano Mattetti*, Alexandra Shulman-Peleg†, Yair Allouche†, Antonio Corradi*, Shlomi Dolev‡, Luca Foschini*
* CIRI ICT, University of Bologna
† IBM Cyber Security Center of Excellence
‡ Ben-Gurion University

***Abstract*—One of the central building blocks of cloud platforms are linux containers which simplify the deployment and management of applications for scalability. However, they introduce new risks by allowing attacks on shared resources such as the file system, network and kernel. Existing security hardening mechanisms protect specific applications and are not designed to protect entire environments as those inside the containers. To address these, we present a LiCShield framework for securing of linux containers and their workloads via automatic construction of rules describing the expected activities of containers spawned from a given image. Specifically, given an image of interest LiCShield traces its execution and generates profiles of kernel security modules restricting the containers' capabilities. We distinguish between the operations on the linux host and the ones inside the container to provide the following protection mechanisms: (1) Increased host protection, by restricting the operations done by containers and container management daemon only to those observed in a testing environment; (2) Narrow container operations, by tightening the internal dynamic and noisy environments, without paying the high performance overhead of their on-line monitoring. Our experimental results show that this approach is efficient to prevent known attacks, while having almost no overhead on the production environment. We present our methodology and its technological insights and provide recommendations regarding its efficient deployment with intrusion detection tools to achieve both optimized performance and increased protection. The code of the LiCShield framework as well as the presented experimental results are freely available for use at https://github.com/LinuxContainerSecurity/LiCShield.git.**

1 Introduction

Shifting away from traditional on-premises computing, cloud environments allow to reduce costs via efficient utilization of servers hosting multiple customers over the same shared pools of resources. Linux containers are a disruptive technology enabling better server utilization together with simplified deployment and management of applications. Linux containers provide a lightweight operating system level virtualization via grouping resources like processes, files, and devices into isolated spaces that give you the appearance of having your own machine with near native performance and no additional virtualization overheads. When comparing between containers and VMs (in terms of CPU, memory, storage and networking resources), containers exhibited better or equal results than VM in almost all cases [24]. Furthermore, container management software, such as the Docker[1] technology [18], enable an easy packaging and deployment of applications, supporting the DevOps model of speeding up the development life-cycle through rapid change, from prototype to production [29], [34]. As a result, linux containers became widely adopted across all of the cloud layers such as Infrastructure as a service (IaaS), where they allow achieving near-native performance and Platform as a service (PaaS), linux containers are used as deployment packages allowing easy on-boarding of applications (e.g. CloudFoundry [11]).

Container threats and protection mechanisms. While optimizing the speed of deployment, linux containers were not designed as a security mechanism to isolate between untrusted and potentially malicious containers. They lack the extra layer of virtualization and thus, are less secure than VMs [2], [1]. Their vulnerabilities range from kernel exploits and attacks on the shared linux host resources to misconfigurations, side channels and data leakage [20]. Thus, container security is considered an obstacle for an even wider adoption of containerization technologies [4]. There are two main types of protection mechanisms that can be applied to container environments: security hardening mechanisms (e.g., AppArmor [16] and SELinux [8]) and host based intrusion detection systems. However, applying both mechanisms to container environments is not straightforward due to several reasons. First, there are limitations in properly deploying them in container environments where part of the workload is executed on the host and part inside the container, in which case multiple processes and applications should be grouped and protected together. Second, their practical application to the noisy container environments (see Section 5) is not straightforward.

Our approach and contributions. We present the LiCShield framework for protection of **Li**nux **C**ontainers and their workloads. Given a container image of interest, we automatically construct the security profiles protecting its execution both on the linux host and within the container. We provide a tool-set to trace and analyze containers' executions, separating the traces on the host and inside the containers. We automatically construct AppArmor rules for two different

[1]Docker and the Docker logo are trademarks or registered trademarks of Docker, Inc. in the United States and/or other countries. Docker, Inc. and other parties may also have trademark rights in other terms used herein.

978-1-4673-7876-5/15/$31.00 ©2015 IEEE

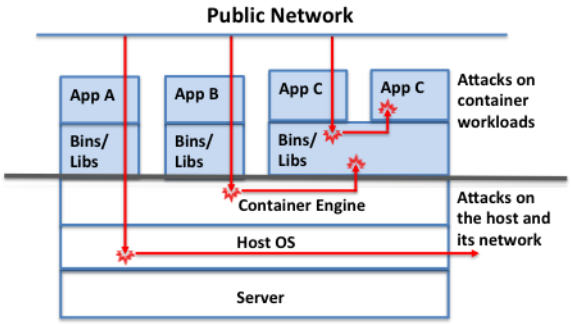

Fig. 1. An architecture of a typical linux server running containers. The arrows represent the attack vectors detailed in Table I. LiCShield profiles allow protecting the containers' workloads and the host, complementing HIDS systems.

stages in the container life-cycle: (1) container spawning on the host; (2) applications and workloads executed inside the container. We evaluate our framework on a representative data set of most popular container images to assess robustness and performance overhead. We provide recommendations regarding the optimal co-deployment of our framework with technologies like Host-based Intrusion Detection Systems (HIDS) to achieve increased protection while optimizing the performance. Finally, we describe important insights and technical contributions that can help others in tracing and protecting container environments. The code of the LiCShield framework as well its experimental results are freely available for use at https://github.com/LinuxContainerSecurity/LiCShield.git.

2 Cloud Setup and Security

Setup and assumptions. Our target are cloud deployment scenarios, in which a cloud provider offers a hosting service allowing co-locating the linux containers of different untrusted customers on the same servers. The users of this service can deploy their containers either via uploading pre-configured container images or via building them locally on the servers of the provider. Both scenarios may expose the service provider and the customers to the attacks from other malicious users.

Pre-production testing environments became a common practice for modern software projects. Their adoption and automation is facilitated by the DevOps paradigm and the raising popularity of continuous integration system such as Jenkins, which is already adopted by the industry leading OpenStack and CloudFoundry projects. Pre-production security testing is known to be a critical part of any new product launch [31]. For example, it typically includes vulnerability scanning[19], which can even be offered "as a service" over the PaaS infrastructure [17]. Assuming the existence of such environments, we show how to use them for the security hardening of linux containers and protecting container clouds that host them. Below we describe the LiCShield approach, which can be provided "as a service" as part of a PaaS to allow customers and cloud providers to protect their applications and servers.

Containers' threats. Containers simplify the application deployment and create a logical separation of workloads without attempting to make these workloads or the host more secure. Unfortunately, most of the popular container management engines are executed with admin (root) privileges and do not fully limit the privileges of the executed containers [1]. However, even when these will mature, it will still be possible to attack the container environments via the vectors detailed in Figure 1 and Table I. When considering the severity of the attack damage, we distinguish between two types of threats (see Table II): ***Threats on the host***, which put at risk the entire system by breaking out of containers and gaining administrators privileges, allowing arbitrary code execution and attack propagation to other components on the same network. These attacks can be conducted via kernel exploits (e.g. where a bug in a shared kernel is exploited for privilege escalation) or attacks on the resources shared between the containers and the host (e.g. filesystem, networking, memory and volumes). ***Threats on container workloads***, which put at risk the workloads that are executed inside the containers. The main dangers are the leakage of sensitive customer data and damage to their workloads or billing. Furthermore, legitimate containers may be exploited for attack prorogation. From the cloud provider perspective, these attacks are less dangerous since they are scoped to components that are accessible from within the container and the attacker is not supposed to gain access to administrators tools. On the other hand, due to the dynamic and intensive nature of containers' environments the detection of these attacks is not straightforward. Moreover, it is both complicated and expensive, since it may introduce high load and performance penalty (see Section 5). The goal of our work is to provide tools and deployment recommendations allowing to harden and protect the servers while minimizing the performance overhead.

Evaluation platform. While our methodology is applicable to any container management service, we have selected the Docker technology [18] as a concrete, currently most popular example [18], [34]. It is implemented as open-source project automating the distribution and deployment of applications as images installed inside linux containers (see Figure 3). Docker technology defines an *image* as a set of read-only layers of a union filesystem, which is constructed via a *build* operation on the instructions listed in a text document called *Dockerfile*. Each statement in the Dockerfile generates a new filesystem layer that is a child of the previous statement's layer. A set of the resulting layers can be distributed and then instantiated to a *container* via *run*, *create* or *start* operations. Containers can be frozen back into images, which will save all the filesystem's modifications done while it was running. So far, the only security mechanism adopted by the Docker technology are default AppArmor/SELinux profiles that aim to ensure that containers do not access some of the critical locations on the host and they suffer from two limitations: (1) they do not protect the workloads inside the containers; (2) they do not provide any protection against possible vulnerability of the Docker daemon itself or the attacks on its execution

Attacked component	Mechanisms	Compromised components	Examples
Host OS	Kernel exploits	Host and containers	A bug in the shared kernel may allow privilege escalation and arbitrary code execution on the host [14]
Host OS	Shared resources, such as filesystem, volumes, memory and networking	Host and containers	Shocker [15], is a code showing how a malicious container can scan the filesystem shared with the host till it gets to the file $/etc/shadow$ with the passwords
Container Engine	Vulnerabilities in the container engine (running as root) or the libraries loaded by it	Host and containers	CVEs at [14], Vulnerabilities in libraries executed as root (e.g. xz loaded for compression [13])
Shared Bin/Libs	Loading malicious modules	Containers	Loading a malicious shared object $/usr/lib/libnginx.so$ [27]
Applications	Cross-container leakage	Containers	One container can access the packets of another container via ARP spoofing [36]

TABLE I
EXAMPLE OF ATTACK ON THE COMPONENTS OF CONTAINER ENVIRONMENTS DEPICTED IN FIGURE 1. ADDITIONAL EXAMPLES CAN BE FOUND AT [12], [14], [20]

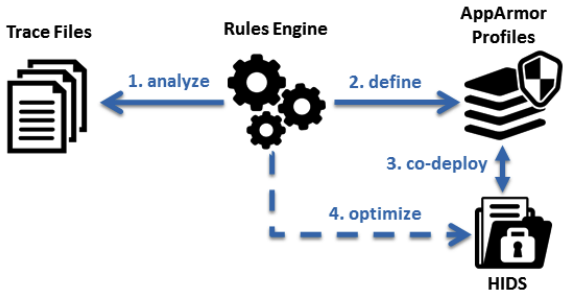

Fig. 2. **Approach Overview.**

[13]. The profiles generated by LiCShield overcome these limits by providing a fine-grained control over the containers and protection against possible vulnerabilities of the container management tools such as Docker daemon.

3 LiCShield Approach

Our main goal is to improve the security of cloud servers executing linux containers, without requiring any significant changes to the code of cloud platforms, linux distributions or the container management software, automating the workflow that can be applied without requiring any other intervention.

Figure 2 provides an overview of the LiCShield architecture consisting of the following stages:

1) *Trace and analyze:* LiCShield traces the container creation and execution in a synthetic testing environment, collecting the information about the performed operations, their resources and required permissions.
2) *Define rules:* The traces are processed to create rules that are used for two purposes: first to generate improved profiles for linux kernel security modules, such as AppArmor, restricting the containers' capabilities; second to generate rules that can be used to improve the intrusion detection systems, by automatically feeding the categories describing normal activities.
3) *Co-deploy:* We advocate that there is a need to differentiate between the protection of the host and the container workloads. For the critical host protection, we suggest to co-deploy LiCShield with HIDS, to achieved higher levels of security. At the same time, we suggest that noisy, low risk components can be protected only by LiCShield.
4) *Optimize:* LiCShield rules can be used to optimize the learning phase of intrusion detection systems, by providing the description of the expected activities. This has several benefits: first, reducing the number of false positive alerts; second, optimizing the setup and learning phase. Collecting the information on a per-image basis in pre-production with LiCShield, saves the overhead of learning the execution of each of containers spawned from the same image in the production setup.

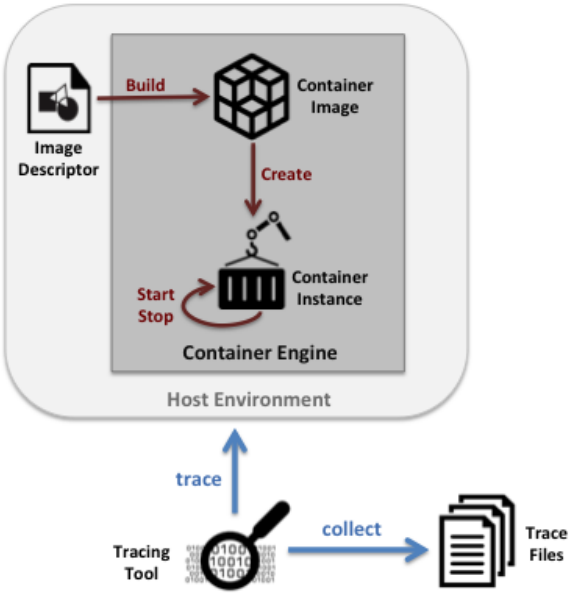

Fig. 3. **Flow Overview.**

4 LiCShield Description

Figure 3 shows the first step of the profile generation process, that we call the tracing phase. In this stage LiCShield takes a Dockerfile as input, starts the Docker daemon, sends to it commands using its REST API, and records their execution. Specifically, it first builds a new container image from the Dockerfile and then runs this image in a new container, while tracing the execution. Below we detail the main mechanisms of LiCShield which include: (1) Tracing the kernel operations;

Characteristics	Threats on the host	Threats on containers
Attack types	Container break-out due to kernel exploits, mis-configurations or vulnerabilities	Data leakage, vehicles for advanced attacks
Damage	Unlimited	Scoped to container environment
Detection challenges	Containers add new risks by exposing the hosts' resources	Performance overhead due to container workload characteristics
Proposed detection mechanisms	HIDS+LiCShield	LiCShield

TABLE II
THREATS ON CONTAINER ENVIRONMENTS

(2) Translating the collected raw data to AppArmor rules; (3) Enforcing the constructed profiles to restrict the execution on the linux host as well as the workloads inside containers.

4.1 Container tracing methodology

While the Docker daemon is performing the build and run operations, LiCShield traces their execution using a SystemTap [23] script to monitor all of the linux kernel operations. Among other available tracing tools which we tested, SystemTap represented the best trade-of in terms of reliability (ten years old project), good community support from major software companies as well as great flexibility thanks to its powerful scripting language that can be extended using native C code. In particular, this last feature allowed us to have direct access to the linux kernel structures and consequently to collect crucial information like, for instance, the control group of a process. Using SystemTap we monitor all the interactions of the Docker daemon and its children processes with the linux kernel. Every time a traced process triggers a probe point in the kernel (hereafter we refer to such process as trigger process) a new line is added to the trace file, with the following structure:

```
<probe point name> <control group path>
<executable path> <resources paths>
<mount namespace root>
```

The meaning of the fields above is:

- $probe\ \ point\ \ name$ — name of the kernel function probed;
- $control\ \ group\ \ path$ — path of the control group to which the trigger process belongs to;
- $executable\ \ path$ — path of the executable file of the trigger process;
- $resource\ \ path$ — paths of the resources given as input to the probed kernel function;
- $mount\ \ namespace\ \ root$ — path of the root directory of the trigger process in the parent mount namespace.

Each of these fields provides the specific information on what is going on in the system. Indeed, the probe point identifies which privileged operation was required. The control group indicates if the request comes from inside a container or from the host. The executable path identifies which process required the operation. Finally, the resource path together with the mount point of the container root directory identify which resource was involved in the operation. More detailed information about the structure of the trace file are provided in Section 5.

4.2 Definition of AppArmor rules

Looking at the sequence of operations necessary to $build$ a container image or to run a container it is possible to distinguish two sets of operations that have a completely different semantics and operate at two different levels of privilege. Indeed, on the one hand we have all the operations performed on the host with full root privileges by the Docker daemon, and on the other hand those performed by the container processes constrained inside a sandbox. The *rules engine* modules of LiCShield takes into account the differences between these two sets and generates two different AppArmor profiles, one specifically targeted to the operations on the host and other one to those performed inside the container.

Build command profiles. The build of a container image is predictable: starting from the same Dockerfile and from the same initial condition, the trace of operations generated by the Docker daemon does not change over the time. Even if there could be some differences due to the multi-threading environment and the possible different states of the network, the set of resources involved in the build of a container image is always the same and depends exclusively on the Dockerfile. This property makes the creation of a profile for the build of a specific Dockerfile quite simple. Another property of the build command is that it executes almost every instruction of the Dockerfile in a new container. All these short-lived containers are handled by LiCShield as a single one and a single unified profile is generated for all of them.

Run command profiles. While the $build$ of a container image is a short process, the run of a container can last an indefinite period of time. This is not a problem for the generation of the host profile, since the operations on the host are limited to the start-up phase. However, in case of the container profile this represents a big issue. Indeed, as any other application, a container can have different behaviors during its life-cycle and such behaviors should be provided in the container profile to prevent a possible policy violation in the production environment. To face this problem, we made some practical assumptions on the behavior of the applications that execute inside the containers. Since a container is a platform that hosts applications, it is reasonable to assume that no system configuration file is modified by the application during its normal execution. Following this hypothesis, the container profile grants the access permissions only to the system files accessed during the early stages of the container as well as to all the files which are under harmless subtrees of the container

filesystem.
Determining for how long should we trace the container execution is not straightforward and there is no single absolute limit. In case of the images in our dataset we observed that tracing their execution for 5 minutes is sufficient to generate profiles that do not cause any false positive.

Glob patterns issue. The number of rules of an AppArmor profile generated by LiCShield can vary considerably depending on the usage of glob patterns (i.e. inclusion of wild card characters in the path names). Indeed, without using them at all, the number of rules may be higher than ten thousands, making their maintenance quite complicated. Furthermore, there are paths and files with randomly generated names, such as the $uuid$s that uniquely identify the corresponding container instance and its root directory on the host filesystem. In these cases, the use of glob patterns is necessary to make the profiles suitable for any container instance spawned from the examined image and to prevent policy violation errors at run-time. On the other hand, an overuse of the glob patterns can grant too much freedom of action, making the use of the profile ineffective. We found the following to be the best definition of glob patterns for our two types of profiles:

- *Glob patterns for host profiles:* In the case of the host profiles most of the rules are targeted to the $/var/lib/docker$ directory, which represents the working directory of the Docker daemon. Therefore, we suggest to replace all these rules with this one */var/lib/docker/** rwkl*, which grants read, write, lock and link permission under the full subtree of the directory. Using this arrangement the number of rules in the host profile decreases to a few hundred.
- *Glob patterns for container profiles:* The container profile required much more work to find a good compromise for glob patterns usage. Tuning the framework during the test sessions we identified for which types of paths the use of the glob patterns is harmless and for which it should be avoided. For instance, the */tmp* directory is often used to store temporary files with randomly generated names, therefore a full file access is granted under this directory. Instead, we exercised caution when using the glob patterns for the files inside the */etc* directory, since it contains the most important configuration files of the system. These measures allowed us to reduce the number of rules of the container profiles to few thousands.

Profiles' technical insights. The simplicity of its rules is the most appreciated aspect of AppArmor. Indeed, it is quite easy to define rules granting write, read, lock, link and memory mapping permissions on files. Mount rules are, instead, less intuitive due to the complex nature of the mount operation that could require a long list of flags. However, while the default profile provided by the Docker technology does not check these flags, LiCShield gives a fine-grained control over the mount operations by translating every flag into the equivalent option accepted by AppArmor mount rules. Another important aspect of the profile generation that needs to be explained is how AppArmor manages the execution of a new process. Among the 4 available options to grant the execution permission to an executable path, LiCShield adopts the one that allows the new process to inherit the parent's AppArmor profile. This allows to keep the complexity of the profiles low and, at the same time, to simplify their management, if compared to options requiring a specific profile for each process.

4.3 Enforcement of the profiles

The design guidelines of LiCShield require that the AppArmor profiles are loaded before the monitored commands execution. To simulate this behavior we developed a bash script that wraps the tested API commands invocation and uses the AppArmor parser utility to load the profiles. After it is loaded, the host profile is automatically applied to the Docker daemon due to the fact that its header section contains the path to the daemon file. Although, it is possible to use the same solution to apply the container profile to the container processes, we chose a different approach that exploits the $pivot\_root$ rule. This rule provides a means to change the profile applied to a process that calls the $pivot\_root$ system call. Since this is one of last operations performed by the Docker daemon before starting the processes inside the container, it represents the perfect point to switch from the host profile to the container one.

5 Results

This section presents the results of our research addressing the evaluation of LiCShield as well as the insights and technological contributions that we gained during its development.

5.1 Data set and workloads

To evaluate our methodology we took images from the Docker Hub [3], which is used by application developers around the world for building and sharing containers. Currently, the hub contains 100 official images and more than 45,300 public images. To construct our dataset we selected 10 images, which are listed within top 20 most popular images providing platforms with workloads (i.e., skipped images like Ubuntu that provide basic OS functionality and have no workloads). The resulting dataset included images of Hadoop, Java, Node.js, PHP, Nginx, MongoDB, MySQL, PostgreSQL, Python and Redis and their workloads are detailed in the LiCShield code repository.

We used the created dataset of images to evaluate the characteristics of container environments. Table III and Figure 4 present the statistics of the system calls measured and averaged over the 10 images in our dataset. One can see that the amounts of frequent system calls can be measured in millions and is thus unrealistic to trace and analyze them on a real-time environment.

5.2 LiCShield rules and protections

LiCShield rules harden the system by blocking everything that was not observed in the simulation/testing environment. Since only legitimate operations are allowed, all the known attacks on linux containers (e.g. shocker) as well as attacks that try to exploit new vulnerability are blocked by definition.

	Container BUILD	Container RUN
Num. of processes	6680	41
Num. of threads	6895	89
Num. of syscalls	17,142,661	1,700,486

TABLE III
AVERAGE NUMBER OF PROCESSES, THREADS AND SYSTEM CALLS OF 10 IMAGES IN OUR DATASET.

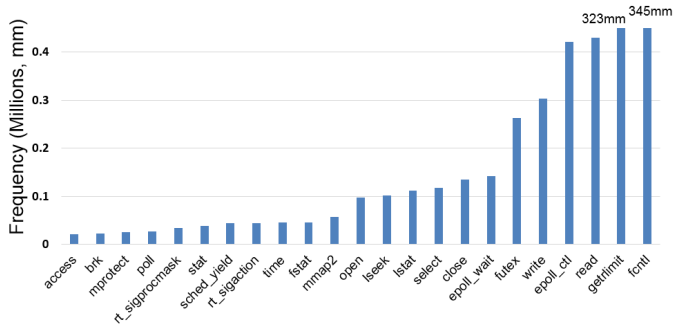

Fig. 4. **Statistics of frequent system calls measured and averaged over the 10 images in our dataset.**

The results that we show in this section were obtained testing LiCShield with the release 1.6 of the Docker technology [18]. Moreover, we modified its code to disable the use of AppArmor in order to avoid possible conflict with our profiles. Below we analyze the effectiveness of LiCShield against the kinds of attack presented in Table I. Moreover, we explain how it allows us to prevent some examples of real attacks.

Attack against the host OS. As shown in Table I there are two possible ways to conduct attacks against the host from inside the containers:

Kernel exploits: Although, LiCShield can not prevent these advanced attacks, it can block their execution and propagation. For example, the attacker will commonly open a process or a shell or will try to overwrites tools used by administrators (such as ls, ps, grep, netstat) to mask their intrusion [26]. LiCShield will block these operations, as they are not expected to be observed in normal execution. However, it is possible that an attacker will manage to get root privileges on the host and will disable the local security mechanisms (e.g. AppArmor profiles). Thus, to obtain the highest level of protection we recommend co-deploying LiCShield with HIDS, which is expected to notify remote dashboards of abnormal activities (e.g. disabling of OS hardening mechanisms). Since in combination with LiCShield, HIDS can be deployed only on the host, the resulted performance overhead is expected to be feasible. But, we will benefit from indications even when the kernel hardening or HIDS are neutralized due to an advanced attack (e.g. via kernel exploit).

Shared resources: Containers are sets of processes for the Host OS, and even if special kernel mechanisms are used to limit the set of resources that they can access, all these resources are shared with the host and, therefore, they represent a wide attack surface. For this kind of attacks LiCShield proved to be very effective as we are going to show with shocker, a popular attack known for this category. We were able to run the shocker attack again with success by granting the DAC_READ_SEARCH capability to the containers. Besides, since the $/.dockerinit$ file used by the first version of the attack is not anymore bind mounted from outside, we replaced it with a $data\ volume$ that is nothing more than a bind-mount directory from the host filesystem. We added to each image of our dataset a data volumes under the path $/shared\_volume$ and we modified the attack code to use this path instead of the $/.dockerinit$ file. Exploiting the DAC_READ_SEARCH capability shocker scans the host filesystem looking for the $/etc/shadow$ file. Once found, shocker tries to open the file, but it is stopped by AppArmor. What happens is that AppArmor resolves the file path as $/shared\_volume/etc/shadow$, and since this path has never appeared during the test phase it is not present in the profiles too. Therefore, it can not be accessed by any process inside the container and the attack can not be carried out.

Attack against the container engine. Let us stress once again the fact that with our approach we do not only constrain the behavior of linux containers, but we also protect the system against possible vulnerabilities in the container engine software. For our tests, we simulated such types of attacks by inserting backdoors in the code of the Docker daemon. For instance, we changed the Docker daemon code to mistakenly mount all the data volumes with both read and write permission. Since our AppArmor profiles strictly check the mount flags, the execution of a container that it is expected to have read-only access to a data volume, is blocked during its early stages. Furthermore, even without considering the mount flags, it would not be possible to write on the data volume since there are no rules granting write access on it.

Attack against container applications. LiCShield grants all the permissions, which the customer application needs during its execution, and blocks every operation that was not observed during the testing phase. Thus, even in the worst scenario in which an attacker succeed in exploiting a vulnerability in the application code, he will never be able to gain higher privileges than those granted to the application. Therefore, LiCShield stops the attack propagation and prevents the attacker from harming the rest of the system.

Attack against shared bin/libs. LiCShield protects the system shared libraries against attempts of tampering by granting only the memory mapping permission on them during the run of the containers. Any attempt to overwrite them during the container execution will be blocked by AppArmor, preventing the attacks described in [26].

5.3 LiCShield performance

Below, we analyze the performance overhead of all of the LiCShield stages. **Profile generation:** This step is done as part of the simulation and testing environment and, as shown in Table IV, it can take a few minutes. However, it is not so critical since it induces no overhead in the production environment.

Profile loading: We can consider this overhead negligible since the loading of the AppArmor profiles takes only few

	MySQL	PHP	PostgreSQL	Node.js	Redis	Nginx	Python	Java	MongoDB	Hadoop
Time in seconds	584	785	965	437	695	824	842	857	1226	1478

TABLE IV
PROFILE GENERATION OVERHEADS. THE TIMES ARE CALCULATED AS THE AVERAGE OF 5 EXECUTIONS OF LICSHIELD FOR EACH IMAGE IN OUR DATASET.

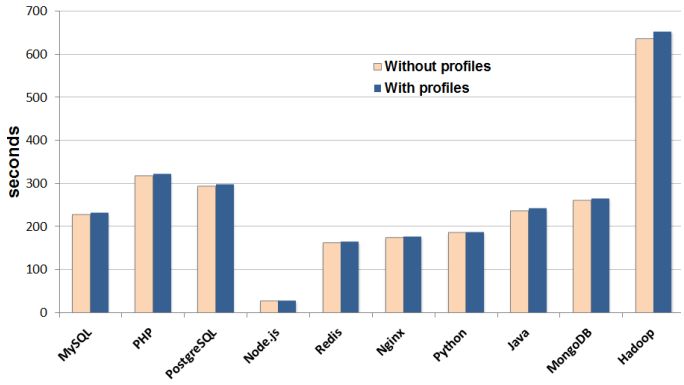

Fig. 5. **Average times of 5 executions of the build operations.**

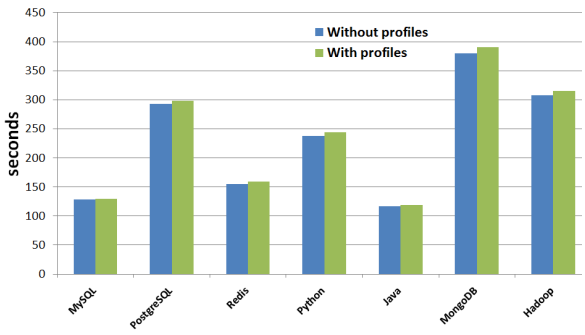

Fig. 6. **Average overhead over 5 executions of the run operations.**

seconds and it can be even reduced by the cache mechanism of AppArmor. Indeed, the first time a profile is loaded, it is compiled in a binary format that allows a more efficient evaluation of the rules at run-time. This compiled version of the profile is stored in a cache directory from which it will be retrieved every time a new profile loading will be required.

Profile enforcement: Figures 5 and 6 show the effect of AppArmor enforcement on the execution times of the build and the run operations. As this step is optimized by the kernel, its overhead is minimal. Indeed, it causes a maximum increase in the execution times of about 3%.

To test the generated AppArmor profiles in realistic environment, we selected a representative set of workloads to execute inside the containers spawned from the images of our dataset. The details of the generated workloads are provided as the supplementary materials in our public code repository https://github.com/LinuxContainerSecurity/LiCShield.git. The times in the graph of Figure 6 are calculated as the times necessary for the containers to complete the workloads assigned to them. The Nginx, Node.js and PHP images are not presented since the HTTP workloads used to stress them have fixed execution times. But, since the values of important performance parameters like the *average response time* and the *error rate* do not change, we can conclude that the enforcement of the AppArmor profiles does not cause any significant performance penalty for the execution of these three images.

5.4 Our experience - technical insights

Here, we describe the main technical challenges in kernel tracing and security hardening, providing insights that can be valuable for others researchers.

Trace kernel functions and not system calls. As presented in Table III the high number of system calls in container environments may make tracing them impracticable due to performance penalties. Furthermore, many of them are not critical or interesting for attackers, since they will not empower them with new privileges. Thus, it is important to focus only on important operations that may allow an attacker to gain new privileges of access to resources. Notably, in most cases the system calls do not provide all of the information that is required to properly describe the operation including its privileges and access resources. In fact, this information is usually retrieved by the kernel during the execution of the system calls. For example, the symbolic link resolution is a process internal to the kernel that can drastically change the effect of a system call and there is no other way to know what is the result of such process other than to look inside path resolution mechanisms of the kernel.

The security hooks are the cross road for data. After realizing that the only way to get the required information is via looking inside the kernel, we had to identify the appropriate points to capture the necessary attributes. The solution to this problem is not so obvious, given that the linux kernel has reached the considerable size of 16 million lines of code. Notably, it is possible to identify a set of points in which all the privileged operations of the kernel converge. This set consists of the hook functions of the LSMs framework that are inserted in critical points of the kernel code to manage the security fields and to perform access control. When the kernel triggers one of these functions during the execution of a system call, it has already resolved the parameters with the references to the final resources. Therefore, these functions represent the best points to capture the information that we were looking for. Furthermore, while tracing them we noticed that every operation executed inside a container triggers two security checks. The first one is done with the path of the resource internal to the container. The second one is executed with the full path from the root directory of the host. This is since the Union File System creates the abstraction of a single coherent file system by transparently overlaying files and directories of separate file systems, known as branches. Thus,

every time the kernel resolves a path internal to the Union File System, it also resolves the path relative to the branch in which the file is actually stored, triggering the second security check.

5.5 LiCShield approach evaluation

Below we discuss the value of LiCShield compared to two closely related technologies: (1) LSMs such as AppArmor and SELinux; (2) HIDS. Let us consider the current limitations of AppArmor in the context of containers. First, there is a technical problem with automatic profile construction ($aa-complain$ option) which prevents from using it with the Docker technology. Second, even after this will be fixed, this tool has no notion of containers, therefore, it is not able to distinguish between the host and the containers environments. Moreover the automatic profile generation mechanism is quite simple and it does not make any use of the glob patterns. Therefore, its profiles would be closely related to the specific container instance used to generate them, without any possibility to reuse them with other instances. Thus, for example, if we have 100 instances spawned from the same image, it will be necessary to create 100 different profiles to protect all of them (as opposed to a single profile generated by LiCShield). Furthermore, unlike our pre-production approach, the profile construction with AppArmor should be done on the specific production server executing the workload potentially influencing the performance. Similarly, let us compare LiCShield to HIDS, which usually have a learning phase in which they trace and learn the system behavior. Traditional, systems are not suited to conduct this learning phase only once and extrapolate the results for all of the container instances spawn from the same image. Thus, LiCShield can be viewed as a complementary tool, which can be used to build the filtering rules which can be used to improve the detection capabilities of HIDS systems.

6 Related Work

As more and more enterprises operate their critical business on the cloud, the number of cloud vulnerability incidents has been rapidly raising [22], [32]. Several frameworks like the CSA Cloud Controls Matrix (CCM) or the FEDRAMP authorization management program of NIST establish a minimum level of security assurance in cloud provider systems. However, the complexity and rapid evolution of the underlying technologies of cloud-based systems turn their monitoring and protection into a challenging task. Linux containers are a real break-through for DevOps, since they significantly speed up and simplify the deployment process. However, their monitoring becomes even more challenging due to several reasons: (1) the fast changing nature of the environment, where containers are constantly added or removed; (2) the large number of events; (3) the inherent insecurity of the environments. Below we review the traditional monitoring and intrusion detection methodologies and discuss their shortcomings.

6.1 Host-based intrusion detection methodologies

While firewalls and network intrusion detection systems became almost mandatory tools, they are not sufficient to protect against advanced attacks that access and modify the host servers (see [26]). Host-based Intrusion Detection Systems (HIDS) guard the executions on the servers and can be subdivided into two categories: signature-based schemes and anomaly detection. While signature-based schemes can be easily evaded by simply slightly varying the attack, anomaly detection systems monitor the interaction of the applications with the underlying operating system. HIDS typically learn the normal behavior of applications and alert upon sequences of system calls that do not conform to the expected behavior and can point out to an attack. Thus, a key objective of traditional HIDS was to develop accurate models of system call patterns of normal behavior. These include finite state automata (FSA), Hidden Markov Models (HMM), neural networks, k-nearest neighbors and Bayes models [25]. However, the applicability of host-based intrusion detection systems in the container's context proved to be rather difficult due to several factors. First, a typical cloud server is expected to run hundreds of different containers, each running high number of different processes. Taking onto account the importance of monitoring not only which systems calls are executed but also what arguments are passed [35] makes the application of system call monitoring tools highly complex and unpractical. Second, it is still possible to evade these systems as was shown in [33] and [35]. To overcome the above limitations, in this paper, we use a higher layer of abstraction, by using traces of kernel operations as our raw data. We process it to extract a set of security rules that harden the system allowing to automatically enforce mandatory access control (MAC) tools which we detail below.

6.2 Popular tools and kernel modules

The Linux Security Modules (LSMs) [37] are lightweight, access control frameworks enabling enforcement of several access control models, implemented as loadable kernel modules. Unlike system call interposition tools (e.g. Systrace [30]), LSM inserts "hooks" at every point in the kernel where a user-level system call is about to result in access to an important resource. SELinux [8] and AppArmor [16] are popular LSMs providing mandatory access control (MAC). SELinux performs access controls based on roles assigned to users and labels assigned to resources. AppArmor enables the system administrator to restrict the capabilities of a program by associating with it an appropriate security profile. Unlike SELinux, which is based on applying labels to files, AppArmor works with file paths, which simplifies the AppArmor installation and configuration. Additional LSMs like Smack [9] and TOMOYO [10] put simplicity as its primary design goal as well. Grsecurity [5] has a simple MAC implementation with address space protection and resource limiting, but requires kernel re-compilation, which makes it unsuitable for off-the shelf linux distributions. A different approach for system hardening is trimming [28], which effectively reduces the attack surface by removing, or preventing the execution of unused kernel code sections. Another approach, called BlueBox[21], uses system call introspection for defining and enforcing fine-grained process

capabilities in the kernel, which were specified as rules/policies for regulating access to system resources on a per executable basis. Additional, relevant open source modules are the Linux Integrity Measurement Architecture (IMA) [6], which allows to ensure file integrity as well as the popular OSSEC [7] software, which performs log analysis, integrity checking, time-based alerting and more. It is very useful to a achieve compliance, but requires multiple manual configurations.

7 Conclusions and Future Work

In this work we addressed the security of the basic building blocks executing applications in cloud platforms. We analyzed the threats on servers linux containers and developed a framework allowing to harden and protect these environments.

Our framework is based on the concept that cloud applications and workloads are tested in a pre-production environment. Given that popular open source projects (e.g., OpenStack) already have such testing environments (typically the Jenkins continuous integration system), we do not consider this to be a limiting assumption. The main drawback of LiCShield stems from the limitations of the mandatory access control (MAC) model of LSMs, which may lead to blocking legitimate operations that were never observed in the training environment. The rate of such false blocking operations depends on the quality of simulation and testing done during the pre-production stage (i.e. the higher the test coverage the lower the false blocking rate). We hope that the testing frameworks will continue to improve, especially due to the growing popularity of the DevOps paradigm, which is the main driving force behind the wide adoption of the linux containers. We also propose a model of co-deployment of LiCShield with HIDS, which is estimated to provide the best protection with minimal performance overhead. To conclude, we would like to emphasize that we do not view LiCShield as a technology competing with LSMs or HIDS, but a supplementary tool which can be used to optimize and complement these well established and valuable approaches. We aim to measure the exact benefits of such combined architectures in production environments as the next step of our research. We envision that our publicly available code will be used by others to protect their cloud services and applications reducing the number of potential security incidents. Lastly, as our future work, we envision extending container services with a profile management module allowing the easy download and enforcement of profiles for the image of interest. The LiCShield framework developed in this work, represents the first step in the implementation of such "Security as a Service" platform for linux containers allowing to prevent attacks and ensure the safety of container environments.

References

[1] Cloud security: Virtualization, containers, and related issues. http://www.dwheeler.com/essays/cloud-security-virtualization-containers.html.

[2] Containers and docker: How secure are they? https://blog.docker.com/2013/08/containers-docker-how-secure-are-they.

[3] Docker hub official repositories. https://registry.hub.docker.com/.

[4] For containers, security is problem #1. http://www.networkworld.com/article/2921006/network-storage/for-containers-security-is-problem-1.html.

[5] Grsecurity. https://grsecurity.net.

[6] Integrity measurement architecture (ima). http://researcher.watson.ibm.com/researcher/view_group.php?id=2851.

[7] Ossec. http://www.ossec.net/.

[8] Selinux. http://selinuxproject.org/page/Main_Page.

[9] Smack. http://en.wikipedia.org/wiki/Smack.

[10] Tomoyo. http://tomoyo.osdn.jp/.

[11] Cloud foundry foundation, 2014. http://cloudfoundry.org/cloud-foundry-foundation-launch.html.

[12] Docker : Vulnerability statistics, 2014. http://www.cvedetails.com/product/28125/Docker-Docker.html?vendor_id=13534.

[13] Docker image insecurity, 2014. https://titanous.com/posts/docker-insecurity.

[14] Linux kernel : Vulnerability statistics, 2014. http://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33.

[15] Shocker - docker breakout poc, 2014. https://github.com/gabrtv/shocker.

[16] Apparmor, 2015. http://wiki.apparmor.net/index.php/Main_Page.

[17] Application security on cloud, 2015. http://www.ibm.com/marketplace/cloud/application-security-on-cloud/us/en-us.

[18] Docker, inc., 2015. www.docker.com.

[19] A. Buecker, K. Browne, L. Foss, J. Jacobs, V. Jeremic, C. Lorenz, C. Stabler, J. Van Herzele, and I. Redbooks. *IBM Security Solutions Architecture for Network, Server and Endpoint*. IBM redbooks. IBM Redbooks, 2011.

[20] T. Bui. Analysis of docker security. *arXiv preprint arXiv:1501.02967*, 2015.

[21] S. N. Chari and P.-C. Cheng. Bluebox: A policy-driven, host-based intrusion detection system. *ACM Transactions on Information and System Security (TISSEC)*, 6(2):173–200, 2003.

[22] CSA. Cloud computing vulnerability incidents: A statistical overview, 2013.

[23] F. C. Eigler and R. Hat. Problem solving with systemtap. In *Proc. of the Ottawa Linux Symposium*, pages 261–268. Citeseer, 2006.

[24] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. *technology*, 28:32, 2014.

[25] S. Forrest, S. Hofmeyr, and A. Somayaji. The evolution of system-call monitoring. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pages 418–430. IEEE, 2008.

[26] M. Goncharov. Russian underground 101. *Trend Micro incorporated research paper*, 2012.

[27] A. Kovalev, K. Otrashkevich, E. Sidorov, and A. Rassokhin. Effusion–a new sophisticated injector for nginx web servers. 2014.

[28] A. Kurmus, A. Sorniotti, and R. Kapitza. Attack surface reduction for commodity os kernels: trimmed garden plants may attract less bugs. In *Proceedings of the Fourth European Workshop on System Security*, page 6. ACM, 2011.

[29] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 239(2), 2014.

[30] N. Provos. Improving host security with system call policies. In *USENIX Security*, volume 3, 2003.

[31] M. Sacks. Production launches. In *Pro Website Development and Operations*, pages 73–92. Springer, 2012.

[32] I. S. Systems. IBM x-force threat intelligence quarterly, 1q 2015, 2015. http://public.dhe.ibm.com/common/ssi/ecm/wg/en/wgl03073usen/WGL03073USEN.PDF.

[33] K. M. Tan, K. S. Killourhy, and R. A. Maxion. Undermining an anomaly-based intrusion detection system using common exploits. In *Recent Advances in Intrusion Detection*, pages 54–73. Springer, 2002.

[34] J. Turnbull. *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014.

[35] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 255–264. ACM, 2002.

[36] S. Whalen. An introduction to arp spoofing. *Node99 [Online Document], April*, 2001.

[37] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Foundations of Intrusion Tolerant Systems*, pages 213–213. IEEE Computer Society, 2003.