Security Namespace : Making Linux Security Frameworks Available to Containers

Yuqiong Sun
Symantec Research Labs

David Safford
GE Global Research

Mimi Zohar
IBM Research

Dimitrios Pendarakis
IBM Research

Zhongshu Gu
IBM Research

Trent Jaeger
Pennsylvania State University

Abstract

Lightweight virtualization (i.e., containers) offers a virtual host environment for applications without the need for a separate kernel, enabling better resource utilization and improved efficiency. However, the shared kernel also prevents containers from taking advantage of security features that are available to traditional VMs and hosts. Containers cannot apply local policies to govern integrity measurement, code execution, mandatory access control, etc. to prevent application-specific security problems. Changes have been proposed to make kernel security mechanisms available to containers, but such changes are often adhoc and expose the challenges of trusting containers to make security decisions without compromising host system or other containers. In this paper, we propose *security namespaces*, a kernel abstraction that enables containers to have an autonomous control over their security. The security namespace relaxes the global and mandatory assumption of kernel security frameworks, thus enabling containers to independently define security policies and apply them to a limited scope of processes. To preserve security, we propose a routing mechanism that can dynamically dispatch an operation to a set of containers whose security might be affected by the operation, therefore ensuring the security decision made by one container cannot compromise the host or other containers. We demonstrate security namespace by developing namespaces for integrity measurement and mandatory access control in the Linux kernel for use by Docker containers. Results show that security namespaces can effectively mitigate security problems within containers (e.g., malicious code execution) with less than 0.7% additional latency to system call and almost identical application throughput. As a result, *security namespaces* enable containers to obtain autonomous control over their security without compromising the security of other containers or the host system.

1 Introduction

Lightweight virtualization (i.e., containers) offers a virtual host environment for applications without the need for a separate kernel, enabling better resource utilization and improved efficiency. It is broadly used in computation scenarios where a dense deployment and fast spin-up speed is required, such as microservice architecture [39] and serverless computation (e.g., Amazon Lambda [26]). Many commercial cloud vendors [23, 20, 1] have adopted the technology.

The key difference between containers and traditional VMs is that containers share the same kernel. While this enables better resource utilization, it also prevents containers from taking advantage of security features in kernel that are available to traditional VMs or hosts. Containers cannot apply local security policies to govern integrity measurement, code execution, mandatory access control, etc. to prevent application specific security problems. Instead, they have to rely on a global policy specified by the host system admin, who often has different security interests (i.e., protect the host system) and does not have enough insight about the security needs of individual containers. As a result, containers often run without any protection [34, 40].

Previous efforts of making kernel security frameworks available to containers are often adhoc and expose the challenges of trusting containers to make security decisions without compromising host system or other containers. For example, a kernel patch [24] to Integrity Measurement Architecture (IMA) [53] suggested that the IMA measurement list can be extended with a container ID, such that during integrity attestation the measurements will become separable based on containers. As another example, AppArmor and Tomoyo introduced the concept of profile and policy namespace [49, 44] to allow certain processes to run under a policy different from the rest of the system. These changes, however, only made limited kernel security features available to containers,

and they all rely on the system owner to specify a global policy, leaving containers no real freedom in enforcing an autonomous security.

In this paper, we explore approaches to make kernel security frameworks available to containers. Due to the diversity of kernel security frameworks and their different design perspectives and details, it is extremely difficult to reach a generic design that can cover all kernel security frameworks in a single step. Instead, this paper explores an initial step, by making two concrete kernel security frameworks available to containers, to investigate the common challenges and approaches behind. Hopefully, the results have enough generality to guide other kernel security frameworks and eventually lead to a generic design. In studying the two popular kernel security frameworks, namely IMA [53] for integrity and AppArmor [41] for mandatory access control, we make the following observations: first, we find that the common challenge for containers to obtain autonomous security control is the implicit *global* and *mandatory* assumptions that kernel security frameworks often make. Kernel security frameworks are designed to be global—they control *all* processes running on the system. They are also designed to be mandatory—only the owner of the system may apply a security policy. However, autonomous security control requires relaxation of both assumptions. A container need to apply local security policies to control a subset of processes running on the system (i.e., processes in the container). Relaxing these assumptions involves security risks. Our second insight is that we can relax the global and mandatory assumptions *in a secure way* by checking if the autonomous security control of a container may compromise the security of other containers or the host system. We do this by inferring from containers' security expectation towards an operation.

Leveraging these insights, we propose the design of security namespaces, kernel abstractions that enable containers to utilize kernel security frameworks to apply autonomous security control. Security namespace virtualizes kernel security frameworks into virtual instances, one per container. Each virtual instance applies independent security policies to control containerized processes and maintains their independent security states. To ensure that the relaxation does not compromise any principal's security (i.e., other containers or the host system), an Operation Router is inserted before the virtual instances mediating an operation. The Operation Router decides the set of virtual instances whose security might be affected by an operation and routes the operation to those virtual instance for mediation. After each virtual instance makes an independent security decision, the decisions are intersected. A specific challenge is that virtual instances may make conflicting security decisions. A Policy Engine is added to detect such conflicts and inform the container owners of potential conflicts before they load their security policies.

We evaluate our design by developing two concrete instances of security namespace, one for IMA and one for AppArmor. Results show that leveraging the namespace abstractions, containers (e.g., Docker and LXC) can exercise the full functionality of IMA and AppArmor and apply autonomous security control, much like a VM or host system. Specifically, we show that the IMA namespace enables containers to independently measure and appraise files that are loaded into the container, without violating any of the host system's integrity policy. For AppArmor namespace, we show that it enables containers to enforce two policy profiles simultaneously, one protects the host system and another protects the containerized application, which was not possible as discussed in Ubuntu LXC documentation [34]. We evaluate the performance of both namespace abstractions. Results show that security namespaces introduce less than 0.7% latency overhead to system calls in a typical container cloud use case (i.e., no nested namespaces) and an almost identical throughput for containerized applications.

In summary, we make the following contributions.

- Through studying IMA and AppArmor, we investigate the common challenges and approaches behind making kernel security frameworks available to containers.
- We develop two concrete security namespace abstractions, one for IMA and another for AppArmor, which enables autonomous security control for containers while preserving security.
- We show that widely used container systems (e.g., Docker and LXC) can easily adopt the IMA and AppArmor security namespace abstractions to exercise full functionality of kernel security frameworks with modest overhead.

2 Background

In this section, we first describe the namespace concept in the Linux kernel and how it is adopted by container. We then discuss security frameworks in Linux kernel.

2.1 Namespace and Container

The Linux namespace abstraction provides isolation for various system resources. According to Linux man page [31]:

> *A namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource. Changes to the global resource are visible to other processes that are members of*

Table 1: Namespaces in Linux kernel.

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

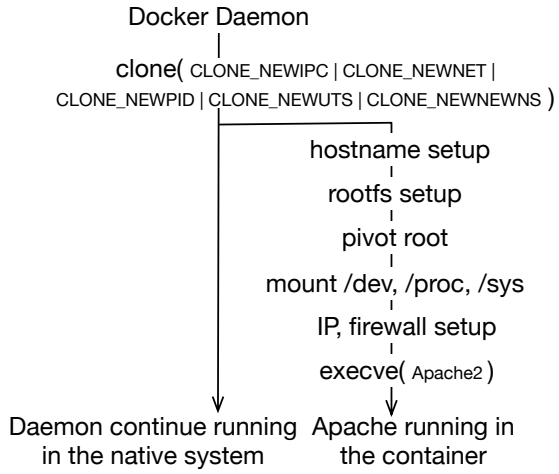

Figure 1: Creating a Docker container.

the namespace, but are invisible to other processes.

We use mount namespace as an example. Without mount namespace enabled, processes running within a Linux OS share the same filesystems. Any change to the filesystems made by one process is visible to the others. To provide filesystem isolation across processes, chroot [6] was first introduced but then found to be vulnerable to a number of attacks [7, 8]. As a more principled approach, Linux kernel introduced the mount namespace abstraction to isolate mount points that can be seen by the processes. A mount namespace restricts the filesystem view to a process by creating separate copies of vfs mount points. Thus, processes running in different mount namespaces could only operate over their own mount points. To date, six namespace abstractions (Table 1) have been introduced into the Linux kernel.

Container [56] is an OS-level virtualization technology. By leveraging the namespace abstractions (together with other kernel mechanisms, e.g., Cgroups, SecComp), a container can create an isolated runtime environment for a set of processes. Well-known container implementations include Docker [13], LXC [33], and LXD [35]. Figure 1 illustrates the procedure of creating a Docker container. It starts from launching a daemon process (e.g., dockerd) on the native host system. The daemon process forks itself (i.e., via *clone*), specifying that the newly forked process will run in different namespaces from the native for isolation. The forked process then properly sets up the namespaces that it runs in (e.g., mounting a different root, setting up its IP address, firewalls, etc.) and executes a target program (i.e., via *execve*). The target program then starts running in an environment isolated from other containers and the native system. The isolation is achieved by using the namespace abstractions. When forking a new process, the *clone* system call accepts different flags to indicate that the child process should run in none, one or several types of new namespaces. Containers often leverage all six types of namespaces at the same time, in order to create a fully isolated environment.

2.2 Kernel Security Frameworks

To protect the system and applications running atop, Linux kernel features many security frameworks. Some of these frameworks are upstreamed to the Linux kernel, such as Linux integrity subsystem [53, 30], SELinux [42], and AppArmor [41]. Some remain as research proposals [43, 63, 2, 28]. Although differing in security goals, these frameworks share a similar design. In general, these security frameworks rely on ”hooks” added into the kernel to intercept security critical operations (e.g., accessing inodes) from a process. Such security critical operations are passed to a security module where decisions (i.e., allow or deny) are made based on security policies.

2.2.1 Linux Integrity Subsystem

The Linux integrity subsystem, also known as the Integrity Measurement Architecture (IMA) [53], is designed to thwart attacks against the unexpected changes to files, particularly executable, on a Linux system. IMA achieves this by measuring files that may affect the integrity of the system. Working with a secure coprocessor such as TPM, IMA could securely store the measurements and then report them to a remote party as a trustworthy proof of the overall integrity status of the system (i.e., attestation). For example, a bank server could leverage IMA to attest its integrity to its users, enabling the users to bootstrap trust before operating over their accounts. In addition to attestation, IMA can also enforce the integrity of a system by specifying which files could be loaded. IMA does so by appraising files against ”good” values (e.g., checksums or signatures) specified by system owners. In the above example, a bank would benefit from IMA to maintain a tightly controlled environment of its servers and enforce that only approved code could be run.

3 Motivation

In this section, we discuss the need for containers to have autonomous security control, and the fundamental challenges of achieving it.

3.1 Autonomous Security Control

As more critical applications are deployed in containers, container owners want to utilize kernel security frameworks to govern integrity measurement, code execution, mandatory access control, etc. to prevent application specific security problems. Ideally, such security control should be *autonomous*, similar to when their applications were deployed on VMs or hosts.

Unfortunately, it is difficult to achieve the autonomy by directly using existing kernel security frameworks. As an example, consider a containerized bank service deployed on a public cloud. The service owner wants to control the integrity of the service by ensuring that critical service components such as service code, libraries and configurations are not modified. However, she cannot use IMA to do so. First, the bank service could not attest its integrity using IMA. The reason is that IMA, as an in-kernel security mechanism, tracks the integrity of the entire system. Consequently, measurements from different containers (and the host system) are mixed together and cannot be accessed independently. Second, the bank service cannot control what code or data can be loaded into the container. Since IMA only allows a single policy maker (in this case, the cloud vendor that controls the host system), individual containers cannot decide what files to measure nor what would be good measurements for those files.

We argue that achieving the autonomous security control is fundamentally difficult because *security frameworks in Linux kernel are designed to be global and mandatory*. Security frameworks are global in a sense that they control *all* processes running on a kernel. In addition, security states (e.g., IMA measurements) are stored centrally for the global system. Security frameworks are mandatory in a sense that only the owner of the system (i.e., system admin) is authorized to specify a policy. Other principals on the system (i.e., container owners) are not allowed to make security decisions.

Enabling containers to have autonomous security control, however, requires relaxation of both the global and mandatory assumption of security frameworks. Security frameworks need to exercise their control over a *limited* scope of processes specified by the container owner and security states need to be maintained and accessed separately; this relaxes the global assumption of security frameworks. Container owners will independently apply security policies and together participate in the process of security decision making; this relaxes the mandatory assumption of security frameworks.

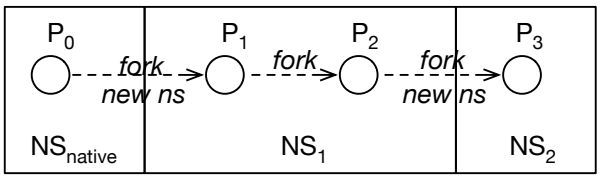

Figure 2: A strawman design of security namespace.

3.2 Security Namespace

To achieve the autonomous security control, one idea is to design a *security namespace abstraction*, similar to how other global resources are isolated/virtualized in Linux. However, unlike other resource namespaces, security namespace needs to relax the global and mandatory assumption which the security of the system often rests upon. Thus, if naively designed, it could introduce security loopholes into the system, invalidating the security offered by security frameworks. In this section, we first introduce a strawman design of security namespace that mimics the design of resource namespaces, and present two attack examples.

Strawman design. Analogous to other resource namespaces, a security namespace has to make it appear to the processes within the namespace that they have their own isolated instances of kernel security framework. An intuitive design is thus to virtualize kernel security frameworks (i.e., by replicating code and data structures) into virtual instances. Each virtual instance becomes a security namespace: it is associated with a group of processes and it makes security decisions over those processes independently. For example, as shown in Figure 2, process P_0 runs in native security namespace NS_{native}. It creates a new security namespace NS_1 and forks itself (i.e., via `clone` with `CLONE_NEW` flag set). The child process P_1 now runs in NS_1. P_1 further forks itself in the same security namespace and P_2 further forks P_3 in a new security namespace. In this case, the strawman design assigns security control of P_0 to NS_{native}, control of P_1 and P_2 to NS_1, and control of P_2 to NS_2. The owner of NS_{native}, NS_1 and NS_2 will independently apply security policies.

While such design achieves autonomous security control in a straightforward way, it introduces two attacks:

Attack Example 1. Consider an example where the security namespaces NS_{native} and NS_1 under discussion are IMA namespaces. Assume the owner of the native system wants to prove the integrity of the native system by using NS_{native} to measure and record all the code that has been executed on the system (Figure 3a). Such measurements serve as an evidence for remote parties to bootstrap trust into the native system. However, a malicious subject P may fork itself into a new IMA namespace NS_1 and then execute a malware inside of it (Figure 3b). In

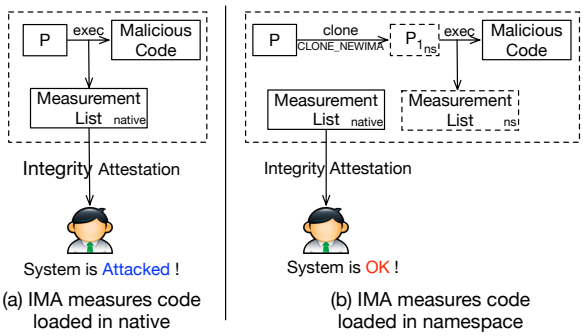

Figure 3: An attack in the strawman design. A remote verifier may be tricked into believing the system is of sufficient integrity to use even though a malware was once loaded on the system.

this case, the measurements of the malware are stored onto the measurement list of NS_1, which will be deleted after the namespace exits, leaving no traces behind. Integrity attestation of the native system, in this case, will cause a remote party to believe that the system is of sufficient integrity to use, despite the fact that the malware was once executed on the system.

In this example, P managed to execute a malware without leaving a footprint on the system, due to that the native security namespace NS_{native} no longer controls P_1, and the security namespace NS_1 that controls P_1 is created and controlled by adversary. This example demonstrates that, in a security namespace design, if the global assumption of a security framework is relaxed in a naive way, adversary may leverage that fact to circumvent system policy.

Attack Example 2. A container associated with security namespace NS_1 shares a file f with another container associated with a different security namespace NS_2. The file is of high integrity to NS_1, and thus is shared in a read-only way. However, since NS_2 has security control over processes running in the second container, it can make f read-write to its processes. As a result, when processes from NS_1 reads f, they read in low integrity input even though they expect the file to be maintained at high integrity. In this example, NS_2 managed to let processes in NS_1 take low integrity input by specifying a policy different from what was expected by NS_1. Worse, since processes in NS_1 mistakenly believe that the file is still at high integrity, most likely they will not take countermeasures that could otherwise protect themselves (e.g., by checking file hash before reading it). Previous researches [22, 60] also show that, when two or more principals try to make security decisions independently, the inconsistencies between them may open additional attack channels. This example demonstrates that, in a security namespace design, if mandatory assumption of security framework is relaxed in a naive way (e.g., by allowing two or more principals to apply security policies freely), adversary may leverage that fact to launch attacks.

3.3 Goals

The high level goal of this paper is to investigate the design of security namespace that enables containers to have autonomous security control. However, in doing so, the security of the system should not be compromised. Due to the diversity of kernel security frameworks and their different design perspectives and details, the design can hardly be generic. But we try to abstract the commonness by studying two commonly used kernel security frameworks, namely IMA and AppArmor, and hopefully it may provide useful guidance for other kernel security frameworks and eventually lead to a generic design.

Autonomous Security Control. By autonomous security control, we mean that *individual security namespaces can govern their own security*. Specifically, we would like our design to have the following three properties:

- The processes associated with a security namespace will be under security control of that namespace[1].
- The principal who owns a security namespace can define security policy for that namespace, independently from other security namespaces and the native system.
- Security states (e.g., logs, alerts, measurements and etc.) are maintained and accessed independently.

Security. By security we mean that when there are two or more principals on the system (including the native), *one principal cannot leverage the security namespace abstraction to compromise the security of another principal*. Here the principals refer to parties with independent security interests and policies (i.e., container owners and native system owner) but share the same kernel. The security of a principal refers to the security requirements of the principal, expressed by his or her security policy. In other words, our design should not satisfy a principal's security requirements at the cost of another principal. Only when all principals' security requirements are satisfied we say that the overall system is secure.

The strawman design satisfies the autonomous security control, but fails to meet the security requirements. The focus of this paper is thus to investigate the design of security namespace abstraction that can achieve autonomous security control without violating security, and

[1]It does not necessarily mean that the processes will only be under security control of that namespace.

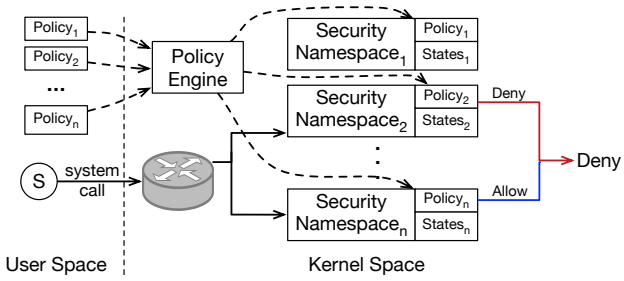

Figure 4: Design overview. A subject's operation is routed to security namespaces who may have an opinion about the operation. Each involved security namespace independently makes a security decision, and the operation is allowed if all involved security namespaces allow the operation.

above attack examples show that how to relax the global and mandatory assumption of security frameworks represents a control point in the tussle.

3.4 Security Model

In this work, we assume the trustworthiness of the kernel. The security frameworks and their namespace implementations reside in kernel space and they can be trusted to enforce the security policies specified by their owners. We do not trust any userspace processes, privileged or unprivileged, on native or in container. They are targets of confinements of security namespaces. In practice, there are often certain userspace processes responsible for loading security policies into the kernel. Such processes are not trusted as well. The kernel ensures the integrity of the policies being loaded by either attesting policy integrity to the policy maker or accepting only policies with valid maker signature. In addition, we do not assume mutual trust among principals on a system. It is the design goal of security namespace abstraction to prevent one principal from abusing the abstraction to compromise security of another principal.

In this paper, we do not aim to provide an unified abstraction for all kernel security frameworks. Instead, each kernel security framework will have its own security namespace abstraction. We leave it for the future work to provide an unified abstraction and functions such as stacking [32]. In addition, although we examine the challenges in applying the design to SELinux (Section 9), we do not claim that the design is already generic. We leave it for the future work to further study the generality of the design and apply it to other kernel security frameworks. Side channel attacks are also out of scope of this paper.

4 Solution Overview

The strawman design shown in Figure 2 provides a straightforward way for containers to achieve autonomous security control. However, the way it relaxes the global and mandatory assumption only considers a single principal's security interest (i.e., the security namespace that is associated with the process), therefore potentially violating the security of other principals on the system. We argue that when relaxing the global and mandatory assumption of security frameworks, we have to account for the security expectations of *all* principals on the system. Only in this way, we can ensure that the autonomous security control of one principal does not come at the cost of another principal. This boils down to two security invariant that we believe must be maintained when global and mandatory assumption are relaxed:

- Given an operation from a process, *all* security namespaces that have an opinion about the operation (i.e., expressed via its security policy) should be made aware of the operation.

- Only if all security namespaces that have an opinion about the operation allows the operation will the operation be allowed by the system.

The first invariant addresses the concern of relaxing the global assumption of security frameworks. Although a security namespace no longer sees every operation on the system, it should be able to see *all* operations that may affect its security. The second invariant addresses the concern of relaxing the mandatory assumption of security frameworks. Every security namespace that is affected by an operation can apply policies over the operation. However, only if all policies allow the operation will the operation be allowed by the system.

Based on this insight, we propose a security namespace abstraction design that is secure, by augmenting the strawman design with a routing based mechanism, as shown in Figure 4.

First, as in the strawman design, we virtualized a security framework into virtual instances. Each virtual instance becomes a security namespace and controls a group of processes associated with it (e.g., security namespace$_1$ to security namespace$_n$ in Figure 4). Each security namespace shares the same code base in kernel, but independently enforce its own security policies and maintains independent data structures for security states. Conceptually, they are isolated from each other.

Second, we added a component named Operation Router to the standard operation mediation process of security frameworks in kernel. When a process performs an operation (i.e., system call), the operation is first sent to the Operation Router. Based on the operation, the Operation Router decides *which security namespaces should be made aware of the operation*. The key challenge in this step is to ensure that every security

namespace whose security might be affected by an operation is made aware of the operation; this underpins security while allowing relaxation of the global assumption of security frameworks. The router then routes the operation to those security namespaces. Each security namespace makes their security decisions independently.

After each security namespaces made their security decisions, a final decision is made by the system, taking into consideration of all those security decisions. To relax mandatory assumption in a secure way, we took a conservative approach which intersects (i.e., apply AND operator) all those security decisions. Thus, only if all security namespaces that were made aware of the operation allow an operation will it be allowed by the system.

Finally, we added a component named Policy Engine that detects and identifies policy conflicts among security namespaces at policy load time. Policy conflicts result in different security decisions at runtime, where an operation allowed by one security namespace is denied by another. Since a security namespace cannot (and should not) inspect security states of another, debugging the cause of the denial becomes a problem. This is particularly problematic for the container cloud case since the container owners do not want containerized applications to encounter any unexpected runtime resource access errors. Therefore we designed the policy engine to detect and identify policy conflicts at policy load time and inform the namespace owner the potential conflicts. The policy owner may decide to revise her security policy to avoid conflicts, or continue to use the system but be aware of the potential runtime denials, or change to a new system where there is no conflicts.

5 Operation Router

The Operation Router identifies the set of security namespaces that may have an opinion about an operation and routes the operation to those security namespaces. To decide which security namespace may have an opinion about an operation, we leverage a simple insight: a security namespace may have an opinion about an operation if by not routing the operation to the security namespace, the two security assumptions, global and mandatory, might be broken for the security namespace. Since an operation can be written as an authorization tuple (s, o, op), we discuss from subject's and object's perspective separately.

5.1 A Subject's Perspective

Security framework makes an implicit assumption about its globalness: it controls *all* subjects on a system that are stemmed from the very first subject that it sees. For native system, this means all subjects forked from *init* (i.e., *PID* 1). For a security namespace, this means all the subjects forked from the first subject of the security namespace. The attack example shown in Figure 3 occurs due to that it breaks this implicit assumption. P_1 is a descendant of P. However, by assigning security control of P_1 to a new security namespace, security namespace NS_{Native} no longer confines P_1, therefore breaking the implicit global assumption of NS_{Native}.

Therefore, a security namespace would have an opinion about an operation if, by removing the operation, the implicit global assumption of the security namespace is broken. To achieve autonomous security control, a subject is under direct control of the security namespace that it is associated with. However, at the same time, since the subject stems from other subjects that may be associated with other security namespaces, those security namespaces also implicitly assume control of the subject. If an operation involving the subject is not routed to those security namespaces, their global assumptions are broken therefore compromising their security. As a result, the Operation Router needs to account for the subject's perspective by not only route an operation to the security namespace that the subject is associated with, but also all security namespaces that the direct ancestors of the subject are associated with.

5.2 An Object's Perspective

Security policy is often a whitelist, enumerating allowed operations from subjects over objects. The mandatory assumption of a security framework implies that, other than those allowed operations, no other operations should be performed over the objects[2]. In other words, a security namespace implicitly assumes a complete (and autonomous) control over the objects that it may access. The attack example 2 shown in Section 3.2 occurs due to that it breaks this mandatory assumption. In the attack, security namespace NS_1 assumes high integrity of file f by ensuring that the file is read only to all its subjects. However, due to the file is also accessible to another security namespace NS_2, NS_2 may allow its subjects to write to f in arbitrary way. Therefore, when subjects from NS_1 access the file, security of NS_1 is compromised without NS_1 is being aware of.

Due to the assumption of complete control over objects, a security namespace may have an opinion about an operation even if the subject of the operation is not under its control. Only in this way can a security namespace ensure that there are no unexpected operations over the objects that its subjects may ever access. As a result, theoretically, the Operation Router needs to account for the object's perspective by routing an operation to all security namespaces whose subjects may ever access the object of the operation to ensure that all their security

[2]Mandatory assumption also implies that subjects should not perform any additional operations that are not allowed by the policy. But it is already covered by the subject's perspective.

expectations are met.

To decide if an object may ever be accessed by subjects of a security namespace, the Operation Router leverages the *resource visibility* defined by the resource namespaces (e.g., mount, network and etc.). The resource namespaces define the visibility of subjects to objects. As long as an object is visible to subjects of a security namespace, it may be accessed by those subjects.

5.3 Shared Objects and Authority

Since security namespaces implicitly assume complete control over objects that they may access, ideally each security namespace is coupled with its own resource namespaces therefore having its own isolated sets of objects. However, in practice, certain objects can be accessed by multiple security namespaces. For example, the `/proc` and `/sys` filesystems and the objects on them are often shared among different containers on a host. Such sharing may lead to two practical issues. First, due to the whitelist nature of security policy, a security namespace allows only its own operations over the object and naturally denies operations from other security namespaces that share access to the object. This results in an unusable system. Second, if the Operation Router routes one security namespace's operation to another security namespace due to that they share access to an object, it may become a privacy breach. For example, a container may not want its operation over `/proc` to be known to another container.

To address this practical concern, we have to adjust policy language of existing security frameworks to make the implicit mandatory assumption explicit. We introduce two new decorators to the policy language, *authority* and *external*. In a security policy, if a security namespace declares authority over an object, its policy over the object becomes mandatory—all the operations over the object, either from subjects associated with the security namespace or other security namespaces, will be routed to the security namespace for mediation. In contrast, if a security namespace does not have authority declared for an object in its security policy, the policy over the object will only be locally effective, meaning that the security namespace will not be able to control how subjects from other security namespaces access the object. The goal of the authority decorator is to let security namespaces explicitly declare their mandatory assumption.

The external decorator is used along with the authority decorator. When a security namespace declares authority over an object, it may define security policies for subjects that are invisible to the security namespace (i.e., associated with other security namespaces). Such invisible subjects are decorated with keyword external in the security policy. A security namespace will assign access permissions to external decorated subjects just like its own

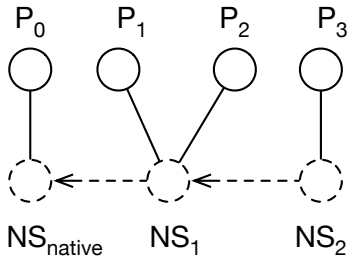

Figure 5: Security namespace graph.

subjects, but all external decorated subjects will have the same permissions because they are indistinguishable to the security namespace. For example, when protecting a read-only file using a lattice policy, a security namespace can assign invisible subjects with integrity label $\{a\}$ and the file with integrity label $\{a, b\}$ to ensure read-onlyness. However, label $\{a\}$ will be universal for all the invisible subjects of the security namespace, because from the security namespace's perspective, those subjects are invisible therefore indistinguishable.

To prevent a security namespace from arbitrarily declaring authority therefore launching denial of service attacks to other security namespaces, the ability to declare authority is tightly controlled by the system. We use a capability-like model where the ability to declare authority over an object is treated like a capability. When an object is created, the security namespace that creates the object is granted the capability. It may use the capability, by declaring the authority in its security policy, or delegate the capability to other security namespaces. In practice, the delegation often happens between parent and child security namespaces.

5.4 Routing Algorithm

Combining the two perspectives and the practical constraint, we can then define a routing algorithm for the Operation Router that meets our goal: given an operation, all security namespaces that may have an opinion about an operation are made aware of the operation. The algorithm is constructed around two data structures, namely a security namespace graph and an object authority table which are maintained and updated in the kernel while new security namespaces are being created and security policies are being loaded.

A security namespace graph is a graph that maintains the <subject $\leftrightarrow$ namespace> and <namespace $\leftrightarrow$ namespace> mappings. It has two types of vertices as shown in Figure 5. One type of vertices are the subjects and another type of vertices are the security namespaces. An undirected edge connects the two. Between security namespace vertices, there is a directed edge, pointing

Input: subject s and object o, security namespace graph G, object authority table T
Output: set of security namespaces Φ

1: $\Phi \leftarrow$native ▷ Native is the ancestor for any security namespace
2: $n \leftarrow$CURRENT(s, G) ▷ Get the namespace that s is associated with
3: **while** $n \neq$ native **do** ▷ Recursively add all n's ancestors
4: $\Phi \leftarrow \Phi \cup n$
5: $n \leftarrow$GET_PARENT(n, G)
6: $\Phi \leftarrow \Phi \cup$AUTHORITY($o$, T) ▷ Get namespaces that declared authority over o
7: return Φ

Figure 6: An algorithm for routing an operation to security namespaces who may have an opinion about the operation.

from the child to its direct parent[3]. The security namespace graph captures the subject's perspective when the Operation Router routes an operation.

Another data structure is the object authority table. An object authority table maintains the mapping between an object to the corresponding security namespaces that have the capability to declare authority over the object. It also maintains the information of whether or not the security namespace actually declared the authority in its security policy. The object authority table is updated when a new object (e.g., inode) is created within the kernel and when new authority delegation happens. The object authority table helps capture the object's perspective under the practical constraint when the Operation Router routes and operation.

Using these two data structures, we define the routing algorithm as shown in Figure 6. The algorithm takes as input the subject and object of an operation, and produces a set of security namespaces that need to be made aware of the operation. At the high level, the algorithm works as the follows: it first recursively add the current security namespace that the subject runs in and all its ancestors security namespaces (down to the native) into the output set. Then it finds all the security namespaces that hold authority over the object and adds them to the output set.

6 Policy Engine

The goal of Policy Engine is to detect policy conflicts at policy load time. Policy conflicts would result in different security decisions, where an operation allowed by a security namespace is denied by another. Such denial often cannot be debugged at runtime, as security namespaces are isolated from each other. This may affect the practical usability of the security namespace abstraction, considering a containerized application can fail unexpectedly. To address this concern, our insight is to move

[3]The parent and child relationship is defined with respect to the subjects. If subjects of a security namespace are forked from subjects of another security namespace, then the two security namespace has a parent and child relationship.

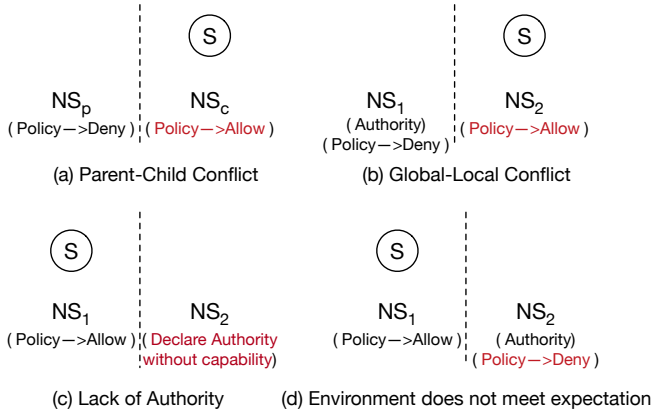

Figure 7: Four types of policy conflicts. Existing and new security namespaces are separated by the dashed line. Conflicting policies are marked in red.

the conflict detection to policy load time and inform respective parties of the potential conflicts. The conflicting party may revise her security policy to avoid conflicts, or continue using the system but be aware of the potential conflicts, or abort using the system as the system cannot meet her expectations. The Policy Engine detects two types of conflicts: DoS conflicts and expectation conflicts. We discuss them separately in this section.

6.1 DoS Conflicts

When a security namespace loads its security policy, if its subjects might be denied of performing an operation by other security namespaces on the system, we call it denial of service conflicts (DoS conflicts). The name comes from the fact that the operation will be eventually denied (after intersecting all security decisions) even though policy of the security namespace explicitly allows the operation.

There are two types of DoS conflicts, corresponding to the subject's and object's perspective of the operation routing. The first type is the ancestor-descendant conflict, where a descendant security namespace's policy violates its ancestors', as shown in Figure 7(a). Recall from Section 5.1, a subject is under control of its own security namespace and all its ancestors. Thus a DoS conflict may arise if the descendant loads a policy that allows an operation but its ancestors would deny it. The second type of conflict is the global-local conflict, where a security namespace's security policy violates an authoritative one, as shown in Figure 7(b). In this case, a security namespace loads a policy that allows an operation over an object (i.e., local), but the operation would be denied by other security namespaces that hold authority over the object (i.e., global).

The Policy Engine detects DoS conflicts using a conflict detection algorithm, as shown in Figure 8. At a high

Input: set of existing security policies S, new security policy s
Output: set of conflicting rules Φ

1: $\Phi \leftarrow \emptyset$
2: $S' \leftarrow$ROUTING_ALG(S) ▷ Set of policies that need to be considered
3: $P_o \leftarrow$PERMISSIONS(S') ▷ Projected permissions of S'
4: $P_n \leftarrow$PERMISSIONS(s) ▷ Projected permissions of s
5: **if** $P_n \not\subseteq P_o$ **then**
6: $\Phi \leftarrow$CONFLICTING_RULES(P_o, P_n)
7: return Φ

Figure 8: An algorithm for detecting DoS conflicts.

level, the algorithm takes as input the security policies of existing security namespaces and the new one, and try to identify if the newly loaded security policy would introduce additional access permissions for the subjects. Such additional permissions are the root cause of an operation being allowed by the new security namespace, but denied by others. Specifically, the algorithm first computes the set of security namespaces whose security policies need to be considered. This is based on the routing algorithm discussed in previous section. Next, by analyzing the policies, the algorithm computes two projected permission sets of each and every subject associated with the new security namespace[4], one based on security policies of existing security namespaces and another based on the newly loaded policy. The permission set of the new security policy should always be a subset of the existing security policies, to ensure that no additional permissions are introduced.

When conflicts are detected, the owner of a security namespace are given two choices. She may revise her security policy to avoid the conflicts, or loading the security policy anyway with the risk of her operations being denied unexpectedly. However, we should note that even in the second case, she only risks DoS but no compromise of security as any operation denied by her own policy will not be executed by the system.

6.2 Expectation Conflicts

When a security namespace loads its security policy, if the policy could deny operations from other security namespaces, we call it expectation conflicts. Expectation conflicts may lead to unexpected operation denials to existing security namespaces, so the system will refuse to load a security policy that may cause expectation conflicts. As its name suggests, the expectation conflicts represent that the existing system cannot possibly meet the security expectation of a new security namespace, therefore the owner of the new namespace should either revise her policy, or abort using the system.

[4]This is a projection, as at the policy load time, there is often no subject or only a single subject of that security namespace actually created on the system, depending on who loads the security policy.

In practice, there are two types of expectation conflicts, both of which can be easily detected by the Policy Engine using the object authority table. The first type of expectation conflicts is shown in Figure 7(c), where in its security policy a security namespace declares authority over an object but it does not have the capability to declare the authority. In this case, the Policy Engine would refuse to load the policy and render a lack of authority error. This delivers an explicit message to the owner of the security namespace that the system cannot meet her security expectation, and she shall not run with the false impression of security (e.g., a security namespace believes a file is read-only file, but it is actually writable to other security namespaces). The second type of expectation conflicts is shown in Figure 7(d), where a security namespace has the capability and declares authority over an object. However, its policy over the object conflicts with policies of existing security namespaces on the system (i.e., it would deny an operation which was already allowed by others). In this case, the Policy Engine would refuse to load the policy as well, since loading the policy may cause unexpected operation denials of other security namespaces. Here the authority represents a right to claim mandatory security over an object, but not a right to override security decisions of others.

7 Implementation

To demonstrate our design, we implemented security namespace abstractions for two widely used kernel security frameworks, IMA and AppArmor. The modification to kernel is $\sim$1.1K and $\sim$1.5K LOC, respectively. The IMA namespace implementation is already open sourced[5] and under review by the kernel community.

7.1 IMA namespace

Operation Router. IMA protects the *integrity* of a system by measuring and appraising what subjects on a system may read or execute. It has a narrow focus on the subject's perspective of access control. This simplifies the implementation of the Operation Router. When a subject reads or executes a file, the Operation Router simply routes the operation to the IMA namespace associated with the subject, and all its ancestor IMA namespaces up to the native.

Measuring Files. Conceptually, each IMA namespace would measure a file independently. However, this can be both expensive (i.e., calculating hash of a file multiple times) and unnecessary. Instead, we re-used the measurement cache in our implementation and make it a global data structure shared by all the IMA namespaces. After the first IMA namespace calculates a mea-

[5]https://git.kernel.org/pub/scm/linux/kernel/git/zohar/linux-integrity.git/log/?h=next-namespacing-experimental

surement of the file, the measurement is put on a global measurement cache. Subsequent IMA namespaces will check with the cache to detect the presence of the measurement and only calculate if it is not present. However, each IMA namespace would still maintain its own measurement list and independently decide whether or not to include the measurement on its list. To some extent, we did not fully virtualize IMA. Instead, we only virtualized the data structures and interfaces that are exposed to userspace to make it appear that they have their own isolated instance of IMA.

File Appraisal and Policy Engine. IMA appraisal prevents unauthorized file from being read or executed by validating file signatures against pre-installed certificates. The certificates are traditionally specified by the system admin and are stored on the `_ima` keyring[6]. To support appraisal, we need to first separate `_ima` keyring such that each IMA namespace can install their own set of certificates to validate files independently. But unfortunately, the existing kernel keyring subsystem does not support namespace abstraction. As a workaround, we implemented a dynamic keyring renaming mechanism. The idea is to allocate a keyring with a different name (randomly generated) in the kernel every time an IMA namespace is created. This keyring is associated with the namespace for its entire life cycle. The namespace owner can thus load and update certificates for his namespace using this keyring. To prevent one namespace from updating the keyring of another namespace, we rely on the access control mechanisms in keyring subsystem. A cleaner way to implement this is to provide a namespace abstraction for the kernel keyring subsystem, which is an ongoing effort of a working group. We will integrate it with IMA namespace once it is done. After separating the `_ima` keyring, each IMA namespaces could independently load its certificates. The certificates are essentially whitelist policies deciding which file can be read or executed by the namespace. To detect policy conflicts at load time, the Policy Engine simply checks if is the certificates loaded by a security namespace is a subset of existing security namespaces.

7.2 AppArmor Namespace

Operation Router. AppArmor implements the *targeted security* MAC policy, which tries to confine privileged subjects on a system. Its original focus is the subject. To extend it with an object's perspective, we made two modifications. First, each AppArmor namespace is assigned with a base profile. In the base profile, a security namespace can declare authority over objects. Other profiles in the namespace will inherit the base profile. Second, we implemented a handler function in the kernel to detect any changes to the base profile so that the Operation Router can be notified to parse the base profile and update its object authority table accordingly.

Pathname Collision. In AppArmor, subjects and objects are identified using their pathnames. This becomes problematic when an AppArmor namespace needs to differentiate subjects or objects in different namespaces. One way to address this is to use absolute pathnames (e.g., `/sbin/dhclient` and `/var/lib/docker/instance-001/sbin/dhclient`). The downside of this approach is, however, there may not always exist a valid absolute pathname. In our implementation, we leveraged the built-in *profile namespace* primitive of AppArmor policy. A profile namespace provides scoping for the pathnames. By creating a profile namespace per AppArmor namespace and assigning it an identifier, we therefore enable AppArmor namespaces to specify a policy using the combination of profile namespace identifier and the relative pathnames in the profile.

Policy Engine. We construct our Policy Engine based on the extended Hybrid Finite Automata (eHFA) [16] of AppArmor. The Policy Engine first identifies the set of policy profiles (including the base profiles) that may be associated with the same subject. Then taking these profiles as input, the Policy Engine tries to construct eHFA. During this process, the Policy Engine will sort and merges rules from profiles, and detect conflicts if there are any.

7.3 Filesystem Interfaces

Both IMA and AppArmor accepts policies and exports security states through `securityfs` interface. Ideally, each security namespace should be able to mount its own `securityfs`. However, currently this is not allowed by the kernel. As a temporary fix, we used the `proc` filesystem instead. The idea is to place the security states and policy files that correspond to a security namespace under the directories of the processes that run within that namespace. We are working with the kernel community to fix the permission issue for mounting `securityfs` (e.g., using jump link).

7.4 Using Security Namespace

In order for userspace program to create an IMA or AppArmor namespace, we extended the *clone* and *unshare* system call. Taking *clone* system call for example, we added a new constant `CLONE_NEWIMA` and `CLONE_NEWAPPARMOR` that userspace program can specify along with other namespace constants[7]. The result is that kernel will clone the process and run it within

[6] Keyring is a kernel subsystem for retaining and caching keys.

[7] There are some debates in kernel community whether or not constants for security namespaces should be on their own. This may affect the interface in future.

the new IMA or AppArmor namespace. The changes to userspace program are minimal. In fact, to make IMA and AppArmor available to Docker, we extended the libcontainer [29] by introducing less than 20 LOC.

8 Evaluation

In this section, we evaluate IMA and AppArmor namespaces from their security effectiveness and performance.

8.1 Security Effectiveness

8.1.1 IMA Namespace

We evaluate the security effectiveness of IMA namespace from two perspectives: autonomous security control and security. To evaluate autonomous security control, we emulate a security setting identical to most commercial container clouds where container host applies a very *lenient* integrity policy (i.e., allow *any* immutable files to be run within the containers). Containers, on the other hand, apply a *strict* integrity policy using IMA namespace (i.e., only code signed by container owner may run in container). We created three types of malicious code that an attacker may run within a container, i.e., code that was not signed, code signed with unknown key, and modified code with an invalid signature. The IMA namespace of container successfully prevents all of them from running. In addition, the individual measurement list of IMA namespace enables the container to attest its integrity to a remote party independently. This experiment demonstrates that IMA namespace enables containers to have their autonomous integrity control, independent from the integrity policy that host system applies.

The second experiment evaluates security, by demonstrating that containers cannot leverage IMA namespace to violate the integrity policy of the host. In this experiment, we emulate a scenario where the host system wants to apply certain integrity control over its containers (e.g., prevent container from hosting malware by allowing only code signed by Ubuntu to run). Containers, on the other hand, try to break it by allowing anything to run in its IMA namespace. In this case, the Policy Engine successfully detects the DoS conflict, and if the container continues loading the policy, code in container that is not signed by Ubuntu is prevented from being run by the native IMA namespace. This experiment shows that despite enabling autonomous security control, IMA namespace will not compromise the integrity of any principal.

Conflict Analysis. IMA supports two sets of security policies: one for measurement that determines which files to measure, and one for appraisal that determines the right measurements for each file. The measurement policy only affects which files each individual IMA namespace will measure, therefore there are no conflicts introduced because each IMA namespace has its independent measurement list. In other words, integrity attestation of individual containers are conflict-free. The appraisal policy may introduce conflicts since a measurement "good" for one IMA namespace may not be "good" for another, as evidenced by above examples.

Table 2: Enforcing both system and container profiles over applications.

Application Profile	Conflicting Rules
Apache2	/proc/[pid]/attr/current *rw*
NTP	/dev/pps[0-9]* *rw*
firefox	/proc/ *r*
chrome	/proc/ *r*
MySQL, Perl, PHP5 OpenSSL, Samba, Ruby, Python Subversion, BitTorrent, Bash dhclient, dnsmasq, Squid OpenLDAP(slapd), nmbd, Tor	None

To avoid appraisal policy conflicts, container owners will have to ensure that the files they allow to load in containers are a subset of the files allowed by the host system. This, in our implementation, means that the certificates that a container owner may load on her `_ima` keyring will be a subset of the certificates that the host system owner loads on the host system's `_ima` keyring. In practice, conflicts are not common since container clouds tend to have a lenient integrity policy (e.g., allow any executable to run within container). However, in a case where a container cloud does have certain integrity requirements over containers, the cloud vendor will have to explicitly inform its users of what they can or cannot run inside their containers (i.e., by revealing the list of host certificates), in order to assist container owners to avoid conflicts.

8.1.2 AppArmor Namespace

According to the official Ubuntu LXC documentation [34]:

> *Programs in a container cannot be further confined — for instance, MySQL runs under the container profile (protecting the host) but will not be able to enter the MySQL profile (to protect the container).*

We thus evaluate the security effectiveness of the AppArmor namespace by showing that container owners can leverage AppArmor namespace to further confine their applications (i.e., have autonomous security control), just like running applications within a VM or directly on the native system.

We selected 20 programs that have default AppArmor profiles in Ubuntu and run them in a container [8].

[8] There are ∼70 programs that have default AppArmor pro-

Containers apply these profiles in an AppArmor namespace to protect their containerized applications. The native system applies `lxc-start`, `lxc-default` and `docker-default` profiles(also shipped as a default in Ubuntu) in the native AppArmor namespace, in order to protect the host system from accidental or intentional misuse of privileges inside the container. Running them together, we evaluate whether or not the AppArmor namespace indeed enables autonomous security control for container, by protecting the containerized application and the host at the same time. Results are shown in Table 2. As shown in the table, except 4 programs (Apache, ntp, firefox and chrome), the application profiles of the other 16 programs can be directly applied to the container on top of the host system profile. This demonstrates that our AppArmor namespace enables containers to have autonomous security control, independent from the host system. For the four programs, the Policy Engine yields DoS conflicting rules, which means that operations of these programs might be denied by the host profile even if they are allowed by the application profile. This demonstrates that 1) containers may not leverage AppArmor namespace to compromise the host, as these conflicting operation will eventually be denied by the system, and 2) our Policy Engine can inform the container at policy load time such that containers will not run into unexpected runtime resource access errors.

Conflict Analysis. We found that policy conflicts often involve operations over filesystems that are shared across containers (e.g., /proc, /dev, /sys). The reason is that these filesystems have been historically used as an interface between kernel and userspace for exchanging information. On one hand, some information on those filesystems are security sensitive—they may break isolation between containers[19]. Therefore, host system needs to apply a security policy to govern their access. In fact, for the default AppArmor container host profiles, majority of the rules ($\sim$60%) are for governing access to these shared filesystems. On the other hand, applications often need to access information on those filesystems, so such access is allowed by their AppArmor application profile. The challenge is, however, both host's and application's profile are often coarse grained (e.g., "/proc r" for firefox). The coarse granularity of policy may be due to the large amount of information on those filesystems, but it creates conflicts.

To avoid conflicts, one way is to fine tune security policies, at both application side and container host side. For example, it seems not to make much sense for firefox to require read access to all files under `/proc` in order to function. Instead, the application developer, or the container owner, should fine tune the AppArmor policies for their applications to enforce a least privilege. The same applies to container host policies as well. Currently, the AppArmor policies enforced by container hosts are less well understood—it is not thoroughly clear which files under shared filesystems are required by applications at runtime and whether or not they might lead to attacks that can break container isolation. Instead, AppArmor host policies are often revised or extended only after an attack is reported. Ideally, we can design a better container host security policy by examining each and every file under these shared filesystems and fine tune it to fit the application[9], but this can be an extremely challenging task given the large amount of information stored on those shared filesystems and the diversified requirements from the containerized applications.

A more principled way to avoid conflicts is to avoid sharing. One such proposal is to design new namespaces for other types of resources that are currently shared across host and containers. For example, the device namespace proposal [12] can help resolve the conflicts of NTP in Table 2. As an orthogonal work, we are also investigating if it is possible to use multi-layered filesystem to conceal sharing of `/proc`, or at least reduce the exposure of files under the shared filesystems.

files in Ubuntu. They are either part of the distribution or the `apparmor-profiles` package. We selected 20 that are mostly often seen running in containers.

Table 3: Latency for IMA and AppArmor namespace to mediate mmap system call.

mmap(μs)	IMA (stdev)	AppArmor (stdev)	slowdown
No security	1.08 (0.01)	1.08 (0.01)	
Native	1.26 (0.01)	1.38 (0.01)	
Native + 1NS	1.26 (0.01)	1.39 (0.02)	0.7%
Native + 2 NS	1.27 (0.01)	1.39 (0.02)	0.8%
Native + 5 NS	1.27 (0.01)	1.41 (0.02)	2.2%
Native + 10 NS	1.28 (0.01)	1.43 (0.02)	3.5%

8.2 Performance

We examine the performance of IMA and AppArmor namespace by measuring 1) the latency for namespaces to mediate system calls and 2) throughput of containerized applications. Our testbed is a Dell M620 server with 2.4Ghz CPU and 64GB memory, installed with Ubuntu 16.10. The kernel version in test is 4.8.0.

Table 3 shows our latency result. We measured common system calls that are mediated by IMA and AppArmor (e.g., `mmap`, `read`, `execve`, `write`), but due to space constraint, only `mmap` is shown. We evaluated the system call latency from various settings, ranging from no security framework to only the native system to native system plus 10 other security namespaces (i.e., a system call is routed to the native system and 10 other security

[9]Docker already provides some container host AppArmor profiles fine tuned towards specific applications such as Nginx [14].

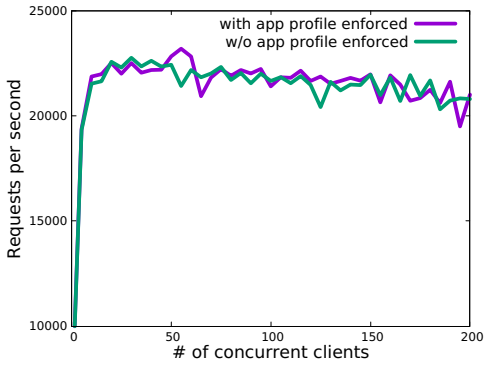

Figure 9: Throughput of containerized Apache with and w/o application AppArmor profile enforced.

namespaces at the same time). Results show that security namespace introduces about 0.7% overhead in the one namespace scenario (the most typical scenario for container cloud) and at most 3.5% overhead even when there are 10 security namespaces in presence. Slowdown for `read` is similar to `mmap`. For `execve` and `write`, the slowdown is even less obvious due `execve` and `write` themselves take longer time to finish. The overhead is almost linear as the number of security namespaces grow[10], because in our current implementation we used a sequential routing to avoid intrusive modifications to the kernel (i.e., system calls are routed sequentially to all affected security namespaces). In theory, since security namespaces are isolated from each other, their mediation of system call can be paralleled leveraging multi-core to minimize the overhead. However, for small number of security namespaces (e.g., one or two), our experience suggests that the added complexity of synchronization can often outweigh the mediation latency.

We also evaluated the macro performance of AppArmor namespace by measuring the throughput of a containerized Apache with and without a default AppArmor profile(on top of a host profile). The result is shown in Figure 9. In the experiment, one host runs a single Docker container containing the Apache and another host runs client sending HTTP requests. As shown in the figure, the throughput is almost identical, since 1) only few of Apache's system calls are actually mediated by AppArmor and 2) latency for single system call mediation is very small as shown above. As a result, we believe our security namespace implementation is practical for the container cloud use case.

[10]Here the number of security namespaces is not referring to the total number of security namespaces on a system, but rather the number of security namespaces that the Operation Router routes to.

9 SELinux and Beyond

By investigating IMA and AppArmor, we hope the lessons we learned can help guide future namespace abstractions for other kernel security frameworks, and eventually lead to a generic and unified security namespace design for all kernel security frameworks. Therefore, in this section we examine challenges in applying the design proposed in this paper to SELinux.

SELinux adopts the type enforcement model to enforce least privilege and multi-level security on a system. SELinux has two features that challenge security namespace designs. The first is the filesystem labeling where a system admin assigns security labels to files (i.e., by setting the extended attributes of files on filesystems). The second is the label transition where subject labels may be changed upon executing new program.

We found the most challenging part of developing a SELinux namespace abstraction is the filesystem labeling, because container filesystems may be loaded dynamically. One possible approach is to have the host system admin to label all the files on a system (i.e., including files within containers). Each SELinux namespace will independently enforce its policy, but its policy must be specified using those labels pre-defined by the host system admin. This approach, however, does not work well in practice. For example, current SELinux policy assigns all subjects in a container with label `svirt_lxc_net_t` and all objects in a container with label `svirt_sandbox_file_t`. Such coarse granularity defeats the purpose of have an SELinux namespace in the first place, since now each SELinux namespace has to work with only one subject label and object label, preventing them from specifying any fine grained security policies.

A more practical approach is to enable SELinux namespaces to independently label filesystems. This means, however, each file may be associated with multiple security labels, depending on how many SELinux namespaces are in control of the file. The kernel will have to maintain the mappings between SELinux namespaces and their views of the security labels and present different security labels accordingly during enforcement. As an example, an web server running in a container can be attached with two labels, `native:svirt_lxc_net_t | container:httpd_t`. The label `svirt_lxc_net_t` is used by the host system during enforcement of the host's SELinux policy and the label `httpd_t` is used by the container during enforcement of the container's SELinux policy.

This approach requires dynamic manipulation of security attributes associated with files during runtime. In addition, files will have multiple SELinux security attributes associated with them. There has been pushback

from the kernel community. One reason is that by allowing runtime manipulation of security attributes without reboot and multiple security attributes at the same time, it may add additional complexity that admins may fail to handle properly. A consensus has yet to be reached within the community.

Since SELinux assigns labels to both subjects and objects, it naturally enables a definition of security from the perspective of both subject and object. Therefore, for enforcement we envision our routing algorithm can be applied without much modification since it already takes into consideration of both perspectives. One thing to note here is that label transition is also part of the subject's perspective, therefore when a subject wants to transition into a new label (e.g., on execution of a binary), not only the SELinux namespace that the subject is associated with should be made aware of the transition, but also all the parent SELinux namespaces.

10 Related Work

VM, Library OS and Container. Virtual machine [66, 58] enables mutually distrusting parties to securely share the same hardware platform therefore becoming one primary success story of the cloud era. However, despite a number of research proposals [17, 21, 62, 64], performance of VM is still not satisfying—it incurs a relatively high spin-up latency and low density [18, 65, 37, 57]. A more efficient solution is the library OS [3, 15, 36, 45]. However, library OS often suffers from compatibility issues for applications running inside and turning a legacy OS into a library OS is a non-trivial task. Container [56, 38] is considered to be an alternative. Containers incurs lower overhead than VM, and allows full compatibility for applications running inside. There are two types of containers, system container and application container. A system container [33, 35, 61] wraps an entire OS into a container, providing system admins and developers an environment similar to traditional virtualization. In contrast, an application container [13, 52] contains a single application, allowing the application to be developed, distributed and deployed in a simple manner. Work presented in this paper can be applied to protect both types of containers.

Container Security. There are a number of security issues identified for container systems. First, the container management program (e.g., docker daemon) often runs as a privileged daemon on a system, making it an appealing target for privilege escalation [47, 46, 48] and confused deputy attacks [67]. To address these concerns, solutions were proposed to enhance container management program with authority check [67] and run it with reduced privilege. Second, the container ecosystem often relies on a public image repository, which can often be leveraged by adversaries to spread malware or launch attacks (similar to issues of VM image repository [4]). Systems such as Clair [9] and DCT [10] were proposed to scan container images for vulnerabilities and/or malware before they are uploaded to the public repository. Third, a number of attacks were found that may break the isolation of containers [55, 50, 51, 25]. To improve the isolation, multiple security mechanisms were adopted such as user namespace [59], seccomp [54] and capability [5]. This paper complements above lines of research by providing kernel security features as a usable function to containers, allowing containers to address their internal threats, much like what a VM or host can do. There is also another line of research aiming to improve the virtualization of container systems. For example, the device namespace abstraction [11] virtualizes physical devices on a system. The time namespace [27] abstraction provides virtualized clocks for containers. Security namespace abstraction follows this line of research. But instead of time and device, the resource it tries to virtualize are kernel security frameworks.

Virtualizing Linux Security Frameworks. There are existing works that try to make Linux security frameworks useful for container systems. For example, a kernel patch [24] for IMA suggested that the IMA measurement list is extended with a container ID, such that during integrity attestation, the measurements will become separable based on containers. As another example, AppArmor and Tomoyo introduced the concept of profile and policy namespace respectively [49, 44]. The goal is to allow certain processes to run under a policy different from the rest of the system. However, these modifications are often adhoc; they do not provide full functionality of kernel security frameworks to container, and they still rely on a centralized authority (i.e., system owner) to specify a global policy, leaving containers no true freedom in enforcing their security independently[11]. In contrast, this works provides a truly decentralized way to allow containers to exercise full functionality of kernel security frameworks. Another line of research is to develop new kernel security frameworks that are stackable and application customizable. For example, Landlock LSM [28] enables userspace applications such as containers to customize their kernel security control. However, they still need to properly handle conflicts when an application is under control of multiple principals on a system, and the policy interfaces are often less familiar and more complex (e.g., eBPF programs) than existing kernel security frameworks.

[11] Contemporary to this work, AppArmor is refining its profile namespace to make it more useful to container alike scenarios. However, it is still under heavy development.

11 Conclusion

In this paper, we presented security namespaces, a kernel abstraction that makes kernel security frameworks available to containers. We first identify the fundamental challenge of enabling containers to have autonomous security control—the global and mandatory assumptions made by the kernel security frameworks. We then develop a novel routing based mechanism that allows the relaxation of these two assumptions without having one container comprising other containers or the host system. To evaluate our design, we built two concrete namespace abstractions for kernel security frameworks, namely the IMA namespace and AppArmor namespace. We show that they allow containers to exercise full functionality of IMA and AppArmor with a modest overhead.

Acknowledgment

The authors thank the following people for their comments and technical contributions: Stefan Berger and Mehmet Kayaalp for work on the IMA namespace implementation; Justin Cormack; the anonymous reviewers; and our shepherd Devdatta Akhawe for their insightful feedback on the paper.

References

[1] AWS Elastic Container Service. https://aws.amazon.com/ecs/.

[2] BATES, A., TIAN, D., BUTLER, K. R. B., AND MOYER, T. Trustworthy whole-system provenance for the linux kernel. In *Proceedings of the 24th USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2015), SEC'15, USENIX Association, pp. 319–334.

[3] BAUMANN, A., LEE, D., FONSECA, P., GLENDENNING, L., LORCH, J. R., BOND, B., OLINSKY, R., AND HUNT, G. C. Composing os extensions safely and efficiently with bascule. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 239–252.

[4] BUGIEL, S., NÜRNBERGER, S., PÖPPELMANN, T., SADEGHI, A., AND SCHNEIDER, T. AmazonIA: When elasticity snaps back. In *Proc. ACM CCS'11*.

[5] Linux Capabilities. http://man7.org/linux/man-pages/man7/capabilities.7.html/.

[6] Change Root. http://man7.org/linux/man-pages/man2/chroot.2.html/.

[7] Break out of chroot jail. https://web.archive.org/web/20160127150916/http://www.bpfh.net/simes/computing/chroot-break.html/.

[8] Is chroot a security feature? https://access.redhat.com/blogs/766093/posts/1975883/.

[9] Docker Vulnerabilities Scan. https://github.com/coreos/clair/.

[10] Content Trust in Docker. https://docs.docker.com/engine/security/trust/content_trust/.

[11] Device Namespace. https://lwn.net/Articles/564854/.

[12] Device Namespace. https://lwn.net/Articles/564854/.

[13] Docker. https://www.docker.com/.

[14] AppArmor profile for Nginx running in Docker. https://github.com/docker/docker.github.io/blob/master/engine/security/apparmor.md.

[15] DOUCEUR, J. R., ELSON, J., HOWELL, J., AND LORCH, J. R. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 339–354.

[16] Extened Hybrid Finite Automata (eHFA). http://wiki.apparmor.net/index.php/TechnicalDoc_HFA.

[17] EIRAKU, H., SHINJO, Y., PU, C., KOH, Y., AND KATO, K. Fast networking with socket-outsourcing in hosted virtual machine environments. In *Proceedings of the 2009 ACM Symposium on Applied Computing* (New York, NY, USA, 2009), SAC '09, ACM, pp. 310–317.

[18] FELTER, W., FERREIRA, A., RAJAMONY, R., AND RUBIO, J. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2015, Philadelphia, PA, USA, March 29-31, 2015* (2015), pp. 171–172.

[19] GAO, X., GU, Z., KAYAALP, M., PENDARAKIS, D., AND WANG, H. Containerleaks: Emerging security threats of information leakages in container clouds. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017* (2017), pp. 237–248.

[20] Google Kubernetes. https://cloud.google.com/kubernetes-engine/.

[21] GUPTA, D., LEE, S., VRABLE, M., SAVAGE, S., SNOEREN, A. C., VARGHESE, G., VOELKER, G. M., AND VAHDAT, A. Difference engine: Harnessing memory redundancy in virtual machines. *Commun. ACM 53*, 10 (Oct. 2010), 85–93.

[22] HAYAWARDH VIJAYAKUMAR AND JOSHUA SCHIFFMAN AND TRENT JAEGER. STING: Finding Name Resolution Vulnerabilities in Programs. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security 2012)* (August 2012). [acceptance rate: 19.4% (43/222)].

[23] IBM Cloud Container Service. https://www.ibm.com/cloud/container-service.

[24] Composite Identifier Field Support for IMA. https://sourceforge.net/p/linux-ima/mailman/message/32844753/.

[25] CVE-2015-3627. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3627.

[26] Amazon Lambda. https://aws.amazon.com/lambda/.

[27] LAMPS, J., NICOL, D. M., AND CAESAR, M. Timekeeper: A lightweight virtual time system for linux. In *Proceedings of the 2Nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (New York, NY, USA, 2014), SIGSIM PADS '14, ACM, pp. 179–186.

[28] Landlock LSM. https://lwn.net/Articles/698226/.

[29] Open Containers. https://github.com/opencontainers/runc/.

[30] Linux Integrity Subsystem. https://sourceforge.net/p/linux-ima/wiki/Home/.

[31] Linux Namespaces. http://man7.org/linux/man-pages/man7/namespaces.7.html/.

[32] LSM Stacking. https://lwn.net/Articles/635771/.

[33] LXC Linux Containers. https://linuxcontainers.org/lxc/introduction/.

[34] LXC - Official Ubuntu Documentation. https://help.ubuntu.com/lts/serverguide/lxc.html#lxc-apparmor/.

[35] LXD Linux Containers. https://linuxcontainers.org/lxd/introduction/.

[36] MADHAVAPEDDY, A., MORTIER, R., ROTSOS, C., SCOTT, D., SINGH, B., GAZAGNAIRE, T., SMITH, S., HAND, S., AND CROWCROFT, J. Unikernels: Library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2013), ASPLOS '13, ACM, pp. 461–472.

[37] MATTHEWS, J. N., HU, W., HAPUARACHCHI, M., DESHANE, T., DIMATOS, D., HAMILTON, G., MCCABE, M., AND OWENS, J. Quantifying the performance isolation properties of virtualization systems. In *Proceedings of the 2007 Workshop on Experimental Computer Science* (New York, NY, USA, 2007), ExpCS '07, ACM.

[38] MERKEL, D. Docker: Lightweight linux containers for consistent development and deployment. *Linux J. 2014*, 239 (Mar. 2014).

[39] Microservice Architecture. http://microservices.io/patterns/microservices.html.

[40] Linux Container Security. https://mjg59.dreamwidth.org/33170.html.

[41] AppArmor Linux application security. http://www.novell.com/linux/security/apparmor/, 2008.

[42] Security-enhanced linux. http://www.nsa.gov/selinux.

[43] POHLY, D. J., MCLAUGHLIN, S., MCDANIEL, P., AND BUTLER, K. Hi-fi: Collecting high-fidelity whole-system provenance. In *Proceedings of the 28th Annual Computer Security Applications Conference* (New York, NY, USA, 2012), ACSAC '12, ACM, pp. 259–268.

[44] Tomoyo Policy Namespace. https://tomoyo.osdn.jp/2.5/chapter-14.html.en/.

[45] PORTER, D. E., BOYD-WICKIZER, S., HOWELL, J., OLINSKY, R., AND HUNT, G. C. Rethinking the library os from the top down. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2011), ASPLOS XVI, ACM, pp. 291–304.

[46] CVE-2014-6407. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6407.

[47] CVE-2014-9357. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-9357.

[48] CVE-2015-3631. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3631.

[49] AppArmor Profile Namespace. http://wiki.apparmor.net/index.php/AppArmor_Core_Policy_Reference#Profile_names_and_attachment_specifications/.

[50] Docker ptrace Attack. https://lkml.org/lkml/2015/6/13/191/.

[51] LXC SYS_RAWIO Abuse. https://bugs.launchpad.net/ubuntu/+source/lxc/+bug/1511197/.

[52] rkt-CoreOS. https://coreos.com/rkt/.

[53] SAILER, R., ZHANG, X., JAEGER, T., AND VAN DOORN, L. Design and implementation of a tcg-based integrity measurement architecture. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13* (Berkeley, CA, USA, 2004), SSYM'04, USENIX Association, pp. 16–16.

[54] Linux seccomp. http://man7.org/linux/man-pages/man2/seccomp.2.html/.

[55] Docker Shocker Attack. http://www.openwall.com/lists/oss-security/2014/06/18/4/.

[56] SOLTESZ, S., PÖTZL, H., FIUCZYNSKI, M. E., BAVIER, A., AND PETERSON, L. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), EuroSys '07, ACM, pp. 275–287.

[57] SOLTESZ, S., PÖTZL, H., FIUCZYNSKI, M. E., BAVIER, A., AND PETERSON, L. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), EuroSys '07, ACM, pp. 275–287.

[58] SUGERMAN, J., VENKITACHALAM, G., AND LIM, B.-H. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *Proceedings of the 2002 USENIX Annual Technical Conference* (2001), pp. 1–14.

[59] User Namespace. http://man7.org/linux/man-pages/man7/user_namespaces.7.html/.

[60] VIJAYAKUMAR, H., GE, X., PAYER, M., AND JAEGER, T. JIGSAW: Protecting resource access by inferring programmer expectations. In *Proceedings of the 23rd USENIX Security Symposium* (2014).

[61] Linux-VServer. http://www.linux-vserver.org/Welcome_to_Linux-VServer.org/.

[62] WALDSPURGER, C. A. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev. 36*, SI (Dec. 2002), 181–194.

[63] WATSON, R. N. M., ANDERSON, J., LAURIE, B., AND KENNAWAY, K. Capsicum: Practical capabilities for unix. In *Proceedings of the 19th USENIX Conference on Security* (Berkeley, CA, USA, 2010), USENIX Security'10, USENIX Association, pp. 3–3.

[64] WHITAKER, A., SHAW, M., AND GRIBBLE, S. D. Scale and performance in the denali isolation kernel. *SIGOPS Oper. Syst. Rev. 36*, SI (Dec. 2002), 195–209.

[65] XAVIER, M. G., NEVES, M. V., ROSSI, F. D., FERRETO, T. C., LANGE, T., AND DE ROSE, C. A. F. Performance evaluation of container-based virtualization for high performance computing environments. In *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (Washington, DC, USA, 2013), PDP '13, IEEE Computer Society, pp. 233–240.

[66] Xen Community. Available at http://xen.xensource.com/, 2008.

[67] ZHANG, M., MARINO, D., AND EFSTATHOPOULOS, P. Harbormaster: Policy enforcement for containers. In *7th IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2015, Vancouver, BC, Canada, November 30 - Dec. 3, 2015* (2015), IEEE, pp. 355–362.