# SNAPPY: Programmable Kernel-Level Policies for Containers

Maxime Belair, Sylvie Laniepce, Jean-Marc Menaud

# SNAPPY: Programmable Kernel-Level Policies for Containers

Maxime Bélair
Orange Labs, IMT Atlantique, LS2N
Caen, France
maxime.belair@orange.com

Sylvie Laniepce
Orange Labs
Caen, France
s.laniepce@orange.com

Jean-Marc Menaud
IMT Atlantique, STACK, INRIA, LS2N
Nantes, France
jean-marc.menaud@imt-atlantique.fr

## ABSTRACT

Compared to full virtualization, containerization reduces virtualization overhead and resource usage, offers reduced deployment latency and improves reusability. For these reasons, containerization is massively used in an increasing number of applications.

However, because containers share a full kernel with the host, they are more vulnerable to attacks that may compromise the host and the other containers on the system.

In this paper, we present SNAPPY (Safe Namespaceable And Programmable PolicY), a new framework that allows even unprivileged processes such as containers to safely and dynamically enforce in the kernel fine-grained, stackable and programmable eBPF security policies at runtime. This is done by making working coordinately a new LSM (Linux Security Module) Module, a new security Linux namespace abstraction (`policy_NS`) and eBPF policies enriched with 'dynamic helpers'. This design especially allows to minimize containers' attack surface. Our design may be applied to any processes but is particularly suitable for container-based use cases.

We show that SNAPPY can effectively increase the security level of containers for different use cases, can be easily integrated with the most relevant norms (OCI, Open Container Initiative) and containerization engines (Docker and runC) and has a performance overhead lower than 0.09% in realistic scenarios.

## CCS CONCEPTS

• **Security and privacy → Virtualization and security**; Access control; • **Computer systems organization →** *Cloud computing*;

## KEYWORDS

SNAPPY, Container, Security, Namespace, eBPF, Programmable, Policy, LSM, Kernel, Linux, Implementation, Cloud

**ACM Reference Format:**
Maxime Bélair, Sylvie Laniepce, and Jean-Marc Menaud. 2021. SNAPPY: Programmable Kernel-Level Policies for Containers. In *The 36th ACM/SIGAPP Symposium on Applied Computing (SAC '21), March 22–26, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3412841.3442037


*SAC '21, March 22–26, 2021, Virtual Event, Republic of Korea*

ACM ISBN 978-1-4503-8104-8/21/03…$15.00
https://doi.org/10.1145/3412841.3442037

## 1 INTRODUCTION

Containerization [29] is a lightweight virtualization technique in which containers are virtual domains offering usermode execution context, while sharing the host kernel at the host level. Containers are isolated from each other and from the host. Containers interact with the host kernel by sending system calls (syscalls) through the virtualization boundary.

Isolation between containers and from the host is achieved through Linux namespaces, a 'kernel abstraction that makes it appear to processes within the namespace that they have their own isolated instance of a global resource' [23]. Each namespace is responsible for managing a resource (PID, user, IPC, ...). Thanks to this mechanism, containers benefit from a better isolation level than raw processes. Hence, while raw processes can see, interact or even kill other processes, containers cannot see the processes running in the host or in other containers.

Compared to full virtualization, containerization reduces virtualization overhead and resource usage, offers reduced deployment latency and finally improves reusability [6]. For these reasons, containerization is massively used in an increasing number of applications such as Multi-tenant Cloud, microservices and Cloud Native Computing [12].

Nevertheless, contrarily to full Virtual Machines (VM) that only share with the host a tight ABI (Application Binary interface) to use their virtual hardware, containers share a full Linux kernel, a way bigger API (Application Programming Interface). That means that containers' attack surface is wider than its VM counterpart. For this reason, containers' isolation and security level is by design lower than VMs'. Additionally, in case of misconfiguration or vulnerability, a compromised container may break out of its isolated context and gain root access to the server. For instance, the vulnerability CVE-2019-5736 [11, 27], further analyzed below, lets a maliciously crafted container arbitrarily rewrite the containerization's core software (runC [18], used by most containerization engines) enabling to get full root access over the host, incidentally taking control of the other containers colocated on the node.

Increasing containers' security level towards VM's would be very appealing since it would allow even security-focused business to switch to containers and take advantage of its improved performance and deployability. But, considering that all containers running on a server share the same host, improving containers' security towards VM's is a very hard challenge. This is partially because the concept of container is a purely userspace abstraction based on Linux primitives such as namespaces and cGroups. Hence, the kernel has no clue about the concept of container [9]. Unfortunately, because there is currently no Linux namespace dedicated to security in the kernel mainline, there is no standard means to isolate and customize container's security treatment at kernel level. Hence, it is hard to increase containers' security level towards VM's.

A lot of previous research tried to fill this security gap by improving or developing new kernel abstractions. A lot of research effort is made on Linux Security Modules (LSM) framework [39], the standard security framework which provides a set of kernel hooks where kernel modules (called LSM modules) can insert orthogonal access control models to Unix's Discretionary Access Control (DAC). A line of research aims to stack and namespace (i.e. make cohabit smoothly) the LSM modules [21, 35, 38] to adapt them to containers. But this consists in ad-hoc solutions for each LSM module that can be very complex for instance for SELinux. Also, because LSM modules are designed to protect the whole system, they require administrator rights to be used. Landlock LSM [33] is another line of research aiming to allow even unprivileged processes to apply kernel-based policies in eBPF code [1] (a *safe* and limited language running in the kernel) to the process that created it and its offspring. However, Landlock LSM only allows to define policies for a few LSM hooks related to files, credentials and ptracing thus lacks of generality.

We argue that allowing even unprivileged containers to safely define rich, fine-grained and programmable kernel-level security policies would allow to protect them against a broad range of attacks and thus would be a significant step forward. Such policies must be mandatory once enforced in the kernel so that they cannot be disabled later in case the container gets compromised. This must be done with a minimal performance impact to keep all the advantages of containers over VMs.

In this paper we present SNAPPY (Safe Namespaceable And Programmable PolicY), a new framework that allows even unprivileged processes such as containers to safely enforce at runtime in the kernel rich, fine-grained and programmable policies, customizable per namespace. This allows to minimize their attack surface.

In order to do this, we propose a new security-oriented Linux namespace that we call in this document `policy_NS` responsible for the storage and the management of the kernel-level security policies to be applied to the processes in this `policy_NS` namespace. Therefore, such policies are enforced to any process in its namespace (or one of its descendants' namespaces, as later explained). These programmable policies are written in eBPF. In addition, we propose the concept of 'dynamic helper' to overcome some eBPF limitations while keeping its security properties. We make these abstraction working coordinately in a new LSM module.

Our design may be applied through namespacing to any processes but is particularly suitable for containers. Therefore, although this paper is mostly focused on containers for the sake of conciseness and clarity, our design and implementation are also functional for raw processes. Additionally, because our framework allows to define fine-grained programmable policies in the kernel at runtime, it can be used to mitigate newly discovered vulnerabilities at runtime without modifying the software's source code nor disabling legitimate features. Our design can be seen as a programmable high-performance kernel-based process reference monitor [19].

SNAPPY design has been successfully implemented and tested with on one hand Linux processes and on the other hand container runtimes, runC in compliance with the Open Container Initiative (OCI) runtime specification [17] and Docker [25] as an extension to Dockerfile [5]. Our prototype is relatively small (~ 750 SLOC) and self-contained. It has a very low performance overhead for realistic applications (<0.09% on our use cases) and thus is not detrimental to containers near-native performances. The usage of our framework is illustrated in this paper with concrete use cases.

We use the following threat model in this paper. We trust the kernel space and the host administrator. All other processes are untrusted. This means that we assume the attacker has access to containers, unprivileged binaries in the host, can maintain a connection and send arbitrary data to the containers and their processes.

The main contributions of this paper are:

- A security-focused Linux namespace (`policy_NS`) allowing to define per-namespace policies for LSM hooks ;
- A kernel interface allowing even unprivileged processes to push eBPF-based policies into the kernel, where they are attached to `policy_NS` namespace ;
- *Dynamic eBPF helpers*, loadable at runtime by the administrator to deport sensitive accesses or complex checkings thus allowing eBPF policies making use of these helpers to be rich and fine-grained ;
- The integration of SNAPPY into relevant containers engines (runC and Docker) and standards (OCI and Dockerfile).

The remaining of this paper is organised as follows. Next section discusses the background. We discuss the design and the implementation of SNAPPY in section 3, its integration into containers engines and specification in section 4, some use cases of SNAPPY in section 5. We evaluate the performance of SNAPPY in section 6, we present the related art in section 7 and we discuss our solution and future works in section 8.

## 2 BACKGROUND

### 2.1 Containers and namespaces

While modern OSes provide a good level of hardware abstraction and isolation (e.g. a process cannot see the memory of other processes), the level of software isolation they provide remains low (e.g. even non-root processes can have information about other processes with commands like `ps`).

Virtualization is an old technique that improves the level of software isolation of a process (or a set of processes) from other virtualized processes [4]. In early implementations, the whole OS was virtualized, potentially with extra features such as hypercalls or optimized drivers.

Containerization is a more recent approach than full virtualization (e.g. hardware virtualization) providing OS-level virtualization that is, userspace virtualization while abstracting the kernel. In some regards, containers can be seen as a successor of chroot [8], a first attempt to isolate processes. The 'namespace' kernel abstraction allows processes within the namespace to have their own isolated instance of a global resource (PID, user, IPC, ...). By enabling all namespaces for a container, it is effectively isolated from the rest of the system, allowing to perform OS-level virtualization.

### 2.2 LSM modules and containers

LSM framework [39] is a kernel-level security framework initially targeting Linux (i.e. not containers). It is the standard approach to provide orthogonal access control models [34] to Unix's Discretionary Access Control (DAC). LSM has a flexible design and offers

a broad range of security types. These controls are implemented in LSM modules built above the LSM framework. Due to their common base, these modules can be used interchangeably at boot time. Some of the most widely used LSM modules are SELinux [37], AppArmor [2] and Integrity Measurement Architecture (IMA) [31].

The LSM framework provides a convenient set of kernel hooks placed at kernel strategical points. Modules subscribe to these hooks allowing them to execute security policies just before the kernel would access an internal object (the so called strategical-point) to perform a specific operation requested through a syscall by an userspace process (e.g. file creation, packet reception, ...). LSM-based policies are executed after the coherence of the call have been checked (error checking, DAC checking, ...) and are able to dereference kernel pointers (e.g. the second argument of the syscall `open` is the address of the file path to open) giving them the visibility needed to finely enforce policies. Hence, based on their policies LSM modules can allow or deny accesses.

Yet, LSM integration with containers is limited. Indeed, the configuration of LSM modules is limited to the administrator therefore is not adapted to unprivileged containers. Additionally, LSM modules apply to the whole system and since there is currently no security-focused namespace in the mainline kernel, namespacing LSM modules is a hard challenge [21]. Some of these modules emulate namespacing with an ad-hoc implementation as part of them (i.e. do not rely on the native kernel namespace implementation). This allows to define per "emulated namespace" policies. But such "emulated namespaces" require a lot of development effort, duplicate a standard feature of the kernel and can be bug-prone thus are not a perfect solution.

Security Namespace [38], the closest work we have been inspired from, is an attempt to allow containers to define customized security by namespacing some LSMs to make them useable for containers. This is done by using a new security-focused namespace abstraction and *manually* modifying these LSM modules to make them work using data from their namespace instead of the global data from the `lsmblob`. A proof of concept have been implemented to containerize AppArmor and IMA.

However, the process to make LSMs namespace-ready must be done manually for each module, is complex and bug-prone, especially for SELinux. Hence, Security Namespace does not provide a general model to namespace LSMs. Additionally, the namespace abstraction of this framework can only namespace a single LSM at a time, defined at kernel compilation, hence does not solve the stacking issues faced by LSMs. Finally, every time a monitored LSM operation is made, Security Namespace must check every namespace of the entire system to decide which policy must executed since it needs to know which policy has visibility over the accessed object to know which policy to execute. This leads to a performance drop for huge servers that run hundreds or thousands containers concurrently.

Contrarily to SNAPPY, Security Namespace with a specific capability can declare `authority` over an object allowing to force their policy to be executed even outside their namespace. However, because this `authority` mechanism requires a capability, it cannot be used for the protection of *unprivileged* containers and its malicious usage can easily lead to DOSes (e.g. if a policy globally denies the access to `/proc` or `/sys`).

### 2.3 extended Berkeley Packet Filter

eBPF [1] is a minimalist binary code primarily designed for packet filtering pushed by the userspace into the kernel, where it can be executed. Because eBPF is by nature fully executed in the kernel, it avoids any context switch to filter a packet therefore increases the network throughput compared to approaches such as iptables [30] and is executed with a very low overhead compared to native code. eBPF is a well established kernel feature. This language is intentionally highly limited to make it unable to attack the system. Therefore, when an eBPF program is loaded into the kernel, a verifier ensures that the eBPF program is valid that is, it does not contain forbidden features such as standard loops, pointer arithmetic, or access to the rest of the kernel... The verifier also makes sure that the eBPF always terminates and limits the number of instructions.

Compared to its ancestor, BPF (or cBPF) [24], eBPF especially allows to use helpers. eBPF helpers are a limited number of helper functions written in native language in the kernel that can be called by eBPF programs to abstract complex processes or processes that need to use some kernel data in a safe way. That allows to circumvent some eBPF limitations and deploy more complex policies while keeping the security of eBPF since these helpers can only provide to the eBPF program data that they should have access to. Nevertheless, because there is only a few helpers which do very specific tasks, the versatility of eBPF programs remains low. Because eBPF helpers are a kernel feature, there is currently no way to add additional helpers to extend eBPF features (unless it is implemented manually, the kernel is modified and recompiled and the server is rebooted with the new kernel, which is not possible for most operational contexts).

Although some vulnerabilities have historically been found in this language's verifier [26, 28], a lot of efforts have and are being made to improve its security. This is why, in our opinion, the security level of eBPF is already high and keeps increasing. Therefore, at the time we write this paper, eBPF is arguably the best way to execute programmable security policies in the kernel, as we aim at.

## 3 SNAPPY: DESIGN AND IMPLEMENTATION

In this section we present SNAPPY, a new framework leveraging LSM hooks to allow even unprivileged processes such as containers to safely self-enforce at runtime in the kernel fine-grained, stackable and programmable eBPF policies, customizable per namespace. This approach is especially useful for container environments. We will show in section 4 that SNAPPY can be integrated into relevant container runtimes (runC and Docker) and standards (OCI and Dockerfile). Our framework reaches this goal thanks to the coordinated usage of the primitives described in the previous section. Specifically, SNAPPY relies on a new Linux namespace (`policy_NS`), LSM operations and eBPF policies, that were introduced in the previous section. We first provide an overview of the main components of our framework before detailing them in the next subsections.

SNAPPY defines a new Linux namespace, `policy_NS`, that contains all the elements required to handle the security of processes in this namespace at kernel-level thus allows to define per-namespaces security policies. This is especially useful for containers that may need their own policies, different from host ones and even from the others containers ones.

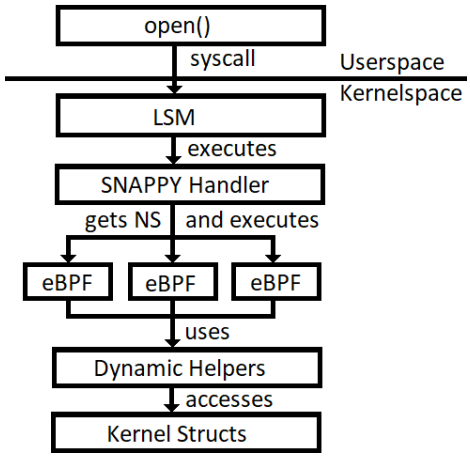


**Figure 1: Global design of SNAPPY**

Any process can load a policy into its namespace to protect a specific LSM operation (opening a file, sending a packet...) by loading in the kernel an eBPF policy through a kernel interface designed for that purpose. This policy is then attached to the `policy_NS` of the process that enforced it, in a field containing the list of policies related to the monitored LSM operation. Once loaded, the policy is applied each time this specific LSM operation is performed by a process in the given namespace or one of its descendant. If any eBPF policy denies the operation, this latter will be in the end denied.

In order to allow unprivileged containers to execute rich and fine-grained policies, we provide a way for the administrator to generate and to push into the kernel eBPF *dynamic* helpers usable by eBPF policies. This process can be done from a simple format (kernel module). After being loaded, the helper can be called from the eBPF code, in a similar way any classical helper would be called. Figure 1 shows SNAPPY's global design. When syscalls are executed, such as `open` syscall, they can trigger LSM hooks, for instance `file_open`. If the LSM operation is monitored by SNAPPY, the SNAPPY handler will execute all the eBPF policies stored in caller's namespace and its descendants that are applicable to the LSM operation (detailed below). These policies can access a set of dynamic helpers that among other, are able to access the kernel structs.

## 3.1 policy_NS Linux namespace

Our new namespace `policy_NS` allows policies to be applied to only a subset of the processes of the system, such as a container. `policy_NS` is a *real* Linux namespace, fully implemented in kernel, and thus can be handled like any other Linux namespace [23]. For instance it can be created using a flag in the syscalls `unshare`, `clone` or `fork`. In this regard, SNAPPY differs from some existing LSM modules which "simply" emulate namespacing with an ad-hoc implementation as part of them.

In order to handle the security of processes, each `policy_NS` contains its own list of policies to be enforced for each LSM operation. These policies are only enforced for processes in the given namespace (or one of its descendant namespaces), i.e. not to the whole system. This is especially useful for containers, that may want different policies than the host and even other containers. Therefore, to apply specific mandatory policies to a given process, it is enough to create it in a new `policy_NS` namespace. Policies enforced to this new namespace are not applied to the rest of the system. That way, our design allows to easily enforce per-namespace policies. It has to be noted that to avoid privilege escalations, a process cannot change of namespace. The only allowed operation for a process is to join a new namespace, child of its current one. Similarly, when a new process is spawned, it can either be in the current namespace or in a new namespace, child of the current one. Our framework allows the administrator to decide which LSM hooks accept SNAPPY policies. That allows to use SNAPPY generically for any LSM hook without suffering the performance overhead that would be generated if we simply enabled all hooks.

To make sure SNAPPY cannot be used by a malicious user to get any new privilege, policies can never be deleted or modified during the whole lifecycle of the namespace. Hence, if an user wants to delete or modify a container's policy, he will have to restart it with the new configuration. This is good from a security perspective and relatively inexpensive since containers can be restarted quickly. On the contrary, policies can still be added at any time. For instance, for containers, some policies can be applied at initialization thus will apply to its whole lifecycle, while others can be applied later (e.g. to block behaviors that are legitimate during the initialization of the container but not later, to mitigate a newly discovered vulnerability without rebooting the container). SNAPPY policies are applied as a "AND" meaning that if any policy denies the operation, this latter will be in the end denied. In other words, adding a policy can only reduce (or let untouched) the set of allowed behaviors. Also, to avoid privilege escalations, a newly created namespace inherits the policies of the current namespace. This is technically done by adding a "parent" field to our `policy_NS` abstraction. When a new namespace is created, this field is initialized with the namespace of the process that created it. Thus, our namespace forms a tree with the root being the `init` namespace and a new branch each time a new namespace is created. When a LSM operation protected by SNAPPY is triggered by a process, a handler executes all the policies applicable to this LSM operation present in the `policy_NS` of the process, then the applicable policies present in its parent `policy_NS` and recursively to the init `policy_NS`. The operation intended by the process is allowed only if all of these policies allow it.

Although in general the parent of a `policy_NS` is the namespace of the process that created it, there is an exception to this rule. Indeed, some containerization engines such as Docker spawn containers through a system daemon. In such scenarios, the newly created container is actually a descendant of this system daemon (e.g. in Docker, the parent of the container is containerd-shim, a descendant of this system daemon [15]). Thus, if we did nothing, our container would inherit of containerd-shim's `policy_NS`, which might be less restrictive than the `policy_NS` of the process that *logically* spawned it (i.e. the `policy_NS` of the process that spawned it in CLI). Thus, it and would break the security of our design. This is why, we force the newly spawned container to be in a `policy_NS` child of its *logical* parent's `policy_NS`. This can be implemented with very tiny modifications over the containerization engine. We implement this for Docker by getting the namespace of the process that "logically" starts the container then modifying the OCI runtime specification of the container (by adding to this JSON file an object of type `"snappy_namespace"` referring to a child of the logical

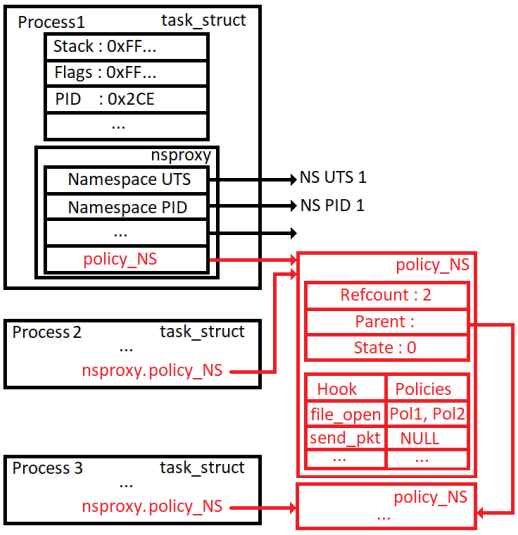


**Figure 2: Example of `policy_NS`**

parent's `policy_NS` in the array `namespaces`). Such a modification is trivial to implement (~12 SLOC) and makes SNAPPY compliant with the OCI – the most relevant specification for containers – thus is easily adaptable to other containerization engines.

Therefore, our namespace tree is totally independent from the process tree. This allows our design to work in any situation, including with nested virtualization.

We allow to execute differentiated policy for the different steps of the namespace's lifecycle by adding in `policy_NS` an integer field representing the `state` of the namespace (e.g. `STATE_INIT = 0`, `STATE_MAINLOOP = 1`, ...). When a namespace is initialized, its state is 0. Because eBPF policies can be applied starting from a particular state, it is possible to apply more and more restrictive policies as the container's state increases. For instance, this allows to apply additional policies to the main loop of a container that would have been too restrictive for its initialization. For security reasons, this field can only be increased i.e. a container can go from `STATE_INIT` to `STATE_MAINLOOP` but the reverse transition is impossible. We typically expect policies to be more and more restrictive.

Figure 2 shows a simplified example of `policy_NS`. It shows that each process contains a pointer to a `policy_NS`, that `policy_NS` contains policies for each LSM hook and that each namespace has a state and a parent field pointing a `policy_NS`.

In conclusion, our design allows to create per-namespaces policies and ensures that under no circumstances a process can have less restrictive policies than at a previous time or than one of its ancestors. This is a really important property from a security point of view because it ensures that the policies intended to a namespace will always be applied.

## 3.2 eBPF-based programmable security policies

We choose the eBPF language to implement our policies for several reasons: 1) Because code-based policies allows to enforce an user-defined logic, they are more flexible than policies based on a fixed system of rules, that are rule-based and configuration-based policies [3], 2) eBPF allows to fully apply policies in kernel (no

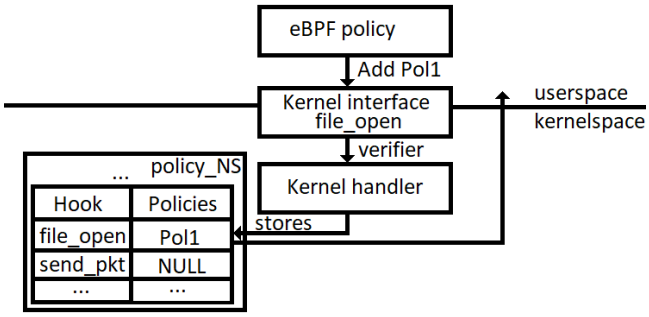


**Figure 3: Loading of eBPF policies**

context switch), leading to good performances, 3) eBPF is relatively mature and well established in Linux community.

In this way, we implemented a new eBPF type for SNAPPY policies. Since we want to have strong guarantees that our solution cannot let processes gain any new privilege nor attack the system, we hardened our eBPF type. We especially deny all accesses to the context pointer from the eBPF code since incorrectly allowing to access this structure might lead in some cases to security issues such as kernel information leaks. To limit our attack surface, we only allow a single helper (further described in the next subsection). Therefore, by default, the attack surface of our design is lower than the attack surface of other types, for instance XDP that allows 15 helpers in Linux 5.7.

Once compiled, our eBPF policy can be enforced by even non-root users for any enabled LSM hooks as illustrated in figure 3. This can be done by pushing the compiled eBPF policy through a specific kernel interface we propose. In the current implementation, a policy can be enforced to the LSM hook `file_open` by sending it through the interface `/sys/kernel/security/snappy/policies/file_open`. Policies can also be pushed using higher level methods such as the OCI Runtime specification. The program is then validated by the eBPF verifier. If the program is valid, it is stored in the right hook of the current process' `policy_NS`, stacked with the other policies for this hook, if any. This policy is enforced as soon as it is stored in the namespace. Hence, any time a process of this namespace (or one of its descendants namespace) will try to execute the LSM-hooked operation monitored by this policy, this latter policy will be executed to check this access, in addition to potential other policies stored in this namespace or a parent of this namespace.

We acknowledge that a fraction of the Linux security community believes that eBPF might not yet be secure enough thus are against the integration in the kernel mainline of new eBPF types usable by unprivileged users. However, since our eBPF policies use a new eBPF type that denies access to the context pointer and contains only a single helper we argue that our attack surface is lower than already integrated use cases such as XDP. Therefore, if a vulnerability was to be found in SNAPPY, it would probably already be present for instance in XDP. Hence we believe that there is no theoretical background against our new eBPF type in the kernel.

## 3.3 eBPF dynamic helpers

With the features used in the two previous sections, we are able to execute policies for specific operations and specific namespaces. Nevertheless, because our eBPF type is highly limited, these features

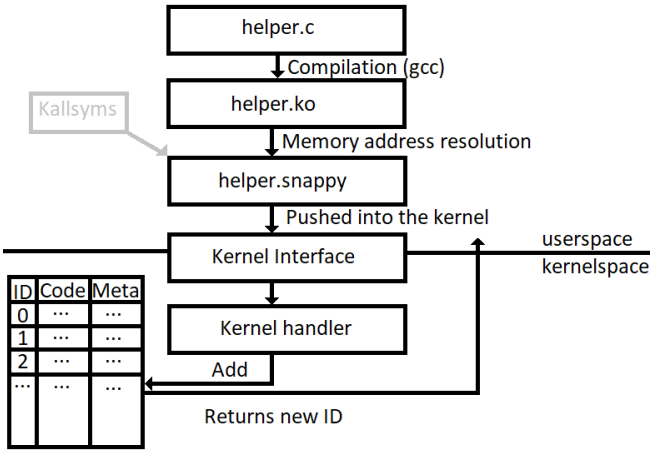


**Figure 4: Loading of dynamic helpers**

alone do not allow fine-grained policies that go far beyond the simple switch "enable/deny the operation".

The standard approach to tackle this issue is to implement helpers for that eBPF type. Such helpers are written in the kernel as native code and are compiled with the kernel. Then, they can be called from the eBPF program just like a regular function in charge of the privileged accesses and the complex operations on behalf of the eBPF code. Yet, as our framework aims to be as generic as possible, that would require to implement a very high number of helpers on the kernel, which would greatly increase our attack surface and would struggle to provide all the features relevant for some use cases. Also, adding a helper would require to recompile the kernel and reboot the machine, which can be a problem for some use cases.

Therefore, we propose the concept of 'dynamic helpers', contrasting with the regular eBPF helpers that are *static* (i.e. integrated in the kernel and no helper can be added without modifying and recompiling the kernel).

The generation and the loading of dynamic helpers is discussed in figure 4. Helpers can be written by the administrator in a relatively simple format: kernel modules. A such kernel module can contain one or several helpers, each of them being a different function. We call *library* this set of helpers.

Libraries are allocated using `__vmalloc`, in an address space terabytes away from the kernel symbols [22]. But since `x86-64` kernels use the small memory model which only allows to reference symbols at less than 2GB of the current instruction, we force the usage of indirect calls with `gcc`. Such indirect calls allow to call symbols using their 64 bit absolute addresses hence avoiding this pitfall.

External symbols' addresses are retrieved using `System.map`. Kernel Address Space Layout Randomization (KASLR) is handled by simply adding the appropriate kernel offset to each entry of the helpers' Global offset Table (GoT) at loadtime. These helpers are pushed to the kernel through a kernel interface (`/sys/kernel/security/snappy/snappy_load`) with some metadata (the LSM hook for which this library is appliable, the helpers' offset for this library, ...). Once loaded, these helpers are ready to be called from the eBPF code.

As discussed in the previous section, our new eBPF type has only one static helper, with the following prototype : `snappy_dynamic_call (int lib_id, int fn_id, void** args)`. This static function acts as

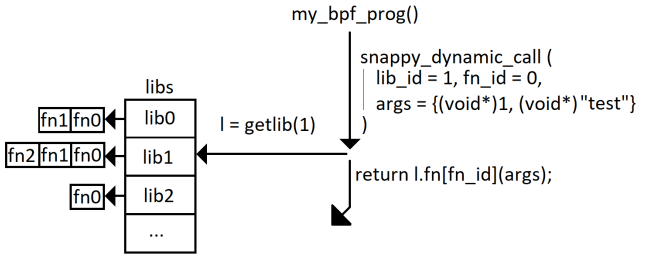


**Figure 5: Calling a dynamic helper from eBPF code**

a proxy to call dynamic helpers as shown in figure 5. Hence, a dynamic helper can be called using the tuple (library id, helper id). This tuple can be found using a specific kernel interface we defined, readable by any user. Any arguments can be passed as `void**` to this dynamic helper, including the eBPF context. When an eBPF program tries to call a dynamic helper, we check that the called library and helper exist. If so, a function pointer is created, referencing the helper (base address of the library + offset of this helper). The helper is finally executed with the given arguments and results of this helper are returned to the eBPF code.

In conclusion, our design effectively allows to enforce at runtime stackable, fine-grained and programmable policies for processes in namespaces such as containers, which was our primary goal.

## 3.4 Security analysis of SNAPPY

In this subection, we evaluate SNAPPY design's security. To do so, we assume that SNAPPY and the underlying softwares used by SNAPPY (eBPF, LSM) do not contain any exploitable vulnerability. This is in our opinion a reasonable assumption since SNAPPY is self-contained, contains a relatively small number of lines of code (~ 750 SLOC) and makes use of standard bricks that are widely used and audited. We show that under these assumptions the system is safe by showing that any entity using SNAPPY cannot compromise more than itself.

- SNAPPY allows even unprivileged users to push eBPF policies to the kernel that will be associated to their own `policy_NS` namespace. Due to eBPF safe nature the policies cannot harm the system. Also, a process cannot create policies that would be applied outside its namespace thus cannot interfere with the rest of the system. This is why, when creating an untrusted process such as a container, it is enough to start it in a new namespace to forbid it to interfere with the current namespace (the policies of the current namespace are applied to the child namespace but the policies of the child namespace are not applied to the current namespace). Thus, the only thing that a process can do is to apply to himself "wrong" policies that can block its expected behavior, which can result in the worst scenario in a DoS. This is not a security issue since a process/container could already DoS itself for instance with `pkill` or `rm -rf`.
- Since eBPF policies cannot be deleted during the whole namespace lifecycle, a process cannot use SNAPPY to gain any new privilege, preventing privilege escalation.

- SNAPPY allows the administrator to load in the kernel eBPF helpers that can be used by the user to easily define rich and fine-grained eBPF policies. While a malicious usage of this feature by an administrator could lead to severe issues such as a deny of service or even a kernel panic, such a behavior could also be done by a malicious administrator without SNAPPY for exemple by deleting the system or by inserting a malicious kernel module.
- SNAPPY enforces limits to prevent a process starving by overusing SNAPPY, the system's ressources and creating a DoS. Hence, we empirically fix a maximum of 32 levels of namespaces and we also set limits to the number of eBPF policies and helpers that can be added to the system.

Therefore, no feature of SNAPPY can be directly used to generate attacks or DoSes.

# 4 INTEGRATION WITH STANDARDS AND CONTAINER RUNTIMES

In this section, we show how SNAPPY can be used by relevant containers engines (runC and Docker) and standards (OCI and Dockerfile). It has to be noted that when integrated with these standards, SNAPPY allows to enforce the security policies from outside of the container that is, from the host. This allows to decouple the containers images from their security and allows security policies to be applied to the whole container's lifecycle.

This latter feature is not possible if policies were pushed on demand through the kernel interface. Indeed, if a policy is pushed from inside a container at runtime, it cannot apply to the whole lifecycle of the container since a container has to be already started and initialized before being able to push policies. Additionally, if the policy is pushed from the host through the kernel interface before starting the container, the policy is also applied to at least a namespace in the host, which is not the desired behavior.

We allow SNAPPY to be used as a configuration by making tiny modification to the Open Container Initiative (OCI) runtime specification the most relevant specification for containers. Indeed, this specification defines since its v1.0.0 the `startContainer` hook that allows to execute commands within the container after it has been fully created but before the user-specific program is started. Hence, SNAPPY policies enforced using this hook are applied to the whole container's lifecycle. If desired, it is possible to apply differentiated policies for each state of the container (initialization, main loop, ...) as shown in section 5.1. Hence, SNAPPY policies can be applied to any container using OCI in a simple way. This adaptation took respectively 34 and 49 SLOC for runC and Docker.

```
"startContainer" : [ {
    "path" : "/bin/snappy_apply",
    "args" : [
      "/mnt/bpf/bpf1.ko", "bprm_check_security",
      "/mnt/bpf/bpf2.ko", "file_open",
    ]
} ]
```

**Listing 1: OCI configuration extract with 2 SNAPPY policies**

Listing 1 shows an example of the OCI runtime Specification's `startContainer` field. The program `snappy_apply` is a simple program in charge of loading eBPF policies to the kernel. Here, two policies are applied to the container. The first protects processes execution (`bprm_check_security`). The second monitors files opening.

Since the OCI specification is a configuration, policies applied via the `startContainer` hook are configuration-based, hence they have the simplicity and the security advantages of this kind of approach. Yet, since SNAPPY policies (eBPF policies) are code-based, they keep the flexibility and the versatility of code-based approaches [3].

SNAPPY can also be integrated to higher level specifications, such as Dockerfile, which is widely used in containerization engines such as Docker or in container orchestrators such as Kubernetes [13]. Since Docker relies internally on the OCI specification it is trivial to integrate SNAPPY to Dockerfile by adding a new command as an extension to the Dockerfile standard such as `ADDSNAPPY <path> <hook_name>` that simply adds the right policy in the OCI's `startContainer` field. Therefore, our design can be easily applied to protect real-life containers.

# 5 USE CASES

## 5.1 Stateful policies enforced through OCI

This subsection shows how simply can an user leverage the OCI runtime specification to enforce stateful SNAPPY policies to protect containers. Although the policy presented there is rather basic, a similar process can be applied for more elaborated and realistic policies and, as later explained in this paper, we are currently working on more elaborated policies that fully exploit SNAPPY's programmability features.

In this example, the container is a computing node that has access to a secret. This container has to create a single connection with the outside to send its computation results to a database.

This way, if the container attempts to create more than one connection in its lifecycle, it means that it is in an unexpected state or compromised (e.g. an attacker tries to exfiltrate the secret). Therefore we should not continue to use it. A way to handle this situation could be to rebuild the container and trigger an alert.

```
int my_main(void* ctx) {
  if(snappy_dynamic_call(ctx, H_PKT, L_PKT_FAMILY, ctx)
     == AF_UNIX) return 0; /* Local packet → OK */
  if(snappy_dynamic_call(ctx, H_ST, L_ST_GET, ctx) > 0)
     return -1; /* State > 0 → denied */
  snappy_dynamic_call(ctx, H_ST, L_ST_INC, ctx);
  return 0; // Increment state and allow
}
```

**Listing 2: eBPF policy for connections management**

We can detect and prevent this attack scenario very simply with a stateful SNAPPY policy for the LSM operation `socket_connect`, as shown in listing 2. This policy works like this. We first check that the communication is not a local connection by getting the packet family with a dynamic helper and comparing it to `AF_UNIX`. We then check if it is the first communication by getting the state of the namespace and comparing it to 0, the initialization value. We then increment its state to track the fact that this connection has been established once. Hence, should the container retry to open a connection, the state will be 1 and we deny the connection attempt.

Because this policy can be applied to the whole container's lifecycle it can be enforced as a configuration through the OCI runtime specification very similarly to the listing 1 and is applied to the LSM hook `socket_connect`. Hence, if we compile this policy under

the name bpf_conn.ko, we can add it as an argument to "/bin/snappy_aply" with the string "bpf_conn.ko", "socket_connect".

The result of this policy can be seen in listing 3. As expected, the first connection attempt is accepted. On contrary, if the container tries to establish a new connection, for instance to exfiltrate the secret, the above mentioned SNAPPY policy will deny it.

```
 $ nc ${ip_normal} ${port} < data # Normal
 $ nc ${somewhere} ${port} < secret #Suspicious
(UNKNOWN) [172.17.0.2] 80 (?) : Connection refused
```

**Listing 3: Container attempting to connect to network a second time**

## 5.2 Vulnerability management with SNAPPY

We show in this subsection that SNAPPY can be used to mitigate at kernel level some vulnerabilities at runtime without disrupting the system and with a very low performance impact. Dynamically mitigate vulnerability thanks to eBPF has recently been stated as a promising line of research [14], thus is a very motivating use case for SNAPPY. Like for other use cases, SNAPPY can be used to perform vulnerability management both by the administrator willing to protect its server or an unprivileged process such as a container willing to defend its "kingdom" (if the features made available by dynamic helpers are relevant to mitigate the vulnerability). These mitigations can be applied to a subset of the system. This way, if a vulnerability is detected and can only affect a container, thanks to our policy_NS, it is possible to apply the mitigation only to this container, effectively correcting the vulnerability for this exposed container without affecting the performance of the rest of the system, which remains unmonitored by this policy.

We demonstrate this claim with the CVE-2019-5736, the latest known vulnerability affecting runC, one of the most relevant brick of containerization. This vulnerability has a CVSS 3.0 score of 8.6 (high), is applicable to runC up to 1.0-rc6, Docker up to 18.09.2, and can also be exploited in Kubernetes, OpenShift, LXC, ... Still, it has to be noted that this vulnerability requires the malicious container to be run as root to be exploited. which is discouraged due to security reasons but remains used in some environments.

Although slightly different variants exist for this attack, the typical scenario is as follow. When executing a process inside a container with commands such as docker exec, the process is spawned from runC using clone. It means that the current executable of the process, available inside the container with /proc/self/exe is runC from the host. Therefore, if a maliciously crafted container makes the container's entrypoint (e.g. /bin/sh) a symlink to /proc/self/exe, when a new process is spawned (e.g. with docker exec), runC will be executed *inside* the container. The main process of the container can wait runC to be run using a busy loop checking the name of the binary using /proc/${pid}/cmdline. When runC is found, the exploit is launched. The malicious container gets a path to runC using open() syscall with the O_PATH flag. Finally, when runC exits, the file can be reopened in write-only mode using /proc/self/fd/${fd}. RunC can now be arbitrarily rewritten for instance to setup a backdoored version or a reverse shell. The attacker has now a full control on the host as root.

One possible way to mitigate this vulnerability at kernel level is to prevent any process from a container to rewrite into runC. Such a mitigation effectively prevents this vulnerability while not disrupting the normal behavior of containers (a container should never rewrite the host's runC). This can easily be done with SNAPPY by adding a kernel-level policy that denies all write access to runC on the file_open hook. This policy can be applied to the containers' policy_NS, i.e. not to the whole system. Listing 4 shows a dummy implementation of a helper for such a policy. It has to be noted that more generic and elegant helpers could be written, but we keep this implementation for the sake of clarity. This code assumes that runC is installed at /sbin/runc. Such a helper can be called by a simple eBPF policy that calls our helper like this: return snappy_dynamic_call(ctx, helper_id, fn_id, ctx).

```
int dynfun(void** args) {
  struct file *f; struct dentry *d;
  f = ((struct snappy_ctx*)args)->file_ctx.file;
  d = f->f_path.dentry;
  if(!(f->f_mode & FMODE_WRITE))
    return 0; // No write : OK
  if(! strcmp(d->d_iname, "runc") &&
     ! strcmp(d->d_parent->d_iname, "sbin") &&
     ! strcmp(d->d_parent->d_parent->d_iname, "/")) {
       pr_err("CVE-2019-5736 attempt!\n");
       return -1; // ATTACK
  } else return 0; // OK
}
```

**Listing 4: Dynamic helper mitigating CVE-2019-5736**

As expected, the attack is blocked by our policy. We also raise an alert with pr_err to alert the administrator that the container has tried to attack the system and is therefore compromised. The administrator can take appropriate measures such as destroying or auditing the container. We will show in section evaluation that such a vulnerability mitigation does not significantly affect the performances of our containers (<0.09%) and has zero performance impact over the rest of the system (i.e. the host). Furthermore, such a hotpatching is actually way simpler than the official corrective that implies to copy runC into a memfd and reexecute it from this memfd before entering in the container's namespaces [20].

# 6 EVALUATION

Our testbed is a HP Zbook 15 G5 laptop with a 2.7GHz CPU. Our evaluations are made in a KVM VM with 6 CPUs and 4GB of RAM. We use a 5.5.0 kernel with debugging symbols. In this section, we first present some microbenchmarks measuring the overhead induced by SNAPPY, we then evaluate this overhead in realistic use cases. We finally show how well SNAPPY scales with the number of policies in a realistic scenario.

We show here how policies on file_open impact the latency of the open syscall. We evaluated that adding a level of namespacing only adds ~ 3*ns* of overhead, negligible for most scenarios. Even with 10 levels of namespacing, the latency of this syscall is only affected by ~ 1.5%. Since the depth of the policy_NS is typically below 4, our namespace abstraction does not significantly affect the performances of the system. We also measured that on average, open syscall's latency is only marginally affected by the addition of a policy. Compared to no security, the latency increase generated by a new policy is of about 49ns (2.56%), 64ns (3.36%) and 77ns (3,89%) for a dummy policy using respectively 0, 1 and 2 empty helper(s).

| | Linux compil. | `ls -aRl` |
|---|---|---|
| SNAPPY Disabled | 195.33 ± 0.12 | 3.296 ± 0.006 |
| No SNAPPY policy | 195.57 ± 0.11 | 3.300 ± 0.003 |
| Network connection unicity | 195.54 ± 0.12 | 3.301 ± 0.004 |
| CVE-2019-5736 mitigation | 195.76 ± 0.12 | 3.306 ± 0.003 |

**Table 1: Execution time of commands as a function of the applied SNAPPY policies. Result: Average time (s) ±99% CI**

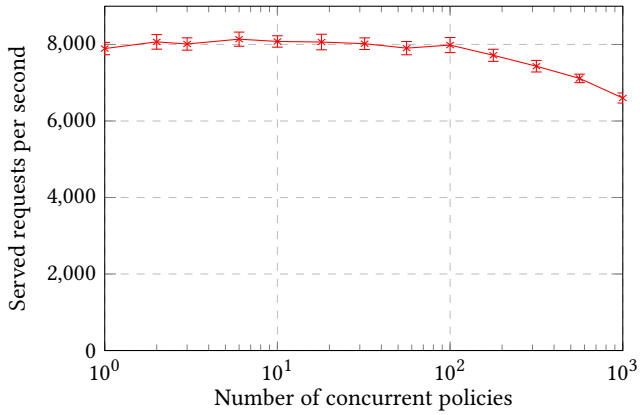


**Figure 6: Throughput of Apache HTTP webserver as a function of the number of SNAPPY policies for this container**

The overhead shown in this microbenchmark is way higher than in realistic scenarios since 100% of the time is spent in the monitored syscall (with no userspace operation) in the monitored namepsace. Hence, it can be seen as a hard upper limit for latency. More insightful data about real overhead can be found in the next benchmarks, typically 1 to 2 orders of magnitude lower.

Table 1 evaluates how policies presented in section 5 affect the system performances compared to a configuration where SNAPPY is entirely disabled and a configuration where no policy is enforced. We use the following scenarios: the compilation of a Linux kernel in `defconfig` mode and the recursive listing of a huge directory with `ls -aRl`. Both benchmarks are executed in a Docker container with both `stdout` and `stderr` disabled to avoid measuring unrelated time. Both of these scenarios open a lot of files but do not realize any remote network accesses. Hence, unsurprisingly, the network-related policy does not add any latency compared to the scenario 'no SNAPPY policy' (aside from statistical noise).

On the other hand, the CVE-2019-5736 mitigation is closer to the worst case scenario since it monitors files' opening. Still, this scenario performs extremely closely to the 'no SNAPPY policy' scenario with only 0.09% ± 0.09% of measured overhead in the kernel compilation scenario and 0.03%(±0.02%) for the `ls -aRl` scenario. This demonstrates that the usage of SNAPPY does not significantly impact performances and is therefore very adapted for containers whose performance is one of their main advantages.

We evaluate SNAPPY's macro performance in figure 6 by measuring the throughput of a containerized Apache2 webserver as a function of the number of SNAPPY policies applied to this specific container. This benchmark is done with `ab` using 100 concurrent requests and uses the policy to mitigate the `runC` vulnerability, presented in 5.2. We show that with a reasonable number of policies there is a minimal overhead. Indeed, with 100 policies, the overhead is not measurable under our 99% CI. With 1,000 policies we measure a 16.4% slowdown (0.0164% per policy). In realistic scenarios, the number of policies per container is expected to remain reasonable (typically under 100) and a huge part of these policies are applied to way less frequently executed hooks than `file_open`. Hence, the performance impact of SNAPPY is very low for realistic use cases.

## 7 RELATED ART

| | SNAPPY | Landlock | Secu. NS. | Seccomp | BPFBox |
|---|---|---|---|---|---|
| Programmable | ● | ● | ○ | ◐ | ● |
| Fine-Grained | ● | ◐ | ◐ | ○ | ◐ |
| Generic | ● | ◐ | ○ | ○ | ● |
| By namespace | ● | ○ | ● | ○ | ○ |
| Unpriv. Usage | ● | ● | ○ | ● | ○ |

**Table 2: Comparison between SNAPPY and the related art**

**Landlock LSM** [33] is a framework aiming to let non-administrative processes such as containers finely sandbox themselves at kernel-level to limit their own rights thus mitigating potential compromises. Like SNAPPY, Landlock implements this by allowing eBPF policies to be pushed from any process to the kernel. Landlock relies on a new dedicated syscall (`landlock`). Similarly to SNAPPY, these policies are mandatory, stackable, and can be applied to only a subset of the system. For instance, this allows to apply MAC policies to define whitelists for read-write files and for read-only files. All other accesses are denied by such a policy. However, contrarily to SNAPPY, the design of Landlock is not generic and only handles a handful of hooks, related to files, credentials and ptracing. Additionally, there is no concept of namespace in Landlock LSM. Instead, policies are applied to a process and its offspring. Thus, Landlock LSM cannot apply the same policies to processes having no parentage link. Containers spawned through a system daemon will not inherit the policies of the process that spawned it, resulting in a risk of privilege escalation.

We presented **Security Namespace** [38] and its limitations in section 2.2. We show here that these limitations are not encountered with SNAPPY. First, although both frameworks allow to execute namespaced policies, because SNAPPY can execute any valid eBPF code and make use of dynamic helper calls it is way more generic than the adaptation of a *single* LSM in the system, as done in Security Namespace. Second, because SNAPPY is based on unmodified established primitives (eBPF, kernel objects, LSM, ...), the correctness of its implementation is easier checked than the manual adaptation of LSM modules, a complex and bug-prone task. Third, the performance overhead of SNAPPY scales better than Security Namespace since it just requires to check the currently active `policy_NS` and its ancestors (typically less than 3) and not *every* namespace, a huge number in big servers, as in Security Namespace. For these reasons, we believe that our design is a better fit for unprivileged and untrusted containers' protection than with Security Namespace.

Other approaches leverage eBPF to improve security at system level. But none of these approaches solve the challenges earlier mentioned in this paper. **BPFBox** [7] allows to enforce eBPF policies in the kernel to improve process' isolation. Yet, BPFBox is not focused on containers and its policies must be enforced by the administrator.

**Cilium** [16] leverages eBPF to secure container's communications. It can be used as a layer 7 firewall, for instance to allow only certain commands to be called from an API. Yet, it remains very specific to network thus can not be used to protect containers themselves. **KRSI** [36] can be used by the administrator to apply eBPF policies to LSM hooks to detect symptoms of potential attacks (e.g. a process uses `LD_PRELOAD`). Yet, such policies are to be applied to the whole system (i.e. not to a single container) and KRSI cannot be used by an unprivileged entity. **Falco** [32] is a syscall tracer aiming to detect threats. It can rely on a kernel module or on eBPF. Still, Falco does not provide a namesapcing mechanism thus does not allow per-namespace policies. Additionally, syscall interception-based approaches are very dependent on the kernel version and do not provide an unified way to map security behaviors thus are arguably less reliable than LSM-based approaches. **Seccomp-BPF** [10] allows to do syscall filtering using BPF (not eBPF). Although it can limit the attack surface of processes, it cannot dereference kernel pointers thus remains very coarse-grained.

A synthetic feature's comparison between SNAPPY and its closest related art can be found in table 2.

## 8 DISCUSSION AND FUTURE WORKS

Our framework allows to enforce at runtime stackable, programmable and fine-grained policies for processes in namespaces. The current limitations still to be improved in future works are:

- As we trust the administrator, the coherence of our helpers is currently not checked (i.e. no safeguards) ;
- A helper should not call the hook which called it in order to avoid recursions in the kernel. Fortunately, if helpers are used properly (i.e. only relies on internal kernel functions and never uses "userspace wrappers", this should not happen). Yet, we are investigating the possibility to solve this issue by blocking the trigger of LSM hooks when already in a SNAPPY hook.

In order to show more in depth the interest of SNAPPY and its ease of integration, a future line of research is to develop policies to protect real-life containers.

Another line of research that we are currently investigating is to transform existing LSM modules into SNAPPY library so they can be used by unprivileged containers transparently and independently from the rest of the system. That would allow to deport the LSM namespacing into SNAPPY, thus avoiding current LSMs stacking and namespacing issues. At the time of writing, our findings show that this could be done semi-automatically with only few code modifications (adapt interfaces, avoid unexported symbols, store data independently from the `lsmblob`...)

In conclusion, in this paper, we presented SNAPPY, a new framework that allows even unprivileged processes such as containers to safely enforce in the kernel stackable, programmable and fine-grained policies at runtime. This enables to minimize their attack surface thus increase their security level. We discussed the design, the implementation and the integration with the OCI specification and containerization engines of this framework. We finally shown that this framework only incurs a tiny performance overhead and is therefore adapted to containers.

## REFERENCES


[1] Pratyush Anand. 2017. A presentation of eBPF. https://opensource.com/article/17/9/intro-ebpf.
[2] Mick Bauer. 2006. Paranoid Penguin: An Introduction to Novell AppArmor. *Linux J.* (2006).
[3] Maxime Bélair, Sylvie Laniepce, and Jean-Marc Menaud. 2019. Leveraging Kernel Security Mechanisms to Improve Container Security: A Survey. In *Proceedings of the 14th International Conference on Availability, Reliability and Security (ARES '19)*.
[4] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. 2017. Hardware and software support for virtualization. *Synthesis Lectures on Computer Architecture* (2017).
[5] Docker. 2019. Dockerfile reference. docs.docker.com/engine/reference/builder.
[6] W. Felter, A. Ferreira, R. Rajamony, et al. 2014. An Updated Performance Comparison of Virtual Machines and Linux Containers. *technology* (2014).
[7] William Findlay, Anil Somayaji, and David Barrera. 2020. Bpfbox: Simple Precise Process Confinement with EBPF. In *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop (CCSW'20)*. 91–103.
[8] Free Software Foundation. 2019. Chroot man page (2). http://man7.org/linux/man-pages/man2/chroot.2.html.
[9] Jess Frazelle. 2018. Containers aka crazy user space fun. In *linux.conf.au 2018*.
[10] freedesktop.org. 2017. Presentation of Seccomp BPF. https://dri.freedesktop.org/docs/drm/userspace-api/seccomp_filter.html.
[11] Nick Frichette. 2019. PoC for CVE-2019-5736-PoC. https://github.com/Frichetten/CVE-2019-5736-PoC.
[12] D. Gannon, R. Barga, and N. Sundaresan. 2017. Cloud-Native Applications. *IEEE Cloud Computing* 4, 5 (2017), 16–21.
[13] Google. 2020. Kubernetes repository. https://github.com/kubernetes/kubernetes.
[14] Thomas Graf. 2020. eBPF - Rethinking the Linux Kernel. In *QCon London 2020*.
[15] Alexander Holbreich. 2018. Docker components explained. http://alexander.holbreich.org/docker-components-explained/.
[16] Isovalent Inc. 2020. Cilium GitHub repository. https://github.com/cilium/cilium.
[17] Open Containers Initiative. 2020. Open Container Initiative Runtime Specification. https://github.com/opencontainers/runtime-spec.
[18] Open Containers Initiative. 2020. runC GitHub repository. https://github.com/opencontainers/runc.
[19] Cynthia E Irvine. 1999. *The reference monitor concept as a unifying principle in computer security education.* Technical Report.
[20] Adam Iwaniuk and Borys Popławski. 2019. CVE-2019-5736: Escape from Docker and Kubernetes containers to root on host. https://blog.dragonsector.pl/2019/02/cve-2019-5736-escape-from-docker-and.html.
[21] Jhon Johansen. 2018. Making Linux Security Modules available to Containers: Stacking and Namespacing the LSM. In *Proceeding of the Free and Open Source software Developers' European Meeting (FOSDEM '18)*. Brussels.
[22] kernel.org. 2020. Linux Virtual Memory Mapping. https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt.
[23] Linux Manual. 2020. namespaces - overview of Linux namespaces. https://www.man7.org/linux/man-pages/man7/namespaces.7.html.
[24] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture.. In *USENIX winter*, Vol. 46.
[25] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article 2 (March 2014).
[26] NIST. 2017. CVE-2017-16995. https://nvd.nist.gov/vuln/detail/CVE-2017-16995.
[27] NIST. 2019. CVE-2019-5736. https://nvd.nist.gov/vuln/detail/CVE-2019-5736.
[28] NIST. 2020. CVE-2020-8835. https://nvd.nist.gov/vuln/detail/CVE-2020-8835.
[29] Claus Pahl, Antonio Brogi, Jacopo Soldani, et al. 2017. Cloud Container Technologies: a State-of-the-Art Review. *IEEE Transactions on Cloud Computing* (2017).
[30] Rusty Russell. 2020. iptables Repository. http://git.netfilter.org/iptables/.
[31] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, et al. 2004. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13 (SSYM'04)*. 16–16.
[32] Jorge Salamero. 2019. Kubernetes Runtime Security with Falco and Sysdig. https://www.cncf.io/wp-content/uploads/2019/12/Kubernetes-Runtime-Security-with-Falco-and-Sysdig.pdf.
[33] Mickaël Salaun. 2018. File access-control per container with Landlock. In *Free and Open Source Software Developer (FOSDEM '18)*. Brussels.
[34] Ravi Sandhu. 2013. Access Control Models. profsandhu.com/cs6393_s13/L2.pdf.
[35] Casey Schaufler. 2019. LSM Stacking - What You Can Do Now and What's Next. In *Linux Security Summit Europe (LSS'19)*.
[36] KP Singh. 2019. Kernel Runtime Security Instrumentation. In *Linux Security Summit North America 2019*.
[37] Stephen Smalley, Chris Vance, and Wayne Salamon. 2001. Implementing SELinux as a Linux security module. *NAI Labs Report* 1 (2001), 43.
[38] Yuqiong Sun, David Safford, Mimi Zohar, et al. 2018. Security Namespace: Making Linux Security Frameworks Available to Containers. In *Proceedings of the 27th USENIX Conference on Security Symposium (SEC'18)*. 1423–1439.
[39] C. Wright, C. Cowan, et al. 2002. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of the 11th USENIX Security Symposium*.