Losing the Beat: Understanding and Mitigating Desynchronization Risks in Container Isolation

Zhi Li*†, Zhen Xu*†, Weijie Liu§, XiaoFeng Wang¶, Hai Jin*‡, and Zheli Liu§
*National Engineering Research Center for Big Data Technology and System,
Services Computing Technology and System Lab, Huazhong University of Science and Technology (HUST)
†Hubei Engineering Research Center on Big Data Security, Hubei Key Laboratory of Distributed System Security,
School of Cyber Science and Engineering, HUST
‡Cluster and Grid Computing Lab, School of Computer Science and Technology, HUST
§College of Cryptology and Cyber Science, Nankai University
¶Nanyang Technological University
{lizhi16, zhenxu, hjin}@hust.edu.cn, {weijieliu, liuzheli}@nankai.edu.cn, xiaofeng.wang@ntu.edu.sg

***Abstract*—The isolation offered by containers today is achieved through leveraging Linux namespaces and cgroups in a highly coordinated way. This foundation for container protection, however, has been shaken by the evolution of computing paradigms, particularly the emergence of serverless computing with strong demands for resource sharing across namespaces. Such sharing weakens the container's isolation model, inducing *namespace-cgroup desynchronization* (NCD) vulnerabilities, as discovered in our research. In this paper, we present a study on such risks, aiming at identifying their root causes and understanding their implications. Our research reveals that popular container tools all suffer from NCD risks, as evidenced by our discovery of four new vulnerabilities and one bug. Fundamentally, namespace sharing expands a container's isolation boundary, which may contravene the restrictions set by the cgroups, thereby undermining the combined protection provided by both mechanisms. This contention often cannot be reconciled by existing container tools.**

To address this challenge and meet the demands for namespace sharing, we propose a kernel-level solution to unify the fragmented responsibilities of namespaces and cgroups in monitoring the resources for container instances. Our design bonds the resource management handled by namespaces with the resource restrictions enforced by cgroups, and identifies the collaborative policies that they should follow. The analysis and evaluation demonstrate that our approach effectively mitigates the NCD risks, as well as incurs a negligible cost to the Linux kernel, mainstream container tools, and real-world applications, maintaining full compatibility with these systems.

I. Introduction

Linux container technology promises a lightweight isolated execution environment for applications [1], which is constructed by piecing together multiple kernel mechanisms on demand, such as *namespace* and *cgroup*. As this technology is widely used for high-productive software development and deployment, the isolation guarantees provided by containers have garnered continuous concern [2], [3], [4], [5], [6], [7], [8], [9]. More fractional kernel mechanisms are involved to isolate sensitive resources from other containers and the host [10], [11], [12]. These patchworks operate independently, yet provide complementary functionalities to prevent processes within a container from accessing the resources outside.

However, this isolation model has been continuously weakened with the emergence of new software architecture (e.g., microservice [13]) and cloud computing mode (e.g., serverless [14]), which increasingly require namespace sharing between containers or between the container and the host. Microservice architectures decouple the traditional monolithic applications into independent, self-deployable services that communicate via APIs to deliver the same functionality [15]. Each service can be a stateless function with simpler logic than a microservice. The flexible assembly of these functions can achieve richer call chains at a lower cost than deploying monolithic software [16]. For example, shared memory is widely used to decrease the latency of data sharing between services in serverless computing [17]. And namespace sharing becomes an ideal solution to improve data transmission and status synchronization [18], to simplify deployment and configuration [19], and to facilitate debugging and monitoring [20]. Unfortunately, sharing namespaces weakens isolation guarantee offered by other kernel mechanisms (e.g., cgroups), which conflicts with the original intention of the container design [21].

Understanding namespace sharing risks. In our research, we discovered *namespace-cgroup desynchronization* (NCD) – a new security risk introduced by namespace sharing across containers or between a container and its host. We found that the namespace sharing extends the isolation boundary beyond one container instance, yet the protection provided by cgroups still stays within the instance. More specifically, the resources created by a container instance are managed within its namespaces, while control of these resources, particularly the restrictions on the consumption of the resources, is handled by the cgroup. When the container is terminated, its cgroup will be destroyed as well, but its namespace, once shared with another container or the host, will not be released. As a result,

*Corresponding authors: Weijie Liu and Zhen Xu.

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.231381
www.ndss-symposium.org

the resources associated with the namespace are no longer under expected control, which is supposed to be enforced by the cgroup. This opens the door for the adversary to exploit the shared namespaces to gain unrestricted access to resources, leading to a cgroup disruption attack. As an example, we discovered that a pair of colluding containers sharing an IPC namespace can collectively abuse memory resources beyond their pre-allocated limits on a commercial platform without incurring additional charges. One container asks for a large amount of IPC resources and then promptly terminates itself, while its cgroup for keeping track of the allocated resources is removed; the other container can continue using those resources without being traced by any cgroup.

All popular container tools, including the orchestration platform (e.g., Kubernetes [22]) and management engines (e.g., Docker [23], Podman [24], and Pouch [25]) support sharing of IPC, network, PID, UTS, and user namespaces across containers or between a container and its host. To understand the impacts of the NCD risk we discovered, we built a detector to analyze these tools' susceptibility to the NCD risks. Running our detector on those container tools, we found that all of them are exposed to NCD risks, with unignorable security implications. Moreover, our measurement study shows that projects on GitHub are using those container tools to deploy their applications, and many of them are configured to share some types of namespaces, especially in the form of Kubernetes pod [26] where containers share the IPC, network, UTS, and user namespaces by default. As a prominent example, most deep-learning, load balancing, and high-performance computing services require containers to share the IPC namespace.

We reported our findings to the Docker, Podman, Pouch, and Kubernetes communities, all of which confirmed the seriousness of the NCD risks. Four new vulnerabilities (CVE-2024-3056, CVE-2024-55528, CVE-2024-57688, and CVE-2024-57689) and one bug have been discovered [27]. Even though the Docker community has verified the problem, they claim that it should be addressed by the providers of the orchestration platforms (such as Kubernetes) rather than itself. Also, the Docker community acknowledges that the problem cannot be easily fixed, requiring nontrivial changes to the Docker engine. The Red Hat community, which maintains the Podman project, attributes the problem to the Linux kernel and considers it a high-severity vulnerability (CVE-2024-3056) [28].

Strawman solutions. A straightforward solution is to create an "oversight" cgroup at user space for the containers that share namespaces, to keep an account of and impose restrictions on the total resources consumed by all the processes in these containers. In this way, even when a container's cgroup is destroyed, the total resources created in these containers are still accounted for by the oversight cgroup. This approach is available for public cloud environments where namespaces are shared between containers only. It, however, cannot be applied to private platform scenarios where the container is allowed to share its namespaces with its host: otherwise, all processes in the shared namespaces including host processes like system services will be put under the resource restrictions set by the cgroup, which undermines the host process's functionality. Therefore, a more fundamental solution is needed for private platform scenarios.

Similar to the Podman community, we also hold the belief that the NCD risks should be addressed by the kernel, which is best positioned to coordinate the joint operations of namespaces and cgroups to ensure proper control over shared resources is always in place. A unique challenge here is the independent operation of namespaces and cgroups in the kernel, making their reliable and efficient coordination difficult. More specifically, a namespace and a cgroup manage resources at different granularities: the namespace handles *virtual resources* like semaphores, while the cgroup keeps track of and caps the consumption of the *system resources* (e.g., memory pages) assigned to the virtual resources. It is less clear how to map the system resources to their virtual counterparts, especially when releasing those already losing required control. We mark the virtual resources left within a namespace after a container exits as *residual resources*. How to securely manage such residual virtual resources should be well thought out to avoid any significant impact on the usability of container-based applications.

Our solution. We came up with the first kernel-side defense called `CANs` (*Cgroup Associating with Namespaces*), which is carefully designed to ensure that the expected cgroup control remains in place on the resources within shared namespaces, without incurring any significant performance or usability impact. `CANs` tags the virtual resources to identify the responsible cgroup. It also maintains transparent oversight cgroups to enforce the restrictions on the residual resources. Note that this can only be done in the kernel space since in the userland, a container tool does not have a direct observation of the relationship between the resource usage in a cgroup and the corresponding resources maintained in a shared namespace. Therefore, our approach follows a set of carefully designed policies to build the relationship and properly manage these residual resources, while maintaining functionalities of existing namespaces and cgroups.

We conducted a security analysis on our design, which demonstrates its capability to eliminate the NCD risks and shows that it does not introduce new attack surfaces. We further evaluated our implementation on Docker, Podman, Pouch, and Kubernetes. Our study reveals that `CANs` incur less than 0.5% average overhead compared to native Linux, a maximum overhead of just below 2.4% for these container tools, and an average overhead of merely 3.6% for popular real-world applications running with containers (Section V-B). Also, `CANs` is fully compatible with all mainstream container tools and completely transparent: it does not affect any operations of these tools and the applications.

Contributions. Our contributions are outlined below:

• *New findings and understanding*. We report the discovery of the NCD risks – a previously unknown threat, and their

impacts. We demonstrate that these risks are fundamentally caused by the expanding gap between the container's isolation design and the growing demands for more aggressive resource sharing, which renders namespaces and cgroups – two building blocks of container isolation, no longer working effectively together. Further, we responsibly disclose our findings to affected parties.

• *New defense*. We developed a kernel-side protection to enhance container isolation, addressing the technical challenges in efficiently eliminating NCD risks while maintaining compatibility and usability. Our approach was evaluated against these risks in all mainstream container tools, demonstrating that it only incurs negligible performance overhead. We have released our code [29] and presented our solution as a proposal for the Linux kernel, which has received positive responses from the downstream Red Hat Linux [30].

II. Background

A. Linux Namespaces

Linux kernel provides various types of namespaces to isolate kernel resources between groups of processes [31]. Each namespace can be instantiated multiple times to serve different groups of processes, with each instance independently wrapping and managing its isolated resources. Linux containers leverage namespaces to separate containers' *virtual resources*, defined as kernel-level resources isolated by namespaces from the host system. Specifically, the process-related resources (e.g., process IDs and inter-process communication) are isolated by PID and IPC namespaces. Mount namespace is responsible for separating the management of mount points between the host and containers. The combination of network and UTS namespaces provides an independent management view of the network-related resources (e.g., virtual network devices) and hostname/domain name for containers. In containers, most security-related identifiers (e.g., user/group IDs, keys, etc.) are isolated by user namespace.

All types of namespaces support multiple instances to independently maintain and manage isolated resources. For example, each instance of the mount namespace has an independent tree formed by mount points, distinct from other mount namespaces, dedicated solely to regulating the mount points within that namespace. Moreover, all resources governed by a namespace instance are forcibly released upon the destruction of that namespace. However, if the namespace instance remains active, the resources it manages will not be released, even if the process that created it exits. Typically, the namespace instance is torn down automatically when the last process within it terminates or leaves [32].

B. Namespace Sharing

Container technology promotes software architecture practices such as microservices and serverless computing. Traditional monolithic cloud applications struggle with efficient development, updates, and maintenance due to their tightly coupled components. Microservices architectures address these issues by decoupling applications into independent, self-deployable services that communicate via APIs to deliver the same functionality [15]. Moreover, serverless computing, also known as Function as a Service (FaaS), supports the deployment of function-level services [16]. Each service can be a stateless function with simpler logic than a microservice. The flexible assembly of these functions can achieve richer call chains at a lower cost than deploying monolithic software.

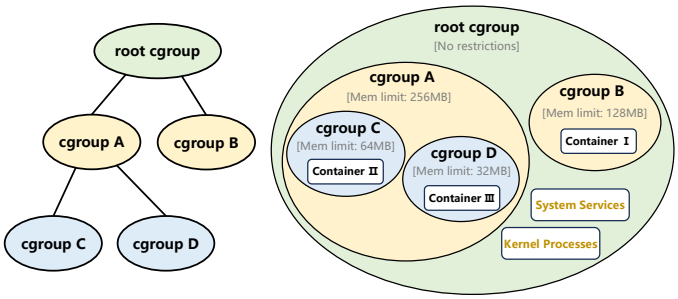


Fig. 1: Cgroup hierarchy

The increasingly fine-grained splitting of cloud applications makes their development and deployment more flexible, scalable, and easier to maintain. However, this comes at the cost of performance loss due to data sharing or interaction between separate services [21]. The need for resource sharing has emerged within the isolated environment of these services to enhance their intercommunication and expedite data transfer. Container tools such as Docker, Podman, Pouch, and Kubernetes all provide interfaces for sharing namespaces between containers to achieve isolated resource sharing. Especially, the default configuration of containers within a Kubernetes pod involves sharing the IPC, network, UTS, and user namespaces. Namespace sharing meets the functionality and performance requirements of modern cloud services. More real-world cases are discussed in Section III-C.

C. Cgroup Architecture

In the Linux kernel, cgroup implementation relies on different resource controllers (or subsystems) to account for and limit the consumption of various types of *system resources*, including CPU time, memory, block I/O, network bandwidth, etc. Linux containers leverage the cgroup to impose resource constraints on each container and prevent a single container from draining host resources. For the billing model in cloud computing, cgroup maintains the resource accounting per container and can be employed to constrain the usage of system resources allocated to each container.

Cgroup hierarchy and controllers. In Linux, cgroups are structured as a hierarchical tree (as shown in Fig. 1), with each node representing a cgroup that tracks and limits the resources used by the processes within it. A process can be a member of only one cgroup per hierarchy, but its resource usage is also restricted by all parent (ancestor) cgroups; consequently, even if a process is to bypass the limitations of the cgroup to which it belongs, it would still be subject

to the resource constraints imposed by the ancestor cgroups. It should be noted that the root cgroup, serving as the top-level node to monitor system-wide resources, cannot have any restriction imposed on it. Since all system services and kernel processes are associated with the root cgroup, any limitations applied at this level could unintentionally restrict essential system functions, potentially resulting in degraded performance or system instability. Moreover, the container runtime (e.g., runc [33]) creates and configures one cgroup for each container. All processes in the container join and are restricted by that cgroup. By default, the parent of these cgroups is the root cgroup, which typically does not impose any limitations.

Controllers attached to cgroups are responsible for resource accounting and restriction in practice, which are implemented by hooking into various resource management units within the kernel. For instance, the memory controller hooks the kernel functions `__alloc_pages` and `slab_alloc_node` called in the `kmalloc` function, which are parts of the memory management unit. When a process allocates memory, these hooks tag the allocated memory pages and slabs using the associated cgroup as the label and count them into that cgroup. Also, a check is performed in these hooks before allocation. This check would block the allocations if they result in the cgroup's memory accounting exceeding the set limit. Note that each cgroup only maintains the accounting of the allocated system resources reported by the controllers, neither managing them nor knowing which virtual resources occupy them. Correspondingly, the kernel function `kfree`, used to free allocated memory, is also hooked. These hooks trace the tags on memory pages or slabs to identify their associated cgroup and then subtract the corresponding accounting after freeing them. As a result, resource accounting in the cgroup is updated only upon resource allocation and release, not during resource access.

Different controllers implement specialized mechanisms for imposing constraints on their associated resources.

Capacity-based resource control. Controllers like PID and memory primarily restrict the quantity of system resources that processes within the cgroup can allocate and retain. Specifically, the PID controller imposes a quantitative limit on the number of process IDs (PIDs) within a cgroup to prevent unlimited process creation. Similarly, the memory controller enforces a capacity limit on the total memory that can be consumed by processes in a cgroup. Once the allocated system resources reach the cgroup limits, processes in that cgroup will be unable to acquire more of them; otherwise, a fault, such as an out-of-memory error, will occur.

Time-based resource control. In contrast to PID and memory controllers, which primarily aim to limit resource quantity, CPU and blkio controllers focus on regulating access to continuously available resources rather than preventing their depletion. For example, CPU resources are controlled by assigning distinct weights or quotas to each cgroup, thus determining the relative share of CPU time available to processes within different cgroups. If the allocated shares are depleted, processes within the cgroup will be temporarily throttled until the next CPU allocation cycle. Similarly, the blkio controller configures I/O prioritization to regulate the I/O bandwidth (e.g., the number of read/write operations or total bytes per second) of processes in each cgroup. In this paper, we concentrate on exploring the intricacies of capacity-based resource management.

D. Threat Model

We consider both private and public container-based platforms that use popular tools such as Docker, Podman, Pouch, and Kubernetes to manage containers. Container instances belonging to the same user or organization can share the same host. The administrators employ cgroups to set the resource limit for each container, preventing host resource exhaustion.

Public container services let users create container instances priced by pre-allocated resources (vCPU and memory). These instances of the same users can share namespaces. We assume an attacker controls at least two namespace-sharing containers and exploits NCD vulnerabilities to disrupt their cgroups, thereby consuming unpaid host resources. Since public platforms do not allow namespace sharing with the host, administrators can thwart attackers by implementing an oversight cgroup for containers that share namespaces (see Section I).

On private platforms, owners can create containers from any image (self-built or public registry), and they have the choice to configure the container to share namespaces with other containers or with the host. Attackers, whether internal or external to the private platform, possess the capability to craft malicious images with the self-activating cgroup disruption attack scripts, designed to perform a DoS attack and disrupt the quality of service on the host. The malicious images can be spread through public registries (e.g., DockerHub) or via the supply chain, often accompanied by READMEs that recommend namespace-sharing settings for deployment. It only requires the users of the private platform to follow these instructions and launch a single container with the malicious image, then this action could exhaust the host's resources.

III. Understanding Namespace-Cgroup Desynchronization Risks

In this section, we first clarify the potential security risks posed by namespace sharing and propose a detection tool for their systematic detection. We then report the *Namespace-cgroup Desynchronization* (NCD) vulnerabilities to all impacted container tools. Subsequently, we dive into the impact of these risks and measure the prevalence of shared namespaces in real-world applications.

A. NCD Risks

The most prevalent method of resource sharing between multiple containers or between a container and the host is through namespace sharing. However, the isolation boundary established by the shared namespace is no longer for the single container instance. The potential cooperation between

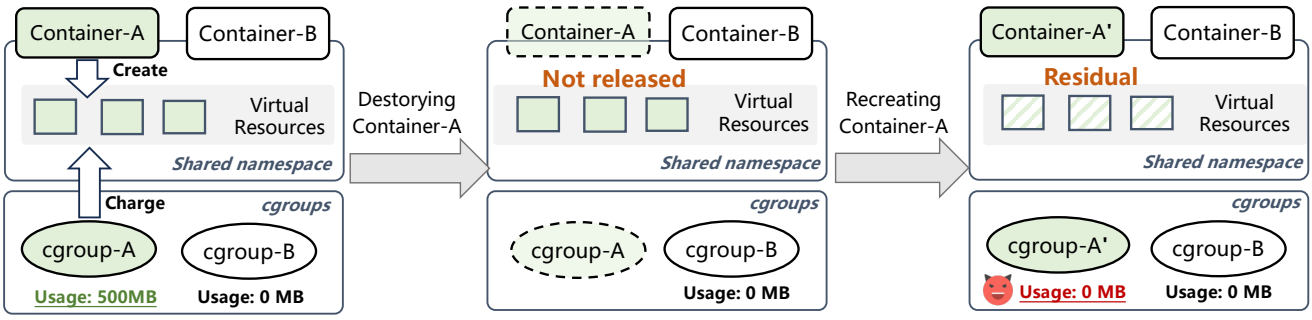


Fig. 2: Cgroup disruption attacks

the shared namespace and cgroups working for a single container instance could be compromised. Specifically, namespace sharing disrupts the uniformity of the lifecycles between the namespace and cgroups, potentially extending the lifecycle of the shared namespace beyond that of the container instance and its cgroup.

For an individual container, the creation and destruction of its namespace and cgroup are synchronous. Upon the container's termination, the resources allocated within this container are released following the destruction of the related namespace, while the accounting of those resources is also canceled as the container's cgroup is destroyed. However, namespace sharing disrupts this synchrony. When a container shares its namespace with other entities (containers or the host), the termination of this container does not destroy the shared namespace, which continues to be sustained by those entities. Additionally, the resources created by the terminated container are also maintained within this namespace, but these resources are no longer accounted for and restricted by that container's cgroup, which has been destroyed with the container. *This disorder between the management and constraint of resources would weaken or undermine the isolation assured by the container.*

Cgroup disruption attacks. By exploiting NCD risks, an attacker can manipulate the containers sharing namespaces to disrupt their cgroups' supervision. Specifically, as shown in Fig. 2, the attacker controls two containers (`Container-A` and `Container-B`) configured to namespace sharing, while each container is governed by its own exclusive cgroup. During the attack, `Container-A` is manipulated by the attacker to continuously generate resources that are managed by the shared namespace. Until reaching cgroup limits of `Container-A`, the attacker could manually utilize interfaces provided by container tools or public container services (e.g., Kubernetes feature of rolling update [34]) to destroy and recreate this container. More commonly, `Container-A` is configured by these tools or services to restart automatically when it exceeds the cgroup limits. Meanwhile, `Container-B` remains active to keep the shared namespace persistent. This ensures that those resources generated by `Container-A` are not released upon the destruction of `Container-A` and its cgroup, as the shared namespace continues to maintain them.

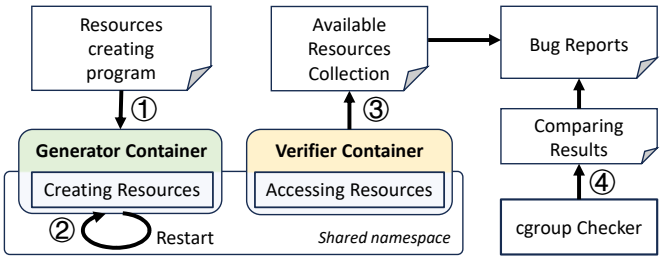


Fig. 3: Detector workflow

As a result, the resources created by `Container-A` become residual resources, no longer traced by its cgroup or any other container's cgroup. Other containers (e.g., `Container-B` and the restarted `Container-A'`) sharing that namespace can still access those resources, but their cgroups do not monitor or account for such access.

B. Vulnerability Discovery

All container tools provide interfaces to configure namespace sharing. We propose a detector to assess whether these configurations are vulnerable to NCD, and identify exploitable virtual resources managed within each namespace for resource depletion or billing evasion attacks.

Detection method. Fig. 3 shows the pipeline architecture of the detector. At the preparation stage, the tested container tool first creates two containers with a specified namespace sharing. In each detection round, one of the containers acts as a generator, creating a specific type of virtual resource (①) mentioned in Table I and then restarting (②). Another container, acting as a verifier, accesses the created virtual resource to confirm its availability (③). Moreover, the cgroup checker records the total resource accounting from both containers' cgroups at the end of the above stages (①, ②, and ③). Once the verifier-container can obtain that virtual resource, the cgroup checker will compare the recorded resource accountings at different stages (④). If the resource accounting at stage ① falls back to zero at stage ② and remains zero at stage ③, it means that such virtual resource is not accounted for by any cgroup after the generator-container restarts, even after being accessed again by the verifier-container. In this case, the detector can confirm that the tested container tool is vulnerable, and issue

TABLE I: Detection setting and results

Shareable Namespace	Namespace-related Resources	Create Operations	Exploitable for NCD risks	Vulnerable Tools
IPC	shared memory	shmget	✔	Kubernetes Docker Podman Pouch
	semaphore	semget	✔	
	message queue	msgget	✔	
Network	virtual network devices	ip link add	✔	
PID	process ID	fork	✔	Docker Podman Pouch
UTS	hostname/domainname	N/A	×	N/A
User	keys	add_key	×	N/A

a bug report to document the exploitable virtual resource in the namespace.

Effectiveness and findings. We detect the widely-used container tools, including Docker, Podman, Pouch, and Kubernetes, and confirm that they all suffer from NCD risks. More details are shown in Table I. Specifically, containers configured to share the IPC namespace can abuse *shared memory*, *semaphore*, and *message queue* in attacks to exhaust or even abuse the host's memory resources. Similarly, containers that share the network namespace can exploit 39 types of *virtual network devices* [35] in attacks. Moreover, containers launched by Docker, Podman, or Pouch with shared PID namespaces can exhaust the host's process IDs by spawning many zombie processes that are not reaped. Lastly, the virtual resources managed within the UTS namespace only occupy a fixed amount of memory resources to store the hostname and domain name, making them not exploitable for gaining more resources illegally. The virtual resource *keys* in the keyrings [36] related to the user namespace is ignored in the detection since cgroups do not monitor or account for this resource [37]. Any container can allocate it without restriction to directly cause a DoS on the host, which is beyond the scope of our study. Note that no container tools support sharing the mount namespace between containers and the host.

Responsible disclosure. We reported our findings to Docker, Podman, Pouch, and Kubernetes, who have all determined the existence of the NCD risks. Four new vulnerabilities (CVE-2024-3056, CVE-2024-55528, CVE-2024-57688, and CVE-2024-57689) and one bug have been discovered [27]. The Docker community believes that the fix is the responsibility of orchestration platforms, as Docker does not track container dependencies. The Podman, Pouch, and Kubernetes communities confirmed that NCD risks arise from resource management vulnerabilities or bugs when containers share namespaces. Furthermore, the Red Hat community, to which the Podman project belongs, believes that the root of these security issues is traced to the Linux kernel [28], as cgroups fail to handle the resources they are intended to restrict. This issue has been treated as a high-severity vulnerability (CVE-2024-3056).

C. Real-World Impact Measurement

Unfortunately, namespace sharing is a widespread demand for many practical applications, which undeniably exacerbates the NCD risks in containerized environments. We measured containerized applications on GitHub to quantify the scale of namespace-sharing requirements, and also assessed the real-world feasibility of the cgroup disruption attacks.

Real-world examples of namespace sharing. We searched all GitHub repositories using GitHub APIs [38] and filtered them by keywords in the README files or YAML files in these repositories. If the README files contain keywords (e.g., 'docker run' or 'podman run'), it indicates that the applications in those repositories can be deployed using popular container tools. Repositories that contain Kubernetes-specific YAML files [39] are also search targets, which can be identified by keywords in fields such as 'apiVersion' and 'kind'. Moreover, all container tools provide different configuration parameters to set a container to share the namespaces with other containers (e.g., `--ipc=container:[id]` and `--network=container:[id]` for IPC and network namespace respectively) or with the host (e.g., `--ipc=host` and `--network=host`). We retrieved the parameters displayed after those matched keywords in the README files and the Kubernetes-specific YAML files. We collected 824,353 GitHub repositories whose applications are able to be containerized. Scanning the configuration parameters of containers from those 824,353 repositories, 128,935 repositories (15.64%) are flagged as requiring their applications' container(s) to run in namespace sharing mode.

We manually analyzed 541 flagged repositories with over 200 stars. Most of these require containers to share either the IPC or network namespace, accounting for 10.10% and 86.14%, respectively. Their primary purpose is to fulfill the functional and performance requirements of applications. 88.68% of the applications that require IPC namespace sharing are associated with deep learning, stemming from their functional requirements. These applications utilize the NVIDIA Multi-Process Service (MPS) runtime architecture [40]; consequently, sharing the IPC namespace with the host is essential for their containers to communicate with the MPS control daemon [41]. IPC namespace sharing for performance is demanded by such as high-performance computing applications to reduce communication latency between containers. 75.54% of network namespace sharing aims to simplify network configuration for containers, a necessity for applications that involve constructing a Virtual Private Network (VPN) or facilitating service discovery and load balancing within a microservices framework. The remaining 24.46% of applications

recommend that their containers share the network namespace with the host for better networking performance, as this allows the containers to utilize the host's native networking stack directly [19]. The requirements for sharing the PID namespace all originate from the monitoring tools for watching the status of processes from multiple containers.

Feasibility analysis of the real-world attacks. Cgroup disruption attacks can produce varying impacts on private platforms and public cloud services (see Section II-D). We assessed the feasibility of achieving these impacts in the real world.

Resource depletion is a common attack that can be performed after disrupting the cgroup on both private and public container-based platforms. We validated its feasibility in a testbed environment, and made its configurations along with attack demonstration videos publicly accessible [42]. To conduct the attack, we built a malicious image that embeds an attack script as its entrypoint. This script activates automatically at container startup, continuously allocating the designated resources (mentioned in Section III-B). We deployed this malicious image and set its parameters to share namespaces with another container (`--ipc=[container]`) and to auto-restart upon exit (`--restart=always`). As a result, the attack proceeds autonomously and depletes the host's 8 GB of memory resources in approximately 23 seconds.

Billing evasion is another feasible attack on public container services. Due to ethical concerns, we refrained from executing this attack on real-world services. Instead, we verified its preconditions by checking the configurations of container instances and their host in these services, all obtained via *Procfs* and *Sysfs* from inside the containers [4]. Specifically, on AWS and Baidu Cloud container services, we deployed two container instances that we could restart or destroy via the providers' APIs. We inspected '/proc/self/ns/' within each container and found identical IPC and network namespace IDs, confirming they share namespaces. This means that they are able to collude to bypass cgroup restrictions. The container instances are billed based on their pre-allocated resources (vCPU and memory), while cgroups are employed to enforce such capacity limits. Escaping the cgroup allows a container to claim host memory beyond its paid quota. We verified that both AWS and Baidu Cloud provide far more memory than they sell: a glance at '/proc/meminfo' shows that, on Baidu Cloud, two 1 GB containers ran on a 32 GB host, while on AWS the host offered exactly twice the containers' combined limit. This surplus confirms the resource-abundance prerequisite for the attack.

IV. Defense

In this section, we first elaborate on the challenges of addressing the vulnerabilities mentioned above. We then propose a mechanism of *C*group *A*ssociating with *N*amespaces (`CANs`) that enables the collaboration between namespaces and cgroups, defending against the NCD threats.

A. Challenges of Addressing NCD Risks

The NCD vulnerabilities arise from failed collaboration between a container's cgroup and its shared namespaces. The sharing of the namespace can prolong the lifecycle of its managed resources beyond the lifecycle of the container's cgroup, creating a gap between resource management and constraints. Intuitively, banning namespace sharing might be the simplest solution to eliminate NCD vulnerabilities. Unfortunately, it would necessitate modifications to all popular container tools and compromise their compatibility with a large number of applications measured in Section III-C. Addressing this issue at the container tool level is also challenging, as it involves specific resource-sharing requirements inherent to certain container operations (e.g., `--ipc=host`).

A possible strawman solution is to designate an "oversight" cgroup as the parent for cgroups of containers that share namespaces. Establishing the parent cgroup can impose collective resource constraints for those containers. For example, Podman and Kubernetes allow grouping containers into a pod with a pod-level cgroup to enforce resource limits. Even after a container and its cgroup are terminated, the previously allocated resources remain subject to the pod-level cgroup's limits. This solution effectively prevents attacks in scenarios permitting inter-container namespace sharing, like public clouds. Unfortunately, it is impractical when the container shares the namespace with the host process. In this case, such a solution should set all containers and the host processes under a parent cgroup, which effectively means configuring the root cgroup (mentioned in Section II-C). Although cgroup limits are essential for constraining container resource usage, applying unified restrictions on containers and host processes is unreasonable. Containers and host processes have fundamentally different resource requirements; various system services on the host do not have any cgroup limits. The root cgroup is designed for monitoring system-wide resources rather than enforcing restrictions. Limiting it could inadvertently restrict critical system functions (e.g., init systems, logging daemons, and kernel threads), then destabilize the system.

On the other hand, upon the termination of a container, the associated cgroup also stops functioning, but the resources it monitored are not automatically released upon the cgroup's shutdown. If the container tool is to release these resources forcibly, it needs to identify them first. However, determining which resources in a shared namespace belong to the terminated container at the user space is a challenge.

B. Guidelines

To preserve original design choices of namespaces and cgroups, and ensure synchronized resource management and restriction between them, we have identified the following requirements for practical applications.

Effective protection. To permanently resolve NCD vulnerabilities when namespace shares, `CANs` must possess the ability to force the namespaces to cooperate with the cgroups for handling the residual resources after the container exits. `CANs`

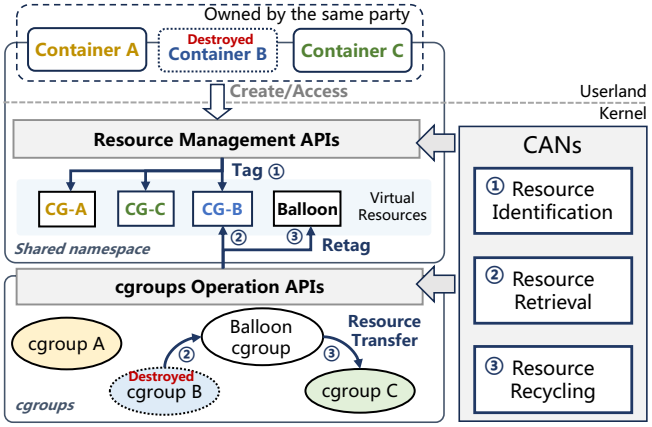


Fig. 4: Defense architecture

should also keep those residual resources within the limits to ensure that the host's resources cannot be abused.

Full compatibility. We aim to deliver seamless protection for user applications with full compatibility, ensuring that there are no changes required to the container tools or the applications they host. `CANs` should also maintain compatibility with all Linux ABIs, including syscall interfaces, without altering the functionality of namespaces and cgroups in the kernel or the usage of container tools at user space.

Service continuity. In our design, ensuring service availability holds paramount importance. Our defense mechanism must guarantee the robustness and resilience of services, maintaining their accessibility and functionality even under adversarial conditions. Furthermore, our defense framework is designed to seamlessly accommodate auto-scaling and rolling updates. These features are ubiquitous in modern container orchestration systems.

Minimal performance penalty. The lightweight and swift deployment are key features of Linux containers and are required by the container-based cloud computing mode. Therefore, `CANs` is expected to be simple and elegant, not significantly compromising the performance of container tools, while maintaining comparable performance to native Linux.

C. Overview

Fig. 4 illustrates the architecture overview. While namespaces are primarily designed for management purposes and cgroups are intended for resource restriction, these two functionalities currently exist at different levels within the Linux implementation. As illustrated in Fig. 5, although their functionalities are complementary, they are realized at distinct layers, with namespaces targeting virtual resources and cgroups targeting system resources. `CANs` bridges the namespace and the cgroups by placing a cgroup tag on the virtual resources managed within the namespace (detailed in Section IV-D). The tags not only link these resources to their responsible cgroup but also help pinpoint residual resources upon cgroup termination. To prevent the residual resources from evading the control of the exited cgroups, `CANs` employs 'balloon' cgroups that serve exclusively to retrieve such resources for a set of containers that share namespaces. The balloon cgroup imposes strict limits on resource usage, ensuring that the residual resources it monitors are controlled. No attacker can breach the safeguards enforced by the balloon cgroup. Through this mechanism, our defense solution eliminates the NCD problem at its root cause.

More specifically, `CANs` firstly labels each container's resource with its dedicated cgroup (①). Subsequently, when a cgroup is set to exit, `CANs` initiates a resource transfer (②), redirecting the outgoing cgroup's resource accounting to the balloon cgroup (detailed in Section IV-E). This ensures continuous monitoring of residual resources, maintaining the integrity of cgroup's restriction functionality. After that, if the residual resources are in use or required again, `CANs` recycles these resources from the balloon cgroup (③), allocating them to the specific container in need (detailed in Section IV-F). It is important to note that other containers within the same namespace may still be accessing the shared residual resources after the resource creator exits. To address this, `CANs` continuously reassigns and correlates the residual resources with the cgroup of the container that explicitly requests them. Upon completion of this recycling, the previously residual resources are claimed by the container, tagged under its cgroup's governance, and consequently, they are excised from the balloon cgroup's holdings.

The functionalities of `CANs` are integrated into the Linux resource management framework through kernel hooks. The advantage of doing so is that `CANs` can be very lightweight and compact during construction, reducing as much redundant code as possible, and also minimizing performance overhead. We have designed `CANs` to be compatible with Linux kernel v6.2.7 on x86 architecture, and made it open source [29].

D. Bridging of Namespace and Cgroup

Cgroups and namespaces have different granularities in resource management. Namespace manages virtual resources, while cgroup limits the system resources occupied by the virtual resources. This would hinder the construction of cooperation between namespaces and cgroups. For instance, cgroup subsystems concentrate on allocating system resources, such as memory pages, that are utilized by virtual resources within namespaces, including shared memory or semaphores. However, cgroups are unable to distinguish which system resources correspond to specific virtual resources. Therefore, to ascertain which cgroup is accountable for restricting these resources, we label the virtual resources with tags. These tags serve as the critical link that binds namespaces and cgroups together, bridging the two to facilitate seamless coordination.

Establishing cgroup-namespace mapping. Cgroup cannot 1) determine the specific namespace instance associated with the resources it monitors, 2) nor can it identify which resources need to be released upon termination within the namespace. Therefore, we create a global hash table `cg_ns_map` in the

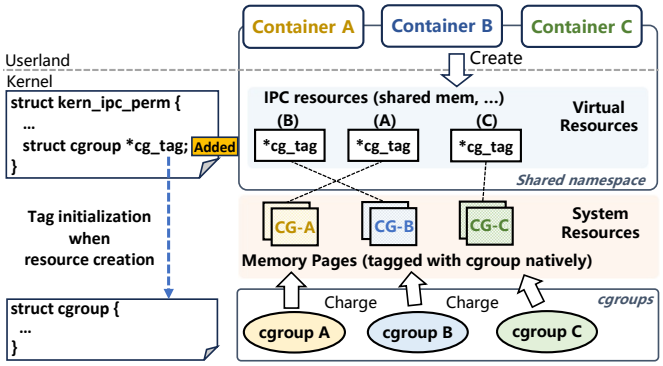


Fig. 5: Associating virtual resources (IPC) to cgroups

kernel to establish associations between cgroups and namespaces using key-value pairs. Each key in the hash table is a cgroup pointer that maps to sets of namespace instances where the processes in this cgroup reside. This hash table is updated whenever a process enters a cgroup. The instances of each process's namespaces (e.g., IPC, network, and PID namespaces) are joined into the corresponding set dynamically, ensuring an accurate and up-to-date mapping. The hash table helps the lookup of associated namespace instances after a cgroup terminates (detailed in Section IV-E).

Associating virtual resource to cgroup. We next introduce how the virtual resources in a namespace instance are linked to their associated cgroup instances. Associating cgroup instances with namespace instances only enables tracing the namespace instances connected to an exited cgroup. However, a single namespace instance could be shared by multiple containers, each with a dedicated cgroup to record resource usage. It is still unclear which virtual resources maintained in a traced namespace are attributed to the exited cgroup. This stems from the different levels at which cgroups and namespaces manage resources. Cgroups identify only low-level system resources (e.g., memory pages) and cannot determine which specific high-level virtual resource (e.g., IPC semaphores, virtual network devices) these pages belong to. Therefore, it is essential to establish a clear association between virtual resources and cgroups.

Fig. 5 shows the interaction between containers, virtual resources, and system resources, emphasizing that different virtual resources correspond to different system resources. Cgroups inherently tag system resources during their implementation, allowing for the identification of the container that owns the memory (mentioned in Section II-C). However, this tagging mechanism does not extend to determining ownership of virtual resources (e.g., semaphore), which requires additional hooks to be integrated into the virtual resource layer. Thus, we add a member to the structures of all isolated virtual resources with a cgroup tag that corresponds to the cgroup instance. Linux provides kernel functions for each kind of virtual resource to manage its lifecycle, including creation and traversal. For example, in the case of semaphores, Linux uses functions such as `ksys_semget` to create and access

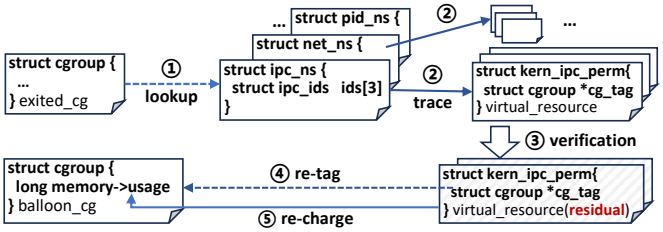


Fig. 6: Residual resources retrieval

them. When a semaphore resource is created, we intercept the `ksys_semget` function. We then tag the instantiated object `kern_ipc_perm` mapping to that semaphore resource, as shown in Fig. 5, with the identifier of the cgroup associated with the process that invokes the creation.

E. Residual Resources Retrieval

Upon establishing the linkage between the cgroup and namespaces, it becomes feasible to implement the reclamation of virtual resources confined by a cgroup upon its termination. Given that the accumulation of unconstrained resources can pose NCD risks, this reclamation process is designed to retrieve resources left behind by the terminated cgroup and reestablish appropriate constraints upon them.

Balloon cgroup initialization. For resource retrieval, we create the balloon cgroup that catches and monitors residual resources across a set of containers, as any container in this set exits. This set includes all containers that share at least one namespace with others in the set, thereby covering all containers belonging to the same party. Note that one or more containers sharing the same type of namespace with the host will also be grouped into one independent set. When implementing the balloon cgroup, the partitioning of container sets is integrated into the namespace creation procedure within the kernel. Moreover, we establish a global hash table to map the balloon cgroup with its governed container set. The key in the hash table is the pointer of balloon cgroup, which maps to a linked list of pointers, representing cgroup instances assigned to the containers within that set. The balloon cgroup is instantiated when the residual resources first arise due to the exit of one cgroup in the set.

Intercepting cgroup exiting. The lifecycle of a cgroup does not align with the lifecycle of its associated resources. When a cgroup exits, the resources it monitors are not automatically destroyed. Therefore, it is necessary to handle the cgroup's accounting during its exit through a comprehensive interception. During the cgroup destroy, the kernel function `cgroup_rmdir` is invoked to deallocate the cgroup instance. We hook this function to check whether residual resources exist when the cgroup is terminated, as depicted in Fig. 6. If residual resources are detected, they are transferred to the corresponding balloon cgroup. First of all, as described in Section IV-D, we look up all namespaces linked to the departing cgroup through the hash table `cg_ns_map` (①), then trace the virtual resources from these namespaces (②). Subsequently,

CANs verifies the resource tags to ensure they correspond with the exiting cgroup instance (③). Upon confirming a match, these residual resources are relabeled with identifiers of the balloon cgroup (④). Responsibility for monitoring and restricting these resources henceforth is transferred to the balloon cgroup (⑤). The dashed lines here denote the implementation specific to CANs, while the solid lines indicate the underlying Linux implementation invoked by CANs.

In addition, according to our design, the accounting of the residual virtual resources must be migrated from the cgroup being destroyed to the balloon cgroup. However, the functions provided by the kernel (such as obj_cgroup_uncharge and obj_cgroup_charge that are designed to handle the transferring) can only identify the underlying system resources consumed by the virtual resources themselves. Therefore to facilitate this transferring, we first identify which system resources underpin the virtual resources since we already associate virtual resources with the cgroup's system resources. Then, our defense leverages the cgroup-related functions to transfer the corresponding system resources to the balloon cgroup. For instance, after our defense tracks memory pages or slabs (specifically, a system resource) associated with the semaphore (i.e., a residual virtual resource), the subsequent transferring process involves two steps: first, the memory accounting is uncharged from the exiting cgroup utilizing the function obj_cgroup_uncharge, and then, it is recharged to the balloon cgroup through the function obj_cgroup_charge, ensuring that the resource accounting is accurately reflected.

Balloon cgroup configuration. The balloon cgroup works as a safeguard to prevent residual resources from growing unrestrictedly. The container owner on public platforms can purchase space for the balloon cgroup to cache residual resources. A *Max_Limit* can be set on the balloon cgroup by the system administrator to ensure that the sum of residual resources across all containers does not exceed this threshold. This approach not only ensures the security of the system but also defers the release of resources.

When containers exit, it is expected that all resources should be freed. Ideally, the balloon cgroup size should be set to zero. However, to accommodate functionalities such as the rolling update feature in Kubernetes [34] where we need to retain residual resources for a period, we allow the balloon cgroup to manage these resources temporarily. This ensures that active containers can draw from these resources as needed, maintaining service continuity during updates. Therefore, when determining the size of the balloon cgroup, administrators must consider the practical requirements and set the size accordingly. The balloon cgroup acts as a buffer, preserving the residual resources that other containers might need during a rolling update, while also guaranteeing that the system's resources are managed securely.

F. Resource Recycling

Since the residual resources would continue to be used by the containers, these kinds of residual resources should be reassigned to those containers and re-monitored by their cgroups. Additionally, when the space reserved for residual resources within the balloon cgroup reaches its maximum limit (e.g., usage reaching the allowed *Max_Limit*), a mechanism should be implemented to forcibly reclaim a portion of these resources. The handling strategies for these two steps have been described as follows.

Reassigning usable resources. When one container exits, the resources shared between containers in the same namespace might become residual. Yet, these resources often continue to serve other containers. In this scenario, responsibility for monitoring these residual resources will shift from the balloon cgroup to the cgroup of the container actively using them. We implement this transition by hooking the kernel functions that are called upon to interact with the residual resources within the namespace. For instance, as mentioned in Section IV-D, the kernel function ksys_semget is designed to access the semaphores managed by a namespace instance. If a semaphore is identified as a residual resource, a transfer procedure is triggered within this function. Each virtual resource is linked to a specific kernel function, providing a controlled access point. By enforcing resource transfers at these access points, we ensure comprehensive coverage of resource management.

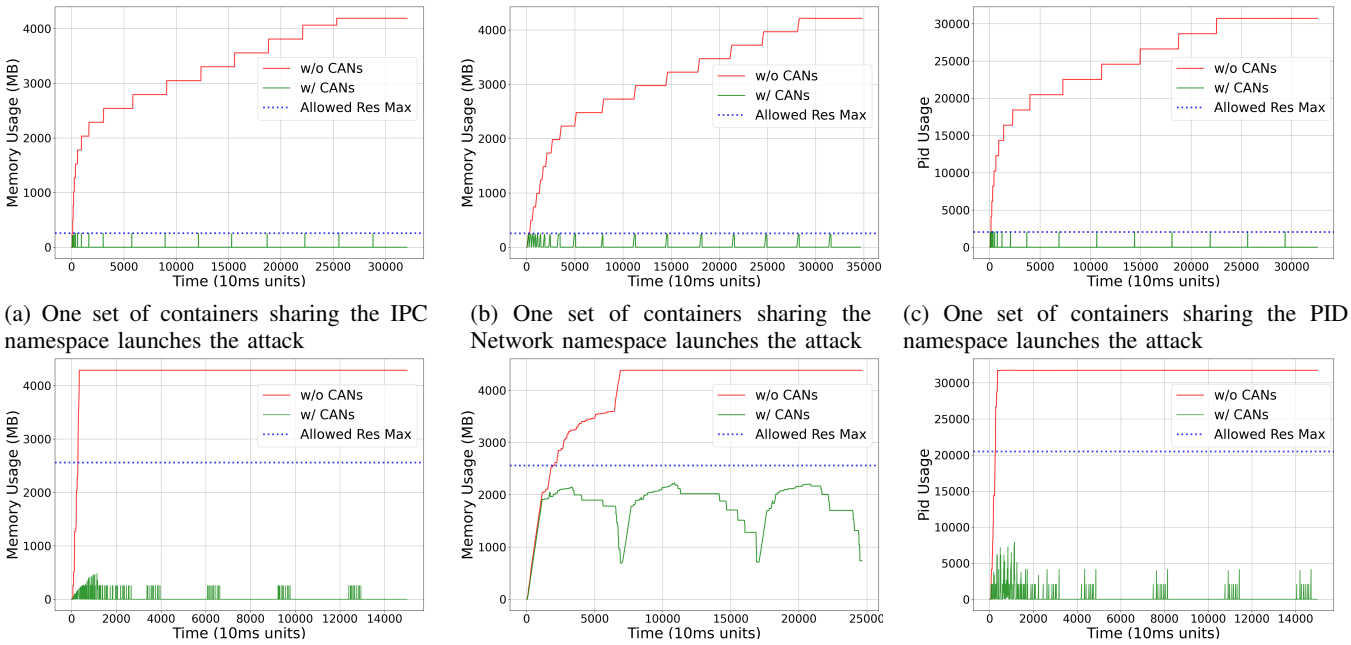
The implementation of such resource transferring is the same procedure as collecting the residual resources into the balloon cgroup (detailed in Section IV-E). Additionally, an audit precedes this transfer to verify that the relocation of resources does not exceed the destination cgroup's capacity. If the audit determines that the transfer would violate the cgroup's limits, the procedure is halted, and the residual resources are forcibly released. Subsequent kernel function calls seeking these resources will return an error.

Freeing expired resources. When the surge of new residual resources approaches the balloon cgroup's capacity threshold, a release of older residual resources is mandated. Upon reaching full capacity, the balloon cgroup employs a recursive release strategy, continually releasing the oldest residual resources until there is enough space to accommodate newly added ones. To facilitate this process, we have added a timestamp as a new field into the tag of each resource. The timestamp allows the balloon cgroup to manage its capacity effectively, ensuring that the oldest residual resources are prioritized for release.

G. Security Analysis

To understand the security guarantee, we investigate whether CANs can enforce the following security property: all virtual resources allocated by the container should be accounted for and restricted for their whole life cycle.

Mitigating attacks. CANs can perceive all residual resources by tracking the creation and destruction of all namespace and cgroup instances in the kernel. A dedicated balloon cgroup is always assigned to manage such residual resources generated by the same set of containers, regardless of whether these containers share namespaces with each other or with the host.



(a) One set of containers sharing the IPC namespace launches the attack

(b) One set of containers sharing the Network namespace launches the attack

(c) One set of containers sharing the PID namespace launches the attack

(d) Ten sets of containers sharing the IPC namespace launch the attack

(e) Ten sets of containers sharing the Network namespace launch the attack

(f) Ten sets of containers sharing the PID namespace launch the attack

Fig. 7: Effectiveness evaluation on CANs

The existence of the balloon cgroup eliminates such risks by imposing additional constraints on each set of containers.

• *Mitigating resource depletion*. Virtual resources created in a container are either actively released by the process, or are destroyed along with the namespaces when the container exits. In the first case, the resources released before container termination will be handled by the container's cgroup, which cancels their accounting correctly. This ensures that these resources cannot become residual resources, which will not incur security issues. In the second scenario, resources may become residual if the container's namespaces persist after the exit of both this container instance and its cgroup. Such residual resources could potentially lead to security issues.

CANs monitors all namespaces and provides the balloon cgroup to retrieve the residual resources (detailed in Section IV-E) that should be charged and restricted by the exited cgroups. The balloon cgroup inherits the duties of those destroyed cgroups to control such resources that were previously restricted. Therefore, CANs can constrain all resources created by containers until their release, preventing the resource depletion attacks from exhausting the host's resources.

• *Mitigating billing evasion*. The virtual resources accessible to a container can be classified into two types: those created by this container, and the residual resources maintained within its shared namespace. Those resources created by the container are charged and restricted by this container's cgroup at their creation, which is ensured by the cgroup architecture. The billing evasion becomes a risk because residual resources accessed by a container rather than the creator are not perceived or restricted by that container's cgroup.

Billing evasion attack takes an extra step compared to the resource depletion attack by continuing to use these residual resources. CANs not only prevent resource depletion attacks but also defend against billing evasion attacks by implementing resource recycling (see Section IV-F). A check is enforced to identify residual resources before they are accessed by a process. This identification is achieved by hooking kernel functions that manage interactions with residual resources. Enforcing resource transfers at these access points ensures comprehensive coverage of residual resource management. This ensures that even the residual resources are stolen by the billing evasion attacks they can still be accounted for and restricted again once being accessed.

No new attack surfaces. At the kernel layer, implementing CANs does not introduce any new interface to user space. As the interfaces communicating with the Linux kernel remain unaltered, no new attack surfaces are introduced that could compromise the kernel. Moreover, only system administrators have the capability to use the existing interfaces provided by cgroupfs [43] to configure the balloon cgroup like a regular cgroup. Since cgroupfs interfaces are unavailable to containers, the attacker owning any container set cannot manipulate or disable the balloon cgroups of its own or other sets. Furthermore, each balloon cgroup built by CANs is dedicated to containers owned by the same party. It is exclusively used for managing resources for its designated containers, without interference from other container sets. This mechanism ensures that residual resources resulting from an attacker's operations (such as container creation, restart, or deletion) within any container set will not affect the balloon cgroups (e.g., occupying its

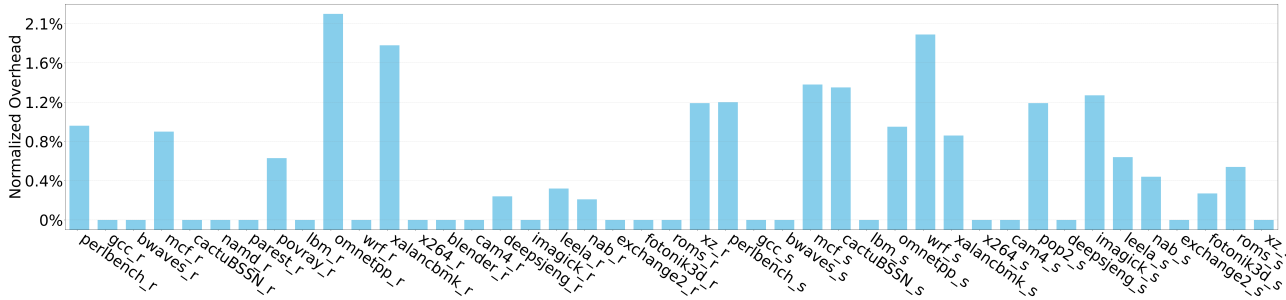


Fig. 8: SPEC 2017 overhead

capacity) belonging to other container sets.

V. Evaluation

In this section, we evaluate the effectiveness of our prototype, and examine its performance impact on benchmarks, container tools, and real-world applications. All experiments were conducted on Ubuntu 22.04, with the Linux kernel v6.2.7, and the hardware settings include a system with an 8-core 2.30GHz Intel Xeon Gold 6129 CPU and 16 GB RAM.

A. Effectiveness Analysis

We evaluated the effectiveness of `CANs` against the NCD attacks that are conducted by repeating the procedures discussed in Section III-B. During these attacks, we track the overall resource consumption of the whole system, using either one or ten sets of containers as the launch points for NCD attacks. Each container set consists of two container instances sharing a specific namespace, with resource limits set to 256 MB of memory and 2,048 PIDs. Additionally, the balloon cgroup limits for each container set are configured as 128 MB of memory and 0 PIDs. Fig. 7 presents the fluctuations in global resource usage during resource depletion attacks, with and without `CANs`. Regardless of the type of namespaces (IPC, Network, or PID) shared by malicious containers, the results demonstrate that without CANs, these containers can readily break out of their allowed resource limits. In contrast, with `CANs` enabled, the resource usage of those malicious containers is consistently restricted within the maximum permitted limits. By conducting the billing evasion attack, we traced the cgroupfs [43] and observed that the stolen resources were correctly re-charged by the cgroup assigned to the attacker's container, thus effectively preventing such attacks.

B. Performance Analysis

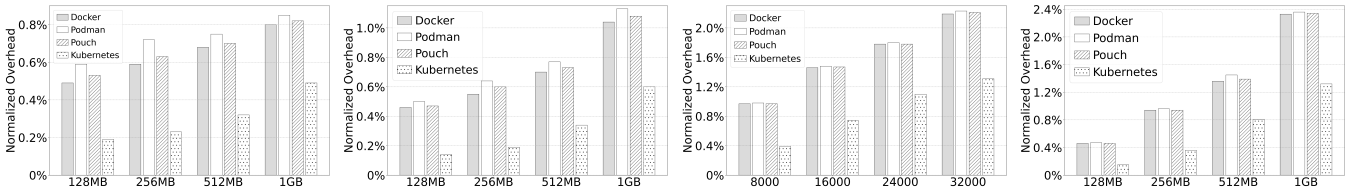
Performance of benchmarks. We evaluated the performance impacts of `CANs` using SPEC 2017 benchmarks [44], [45] and UnixBench [46], [47]. The overheads on SPEC 2017 and UnixBench are presented in Fig. 8 and Table II. Our results indicate that `CANs` introduces only minor overhead on computing operations in SPEC 2017. Specifically, after successfully running 43 benchmarks on the kernel with `CANs`, the overhead on all of them was observed to be below 2.10%, with 34 of them having an overhead below 1.00%. The overheads on 22 out of 43 benchmarks are almost zero, with an average overhead across these benchmarks is 0.47%, deemed negligible. Moreover, the highest overhead on UnixBench caused by `CANs` was observed at 2.14%, with overheads on 10 of 24 benchmarks being almost zero. The average overhead on these benchmarks was 0.33%, being practically negligible.

TABLE II: UnixBench overhead

	Benchmarks	Mean	Max	Min
UnixBench	24/24	0.33%	2.14%	0%

Performance of container tools. We next evaluated the overheads introduced by `CANs` on the container restart procedures (including creating and destroying containers) conducted by different container tools. These overheads primarily arise from retrieving residual resources made by the restarted container. To quantify such overheads, we measure the elapsed wall time of restarting a container that contains various NCD-exploitable resources with different amounts (discussed in Section III-B). More specifically, we utilized docker-ce v24.0.5, Kubernetes v1.29.3, Podman v4.8.3, and Pouch v1.3.1 to create two containers that share IPC and network namespaces, and generated one kind of resources (including semaphore, shared memory, message queue, and virtual network devices) within the restarted container for each restart iteration. The amounts of semaphores, shared memory, and virtual network devices in each evaluation range with their memory usage from 128 MB to 1 GB, and the number of message queues ranges from 8,000 to 32,000. The exploitable resource in the shared PID namespace is the process, which is not included in our evaluation, as such processes will not be retrieved but terminated by `CANs`, thereby avoiding additional overheads during container restarts.

Fig. 9 depicts the overheads on Docker, Podman, Pouch, and Kubernetes to restart a container respectively, and demonstrates that `CANs` only introduces tiny performance overheads. Specifically, Fig. 9a, 9b, 9c, and 9d depict the performance overheads of restarting a container that contains shared memory, semaphores, message queues, and virtual network devices with various amounts. While the overheads increase with the amounts of exploitable resources getting larger, the maximum overheads imposed by `CANs` for all container tools are not over



(a) shared memory (b) semaphore (c) message queue (d) virtual network device

Fig. 9: Overheads on container tools

2.36%, caused by retrieving 1 GB of virtual network devices. It is worth noting that the overheads observed in the experiments might exceed those encountered in real-world scenarios, where the amount of residual resources is unlikely to reach 1 GB. Further, the performance impact of CANs on Docker, Podman, and Pouch is similar but exceeds the overhead on Kubernetes, with the latter suffering a maximum overhead of only 1.32%. This is because the base time Kubernetes requires to restart containers surpasses that of other tools.

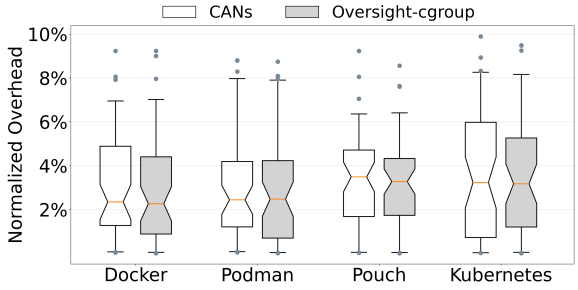


Fig. 10: Overheads on real-world applications

Performance of real-world applications. According to the architecture of CANs, the overhead is introduced at the creation phase of benign applications, when the connection between cgroups and namespaces of all container instances is established. The strawman solution also introduces overhead at this phase from creating and assigning an oversight cgroup as the parent cgroup for containers that share namespaces. To evaluate and compare their overheads, we sampled 150 top-starred applications from the repositories discussed in Section III-C. These applications are deployed by Docker, Podman, Pouch, and Kubernetes using multiple containers that are configured to share namespaces, either with each other or with the host, as required. We measured the elapsed wall time of deploying these applications, both with and without the mitigation, and evaluated the performance degradation. Note that the measured duration excludes image pulling, as it is unaffected by the mitigation and can be time-consuming.

Fig. 10 presents the distributions of the performance degradation caused by both solutions to those real-world applications. The boxplot bounds represent the 5th and 95th percentiles of overheads across all sampled application deployments, with outliers shown as dots outside the boxes. We observe that across all container tools, CANs and the strawman solution show comparable relative overheads. Their average is highest in Kubernetes, at 3.59% and 3.44% respectively, with corresponding absolute overheads averaging 42ms and 38ms. With the comparable overheads introduced, note that the strawman approach does not work in the presence of sharing namespaces with the host, while CANs does.

VI. Related Work

Container security has been extensively studied, revealing vulnerabilities and incompleteness in OS-level container techniques. Existing research has primarily focused on two key areas: cgroup limitations and namespace weaknesses.

A. Resource Accounting in Cgroups

As one of the cornerstones of container technology, cgroup is a focal point of research. Gao et al. [48] first disclosed that the cgroups are inadequate to account for and restrict all types of system resources. This inadequacy allows attackers within the container to leverage these unmonitored resources, launching a Denial of Service (DoS) attack on the host machine. Yang and Shen et al. [49] revealed that the abstract resources in the kernel, such as kernel variables and data structure instances, are also not governed by the cgroups and can be exploited by a container to exhaust the resources of the host. Yang et al. [37] proposed a detection method specifically designed to identify the types of kernel resources whose memory usage is not charged and restricted by cgroups. McDonough and Gao et al. [50] designed a fuzzing framework to discover the resources that the cgroups cannot restrict appropriately and can be exploited at userland by containers. Prior works have highlighted that cgroups cannot completely account for certain types of system resources that fall outside their monitoring scope. However, since NCD risks arise from monitored resources evading cgroup restrictions, their proposed solutions do not apply to our work.

B. Isolation Deficiencies in Namespaces

Deficiencies of namespaces to guarantee container isolation are another key research point. Gao et al. [4] reported that the Proc filesystem within containers does not entirely isolate from the host and exposes many information leakage channels for profiling host activities. Lin et al. [3] discovered that the settings for different Linux security mechanisms are not isolated among containers and pose challenges for setting independently. Sun et al. [51] implemented a security namespace to virtualize IMA for each container, which can verify the container's integrity independently. Li et al. [8] revealed the inadequate isolation on the filesystem between the host

and containers, and implemented a kernel-level approach to enhance the filesystem isolation. Liu et al. [52] proposed a detection framework to discover resource isolation bugs resulting from the incomplete implementation of namespaces or a lack of isolating mechanisms. All these works concentrate on the weaknesses of individual isolation mechanisms for container technology, yet overlook the security risks caused by their interplay.

VII. Conclusion

In this paper, we report the first systematic study on the NCD risks aroused from the collaboration breakdown of namespace and cgroup during namespace sharing. Our research discloses the wide impact of NCD risks across mainstream container tools and various real-world applications they deploy. Further, we reveal the underlying cause of the NCD risks and clarify the vast obstacle to eliminating them at userland. Therefore, we present a kernel-level solution called `CANs` to collaborate namespaces and cgroup. Our security analysis demonstrates that `CANs` eliminates the NCD risks effectively. Our approach incurs only a negligible performance impact on the kernel, container tools, and real-world applications, while providing excellent compatibility. Our discoveries and new isolation enhancement have made a step toward better designing the robust isolation for OS-level virtualization.

Appendix A
Ethical Considerations

Disclosures. We have reported our findings to Docker, Podman, Pouch, and Kubernetes, who have all confirmed the existence of the NCD risks we discovered [27] and allowed us to disclose it with the GitHub issue. Thus, we believe that publishing our work will not expose people to negative outcomes, such as harms or rights violations.

Experiments with live systems without informed consent. We believe that our experiments did not have any negative impact on the live systems involved in our work. Firstly, the local environment is enough for us to verify the effectiveness of the attacks and defense. Thus, we conduct all experiments with the local simulated environment, which will not impact any live systems. Secondly, for our measurement, we use the open APIs provided by GitHub at the allowed rate to collect data from public repositories. All use of the GitHub APIs in our work is in full compliance with their Terms of Service.

Acknowledgment

We thank the anonymous reviewers for their insightful comments, and Shepherd for the constructive feedback. Zhi Li, Zhen Xu, and Hai Jin are supported by the National Natural Science Foundation of China under Grant No.62202191 and No.62202196. Weijie Liu is partially supported by the Tianjin Municipal Natural Science Foundation No.24JCQNJC02140.

References

[1] P. Sharma, L. Chaufournier, P. Shenoy, and Y. Tay, "Containers and virtual machines at scale: A comparative study," in *Proceedings of the 17th International Middleware Conference*, 2016, pp. 1–13.

[2] "What is Enhanced Container Isolation?" https://docs.docker.com/security/for-admins/hardened-desktop/enhanced-container-isolation/.

[3] X. Lin, L. Lei, Y. Wang, J. Jing, K. Sun, and Q. Zhou, "A measurement study on linux container security: Attacks and countermeasures," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 418–429.

[4] X. Gao, B. Steenkamer, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang, "A study on the security implications of information leakages in container clouds," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 1, pp. 174–191, 2018.

[5] Z. Shen, Z. Sun, G.-E. Sela, E. Bagdasaryan, C. Delimitrou, R. Van Renesse, and H. Weatherspoon, "X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 121–135.

[6] Y. Ren, G. Liu, V. Nitu, W. Shao, R. Kennedy, G. Parmer, T. Wood, and A. Tchana, "Fine-Grained isolation for scalable, dynamic, multi-tenant edge clouds," in *Proceedings of the 2020 USENIX Annual Technical Conference*, 2020, pp. 927–942.

[7] Z. Li, J. Cheng, Q. Chen, E. Guan, Z. Bian, Y. Tao, B. Zha, Q. Wang, W. Han, and M. Guo, "RunD: a lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing," in *Proceedings of the 2022 USENIX Annual Technical Conference*, 2022, pp. 53–68.

[8] Z. Li, W. Liu, X. Wang, B. Yuan, H. Tian, H. Jin, and S. Yan, "Lost along the way: Understanding and mitigating path-misresolution threats to container isolation," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 3063–3077.

[9] M. S. Haq, T. D. Nguyen, F. Vollmer, A. S. Tosun, A.-R. Sadeghi, and T. Korkmaz, "Sok: A comprehensive analysis and evaluation of docker container attack and defense mechanisms," in *Proceedings of the 45th IEEE Symposium on Security and Privacy*, 2024, pp. 4573–4590.

[10] "capabilities(7)," https://man7.org/linux/man-pages/man7/capabilities.7.html.

[11] "seccomp(2)," https://man7.org/linux/man-pages/man2/seccomp.2.html.

[12] "SELinux," https://github.com/SELinuxProject/selinux.

[13] J. Thönes, "Microservices," *IEEE Software*, vol. 32, no. 1, pp. 116–116, 2015.

[14] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, "Serverless computing: Current trends and open problems," in *Chaudhary, S., Somani, G., Buyya, R. (eds) Research advances in cloud computing*. Springer, Singapore, 2017, pp. 1–20.

[15] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," in *Mazzara, M., Meyer, B. (eds) Present and ulterior software engineering*. Springer, Cham, 2017, pp. 195–216.

[16] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 1063–1075.

[17] S. Qi, L. Monis, Z. Zeng, I.-c. Wang, and K. Ramakrishnan, "Spright: extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing," in *Proceedings of the 2022 ACM SIGCOMM Conference*, 2022, pp. 780–794.

[18] O. Al-Debagy and P. Martinek, "A comparative review of microservices and monolithic architectures," in *Proceedings of 18th International Symposium on Computational Intelligence and Informatics*, 2018, pp. 149–154.

[19] "Docker: Host network driver," https://docs.docker.com/network/drivers/host/.

[20] "Example, join another container's PID namespace," https://docs.docker.com/reference/cli/docker/container/run/#example-join-another-containers-pid-namespace.

[21] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *Proceedings of the 2020 USENIX Annual Technical Conference*, 2020, pp. 419–433.

[22] "Kubernetes," https://kubernetes.io/.
[23] "Docker: Accelerated, Containerized Application Development," https://www.docker.com/.
[24] "Podman," https://podman.io/.
[25] "Pouch," https://pouchcontainer.io/.
[26] "Kubernetes Pods," https://kubernetes.io/docs/concepts/workloads/pods/.
[27] "NCD vulnerability reports," https://sites.google.com/view/containers-isolation/ncd-vulnerabilities.
[28] "CVE-2024-3056 Issue," https://github.com/containers/podman/issues/24192.
[29] "CANs," https://github.com/CGCL-codes/CANs.
[30] "Red Hat community Discussion," https://bugzilla.redhat.com/show_bug.cgi?id=2270717.
[31] E. W. Biederman and L. Networx, "Multiple instances of the global linux namespaces," in *Proceedings of the 2006 Linux Symposium*, 2006, pp. 101–112.
[32] "namespaces(7)," https://man7.org/linux/man-pages/man7/namespaces.7.html.
[33] "opencontainers/runc," https://github.com/opencontainers/runc.
[34] "Performing a Rolling Update," https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro.
[35] "ip-link(8)," https://man7.org/linux/man-pages/man8/ip-link.8.html.
[36] "keyrings(7)," https://man7.org/linux/man-pages/man7/keyrings.7.html.
[37] Y. Yang, W. Shen, X. Xie, K. Lu, M. Wang, T. Zhou, C. Qin, W. Yu, and K. Ren, "Making memory account accountable: Analyzing and detecting memory missing-account bugs for container platforms," in *Proceedings of the 38th Annual Computer Security Applications Conference*, 2022, pp. 869–880.
[38] "GitHub REST API documentation," https://docs.github.com/en/rest.
[39] "Objects In Kubernetes," https://kubernetes.io/docs/concepts/overview/working-with-objects/.
[40] "NVIDIA Multi-Process Service," https://docs.nvidia.com/deploy/mps/index.html.
[41] "Google Kubernetes Engine: Share GPUs with multiple workloads using NVIDIA MPS," https://cloud.google.com/kubernetes-engine/docs/how-to/nvidia-mps-gpus.
[42] "Videos of end-to-end attacks," https://sites.google.com/view/containers-isolation/ncd-vulnerabilities/attack-videos.
[43] "cgroups(7)," https://man7.org/linux/man-pages/man7/cgroups.7.html.
[44] "SPEC," http://www.spec.org/index.html.
[45] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.
[46] "Byte-UnixBench," https://github.com/kdlucas/byte-unixbench.
[47] "UnixBench," https://www.usenix.org/legacy/publications/library/proceedings/usenix01/freenix01/full_papers/loscocco/loscocco_html/node15.html.
[48] X. Gao, Z. Gu, Z. Li, H. Jamjoom, and C. Wang, "Houdini's escape: Breaking the resource rein of linux control groups," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1073–1086.
[49] N. Yang, W. Shen, J. Li, Y. Yang, K. Lu, J. Xiao, T. Zhou, C. Qin, W. Yu, J. Ma, and K. Ren, "Demons in the shared kernel: Abstract resource attacks against os-level virtualization," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 764–778.
[50] K. McDonough, X. Gao, S. Wang, and H. Wang, "Torpedo: A fuzzing framework for discovering adversarial container workloads," in *Proceedings of the 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2022, pp. 402–414.
[51] Y. Sun, D. Safford, M. Zohar, D. Pendarakis, Z. Gu, and T. Jaeger, "Security namespace: making linux security frameworks available to containers," in *Proceedings of the 27th USENIX Security Symposium*, 2018, pp. 1423–1439.
[52] C. Liu, S. Gong, and P. Fonseca, "Kit: Testing os-level virtualization for functional interference bugs," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2023, pp. 427–441.