



Lost along the Way: Understanding and Mitigating Path-Misresolution Threats to Container Isolation

Zhi Li

Huazhong University of Science and
Technology
Wuhan, China
lizhi16@hust.edu.cn

Weijie Liu✉

Ant Group
Beijing, China
yangzhi.lwj@antgroup.com

XiaoFeng Wang

Indiana University Bloomington
Bloomington, IN, USA
xw7@iu.edu

Bin Yuan

Huazhong University of Science and
Technology
Wuhan, China
yuanbin@hust.edu.cn

Hongliang Tian

Ant Group
Beijing, China
tate.thl@antgroup.com

Hai Jin

Huazhong University of Science and
Technology
Wuhan, China
hjin@hust.edu.cn

Shoumeng Yan

Ant Group
Beijing, China
shoumeng.ysm@antgroup.com

ABSTRACT

Filesystem isolation enforced by today's container technology has been found to be less effective in the presence of host-container interactions increasingly utilized by container tools. This weakened isolation has led to a type of path misresolution (Pamir) vulnerabilities, which have been considered to be highly risky and continuously reported over the years. In this paper, we present the first systematic study on the Pamir risk and the existing fixes to related vulnerabilities. Our research reveals that in spite of significant efforts being made to patch vulnerable container tools and address the risk, the Pamir vulnerabilities continue to be discovered, including a new vulnerability (CVE-2023-0778) we rediscovered from patched software. A key insight of our study is that the Pamir risk is inherently hard to prevent at the level of container tools, due to their heavy reliance on third-party components. While security inspections should be applied to all components to mediate host-container interactions, third-party component developers tend to

believe that container tools should perform security checks before invoking their components, and are therefore reluctant to patch their code with the container-specific protection. Moreover, due to the large number of components today's container tools depend on, re-implementing all of them is impractical.

Our study shows that kernel-based filesystem isolation is the only way to ensure isolation always in place during host-container interactions. In our research, we design and implement the first such an approach that extends the filesystem isolation to dentry objects, by enforcing access control on host-container interactions through the filesystem. Our design addresses the fundamental limitation of one-way isolation characterizing today's container, uses carefully-designed policies to ensure accurate and comprehensive interaction control, and implants the protection into the right kernel location to minimize the performance impact. We verify our approach using model checking, which demonstrates its effectiveness in eliminating the Pamir risk. Our evaluation further shows that our approach incurs negligible overheads, vastly outperforming all existing Pamir patches, and maintains compatibility with all mainstream container tools. We have released our code and filed a request to incorporate our technique into the Linux kernel.

*Weijie Liu is the corresponding author. Part of this work was done while Zhi Li was visiting Ant Group.

Zhi Li and Bin Yuan are with the School of Cyber Science and Engineering, Huazhong University of Science and Technology, Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Jinyinhu Laboratory, Wuhan, 430074, China.

Hai Jin is with the School of Computer Science and Technology, Huazhong University of Science and Technology, National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, Jinyinhu Laboratory, Wuhan, 430074, China.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](https://permissions.acm.org).

CCS '23, November 26–30, 2023, Copenhagen, Denmark

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0050-7/23/11...\$15.00

<https://doi.org/10.1145/3576915.3623154>

CCS CONCEPTS

• Security and privacy → Virtualization and security.

KEYWORDS

cloud native technology, OS-level virtualization, container security

ACM Reference Format:

Zhi Li, Weijie Liu✉, XiaoFeng Wang, Bin Yuan, Hongliang Tian, Hai Jin, and Shoumeng Yan. 2023. 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 (CCS '23)*, November 26–30, 2023, Copenhagen, Denmark. ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3576915.3623154>

1 INTRODUCTION

Container technologies promise secure, cost-effective, elastic, and high-productive software deployment and development. These technologies pack an application's code, data, and dependencies into a *container*, which leverages multiple kernel mechanisms, such as namespaces and cgroups, to achieve lightweight isolation of the application's runtime environment, including filesystem, network, process, and other system resources, from other applications and its host *operating system* (OS). Particularly, filesystem isolation for Docker, the most popular container, is implemented using 'chroot' to prevent processes within a container from accessing the host [4], and 'mount namespace' to isolate the mount points between the container filesystem and the host [25]. This combination is deemed sufficient for isolating the container's filesystem from the host, under the assumption that the host process refrains from accessing the container filesystem, which is considered untrusted.

Nonetheless, with the emergence of cloud-native development practices like CI/CD [1], and the new features it offers (such as copy and volume), host-container interactions now become necessary. However, such interactions open up new opportunities for security breaches, allowing a containerized malicious application to break the isolation. More specifically, such interactions are typically initiated by the container's owner, with the aim of importing host resources into the container or exporting the results of the container service back to the host. However, today's filesystem isolation between the container and host is inadequate for preventing potential security violations arising through the interfaces of such interactions. Indeed, in the past 6 years, a type of *path misresolution* (a.k.a., *Pamir*) risks have persisted on these interfaces and are responsible for nearly half of the 27 high-severity vulnerabilities reported for popular container tools, including the orchestration platform (e.g., Kubernetes [21]), management engines (e.g., Docker [12] and Podman [29]), and the runtime (e.g., containerd [5] and runc [26]). These vulnerabilities can be exploited by manipulating path resolution during the interaction, thereby enabling an escape from the container. Even more concerning is the fact that these vulnerabilities continue to be revealed across all container tools.

Understanding Pamir risk on container tools. In our research, we performed the first analysis on the Pamir risk, by looking into 12 high-profile vulnerabilities, as well as a series of bugs caused by the Pamir risk, across the five most prominent container tools mentioned earlier. To identify the origins of these vulnerabilities and the steps that can be taken to address them, we extracted and compared multiple call graphs related to the vulnerabilities from these container tools. Each call graph is drawn from these tools' vulnerable versions and the related fixes according to the GitHub issues. In addition, we manually analyzed the comments on the fixes from GitHub to understand how they evolved.

Looking into the distribution of these vulnerabilities and bugs, we found that the community of each container tool has been struggling for years to address the Pamir risk. However, the problem appears to be hard to resolve, since existing fixes tend to be circumventable or in some cases even introduce new vulnerabilities. Of particular interest is the observation that even after numerous iterations, when a security fix has been improved to the level that the Pamir risk to a given container tool can indeed be addressed, new

features that upgrade the tool disregard the fix and expose the tool again to the same risk, as evidenced by our discovery of a new vulnerability (CVE-2023-0778) in Podman whose Pamir vulnerability has been patched before [27, 30].

A key insight of our study is that the Pamir risk is fundamentally hard to prevent at the level of host applications (such as container tools), due to their heavy dependencies on third-party components (libraries, third-party executables, OS interfaces, etc.). The problem comes from today's one-way isolation of the in-container filesystem: while the resources outside the filesystem are beyond the container's namespace and therefore invisible to the containerized application, the host executables (including the container tool and the components it depends on) do not see any constraints in visiting the in-container filesystem. As a result, any security control on host-container interactions needs to be enforced on all filesystem-related operations performed by the host executable and its components. This becomes hard when it comes to the third-party components utilized by container tools: for example, the Linux mount tool is needed by Kubernetes to achieve the volume feature. Rendering this protection strategy all but impossible is the fact that the developers of these components do not have incentive at all to implement such container-specific control, instead believing that security checks should be performed by container tools before invoking their dependent components [31].

To get out of this dilemma, the developers of today's container tools come up with various ad-hoc solutions to avoid the risky behaviors of third-party components, such as preloading trusted dynamic-link libraries for a component before interactions. These patches, however, incur significant performance burdens [9]: for example, the overhead of the fixes to the Pamir risk on 'docker cp' can reach 200% (Section 6). More seriously, those patches are fragile and could be circumvented or even completely disabled by any update to the components or their supply chains: an example is the aforementioned fix to Docker's vulnerability CVE-2019-14271 [8], which becomes no longer effective with an update to glibc that enables automatic reloading of the preloaded libraries during a context switch. As a last resort, the developers of some container tools are forced to partially re-implement the replacements for non-compliant components. Our analysis shows that this will take considerable engineering efforts, since the vulnerable features of each container tool are related to more than 1,090 Golang packages on average. With more functionalities being added to container tools, more third-party components become inevitable, rendering such re-implementation increasingly hard.

Our solution. Our systematic analysis on known Pamir vulnerabilities shows that this security risk cannot be effectively controlled in the userland, by the container tool. Therefore, kernel-based filesystem isolation becomes the only viable solution to comprehensively mediating filesystem accesses from different kinds of third-party components to ensure isolation always in place during host-container interactions. This effort, however, is nontrivial, due to the current isolation design for filesystem, which is transparent even to the kernel. More specifically, to minimize the overhead for mediating access to filesystem, the current design simply segregates the mount points between the container and the host to confine the container application's operations within its namespace. As

a result, from the *virtual filesystem* (VFS), the kernel cannot tell whether a directory entry (dentry) object belongs to a container or not. This renders any illegal access to the filesystem hard to identify, as long as the path of the access request can be translated into the dentry object through the VFS. Further, even given the access target information (container or not), it is still challenging to design policies for accurate and comprehensive access control: for example, although the container tool should be allowed to access the object outside the container, this access request should not be induced by the application inside the container (Section 3.2). Finally, enforcement of the policies also needs to be well thought out to avoid any significant impact on the system’s performance and also ensure full mediation of the access.

In our research, we designed and implemented the first kernel-side defense against the Pamir risk. Our approach, call *Patrol* (Pamir control), extends the filesystem isolation to dentry objects, ensuring full mediation of host-container filesystem-related interactions. For this purpose, Patrol tags dentries according to their relations with containers and utilizes a set of carefully designed policies to regulate the access to these objects. Further, we analyzed the procedure of translating a userland path to the dentry object in the VFS, identifying the optimal location inside the path lookup function to enforce the policies, so as to ensure both complete and efficient control on the host-container interactions.

We verified the design of Patrol through model checking, showing that it is capable of completely eliminating the Pamir risk. We further evaluated our implementation on Docker, Podman, and Kubernetes. Our study demonstrates that Patrol vastly outperforms all userland fixes on vulnerable container tools, and only incurs an overhead of less than 4% on average. Also, Patrol is fully compatible with all mainstream container tools and is completely transparent: it does not affect the operations of the container tools, the components they depend on, and other applications. We have released our code [27] and filed a request to incorporate our technique into the Linux kernel.

Contributions. The paper’s contributions are outlined below:

- *Findings and takeaways.* We performed the first systematic study on the high-impact Pamir risk and demonstrate that this risk is fundamentally caused by insufficient filesystem isolation between the container and the host and cannot be addressed by the current protection. As evidence to the limitations of the solutions today, we identified and reported new Pamir flaws in the container tools that have been patched before.
- *New defense technique.* We designed a novel isolation technique and addressed the technical challenges in implementation inside the kernel efficiently and comprehensively. Our approach has been verified through model checking and evaluated against mainstream container tools, demonstrating the new solution to be more effective than any alternative proposed so far and also practical, with an exceedingly low-performance impact.

2 BACKGROUND

2.1 Virtual Filesystem

“Everything is a file” is one of the basic philosophies of Linux. Not only ordinary files, including directories, character devices, block

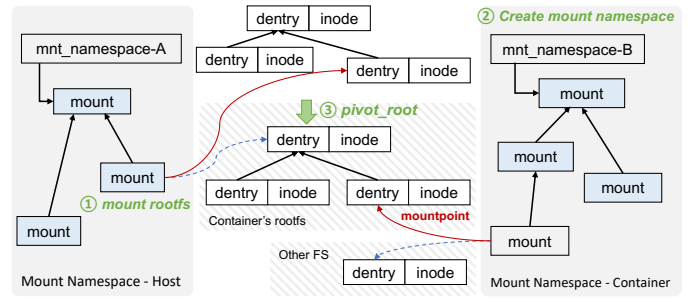


Figure 1: Container filesystem isolation

devices, sockets, etc., can be treated as files. The foundation of this design is Linux’s *virtual filesystem* (VFS) mechanism. It provides a unified interface layer for user programs to operate files, shielding the differences and operation details of different filesystems.

Dentry and Dentry tree. Dentry is a data structure that represents a file or directory object in the kernel. The dentry contains the name of the corresponding file or directory, and the hierarchical relationship between files and directories can be directly mapped on their dentries. Each dentry contains pointers referring to its parent’s dentry and the list of its children’s dentries, which makes dentries form a dentry tree. The VFS uses dentry trees to maintain the hierarchy of the file and directory objects. Of course, for large systems, it is impossible to put all dentries of the filesystem into memory. So these dentries in the tree are reference-counted. Dentry cache (Dcache) only maintains the referenced dentries and prunes the unreferenced dentries from the dentry tree to release memory.

Filesystem mounting. Linux kernel constructs mount trees to maintain the mount information in the system. As shown in Fig. 1, the leaf of the mount tree is a data structure named ‘mount’. The ‘mount’ structure contains a pointer to the dentry of the mount point on which the new filesystem mounts, as well as a pointer to the dentry of the new filesystem’s root directory. The dentries of the mount point and the new filesystem’s root do not have direct parent-child relation, but the ‘mount’ structure acts as a bridge connecting the dentries of the filesystems that have the mount relation.

Path lookup. The most common operation in the VFS is path lookup. The kernel uses the path lookup to translate the pathname from userland to its corresponding dentry object in the kernel. Typically, for every filesystem-related syscall like open, read, write, stat, mount, etc., the path lookup for resolving pathname is necessary. Specifically, the path lookup is to walk the pathname and find the dentry corresponding to the final component in the pathname (which we call *final dentry*), where a component is a substring delimited by ‘/’ characters. Each lookup process creates a ‘nameidata’ object to store its intermediate results (e.g., dentries for the components). At the beginning of path lookup, the first component’s dentry in the ‘nameidata’ object is initialized with the *starting point* of this lookup process, which is determined by the first character of the pathname. If a pathname starts with the ‘/’ character, the starting point is the dentry of the calling process’s root directory. Otherwise, it is the dentry of the process’s current working directory. And then, each component of the pathname is looked up within the children of the previous component’s dentry.

This is done by comparing the name of the component with the name stored in the dentry. The dentry discovered is saved into the 'nameidata' object by the kernel function `path_to_nameidata`. If a dentry for the component cannot be found, the path lookup process terminates with a failure. Regardless of whether the path lookup is successful, the kernel function `complete_walk` is called at the end of the lookup process to finish the path lookup and return a status code. During the lookup process, if a dentry represents a symbolic link (a.k.a., symlink), the kernel function `trailing_symlink` is called to follow this symlink and extract the pointed pathname. This pathname then replaces the symlink to continue the process of path lookup. If a dentry serves as a mount point, the lookup process follows the corresponding 'mount' structure in the mount tree to locate the dentry representing the root directory of the mounted filesystem, and continues the component lookup from this dentry.

Generally, the **starting point** and the **mount point** are the two key elements to decide the results of path lookup. If the process reaches a different root directory by such syscall `chroot`, the changed lookup starting point would lead the same pathname to a different dentry. Besides, if the process joins a new mount namespace with an isolated mount tree, the path lookup would also get distinction since the different mount namespaces could present a different mount view under the same mount point.

2.2 Container Filesystem Isolation

Linux kernel utilizes both namespace and `chroot`-like functionalities to isolate container filesystems. Linux namespaces enforce process-level resource isolation. Particularly, using mount namespaces, an isolated filesystem mount hierarchy for each namespace can be created [25]. This ensures that processes running in each mount namespace have their own independent views of mount points. Further, the `chroot`-like functionality can designate a new root directory to the processes, restricting the resources accessible to them within this new root directory. The independent view of mount points prevents the process from accessing the filesystems mounted in other mount namespaces, while `chroot` forces the process can only access the resource within the given directory. As a result, the process is completely isolated from unauthorized resources.

Specifically, the progress of creating and isolating a container filesystem is illustrated in Fig. 1. Initially, the container tool extracts the rootfs from a container image and mounts it on a given host's directory (①), such as `"/var/lib/docker/[storage driver]/[ID:sha256]/"` [14]. Then, the container tool invokes the syscall `clone` with the flag `CLONE_NEWNS` to create the container's initial process. This generates a new mount namespace in conjunction with this process, and all processes within the container join this namespace (②). Typically, a new mount namespace is cloned from the mount namespace of the invoking process, which causes the container's mount namespace to inherit the mount tree of the host. Only the new mounts added within these namespaces are invisible to other namespaces. In this case, the container processes can still traverse the host's filesystem. Thus, the `chroot`-like syscall `pivot_root` is used to reset the root directory of these processes with the container's rootfs (③) to ensure that the processes can only access files and directories, including mount points, under the rootfs. Nonetheless, this isolation is still incomplete, and many vulnerabilities can bypass it.

3 INCOMPLETE FILESYSTEM ISOLATION

Although each container has a separate filesystem view from the host, the isolation between their filesystems is not fully complete. In this section, we first clarify the threat model and discuss the Pamir vulnerabilities (bug issues/CVEs) in container isolation next. Lastly, we shall showcase the measurement and comprehension we derived from the present-day industry solutions to such vulnerabilities.

3.1 Threat Model

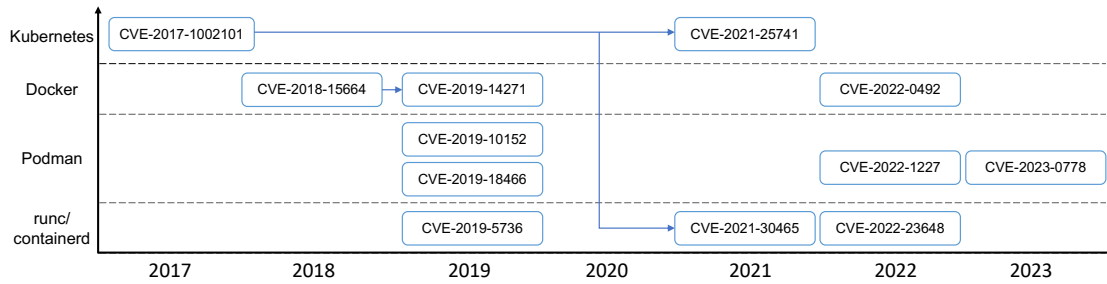
We consider container-based platforms managed by container tools (e.g., Docker and Kubernetes), and multi-tenant containers running on these platforms can share the same OS. The container tools and the OS on the platforms are trusted but vulnerable. The adversary in container-based platforms can control multiple containers and attempt to disrupt the OS or other tenant's containers. Moreover, the adversary holds the capacity to manipulate all resources within containers, such as the filesystem and processes. Additionally, they can exploit legitimate interfaces to prompt the platforms to interact with these resources in order to commit an escape. These legitimate interfaces encapsulate container tool functionalities such as copy, run, and volume. This is also the typical threat model in every other container escape scenario.

3.2 Pamir Vulnerabilities

During host-container interactions, various container tools access the container filesystem from the host context. Nevertheless, the container is considered untrustworthy in the eyes of these tools and may deceive them into accessing malicious payloads in its filesystem. The vulnerabilities that emerge due to path misresolution (Pamir) are recurrently exposed in these attributes and have an impact on all widely-used container tools, despite being developed by different communities (refer to Fig. 2). These vulnerabilities are primarily exploited in two ways to compromise the host, enabling instances of information disclosure, privilege escalation, arbitrary file manipulation, and other related security concerns.

Symlinks resolution cheating is one of the ways to exploit the Pamir vulnerabilities. Typically, during host-container interactions, container tools may access a symlink in the container filesystem, either to read or write to it. However, this symlink is resolved within the host context instead of within the container context. A sophisticated symlink in a malicious container can deceive container tools into navigating beyond the container filesystem and accessing a sensitive location (such as `/etc/passwd`) on the host.

Malicious containers can exploit 'volume' features that have Pamir vulnerabilities to traverse arbitrary files on the host. The 'volume' feature is used to mount a volume into the container filesystem, which can be a host's directory, to provide persistent storage. Moreover, the 'subPath' property in Kubernetes enables sub-directories in a volume used by one container to be mounted into other containers [22]. However, if the subdirectory is replaced with a symlink by the malicious container, an arbitrary directory on the host that is referenced by this symlink will be mounted into another malicious container, leading to the earliest vulnerability CVE-2017-1002101 [7] in the Kubernetes 'volume' feature. Although Kubernetes has rectified this issue, similar attack surfaces still exist in its dependent components, resulting in comparable

Figure 2: Chronology of Pamir-related CVEs¹

vulnerabilities such as CVE-2022-23648 and CVE-2021-30465 in containerd and runc, respectively. The C version of runc also has the same vulnerability [32].

Homologous vulnerabilities also exist in ‘copy’ functionalities and allow to traverse files on the host as well as overwrite any host’s file. The ‘copy’ features are implemented by the container tools to copy files between the container and host. However, whether copying from or to the container, either the source or destination of copying is under the untrusted container control. Specifically, if a malicious symlink replaces the file being copied from a container, the source of copying could be led out of the container by this symlink. Similarly, a malicious symlink that substitutes the destination of copying in the container can mislead a file being copied to an arbitrary directory on the host rather than this container. Similar vulnerabilities CVE-2019-10152 and CVE-2019-18466 also arise in Podman, despite the fact that Podman’s developers, inspired by CVE-2018-15664 [30], strive to avoid this kind of vulnerability. Moreover, we also discover a vulnerability, CVE-2023-0778, in Podman’s feature ‘export volume’ that follows the same principle.

In addition, Procsfs creates specific symlinks, such as ‘/proc/self/exe’, which can also be exploited for container escape. Since container tools may enter the container during host-container interaction, such as joining the container’s namespaces, they may be tricked into accessing these symlinks within the container. For instance, if ‘/proc/self/exe’ is accessed, the container tools’ executable on the host can be opened within the container. Notably, vulnerability CVE-2019-5736 exemplifies how processes within the container can rewrite the runc executable via this channel.

Inducing illegal file execution is another way to exploit the Pamir vulnerabilities. Apart from being read or written, files in the container also can be executed by the host’s process (e.g., container tools) during host-container interactions. However, this execution might cause malicious payloads in the container to act in the host context.

Although all container tools strive to avoid this kind of execution, many unforeseen executions are still concealed in their features. For instance, Docker is unaware that it will load a dynamically linked library ‘nsswitch’ in the container during the ‘copy’ feature. Specifically, this library is required and loaded by a Golang package, which is invoked by Docker to bundle the copied files, but this package does not realize that this loading happens after the syscall

chroot is invoked. Using syscall chroot is the approach to the fix of vulnerability CVE-2018-15664 to restrict the symlink resolution within the container. However, in this way, the library ‘nsswitch’ is also forced to be resolved within the container filesystem. But the process still belongs to the host, leading to a new vulnerability, CVE-2019-14271. Similarly, vulnerability CVE-2022-1227 in Podman originates from invoking the ‘nsenter’ executable in the container through the ‘podman top’ feature in the host context.

Additionally, it is worth noting that the system process running on the host is not immune to executing tampered files originating from an untrusted container. These kinds of files are provided by Cgroupfs and act as the interaction interfaces for sharing the cgroups information between the host and container [3]. Among them, a special file ‘release_agent’ in the Cgroupfs would be executed by the host system process after the container is terminated. The CVE-2022-0492 vulnerability is a direct result of this file being compromised by a malicious container. Typically, the container can only access the file ‘release_agent’ from the ‘/sys/fs/cgroup’ within its own filesystem; however, a container with an inadequate security configuration can bypass the permission check and alter this file. This allows any malicious payload in the ‘release_agent’ file to be executed on the host system.

3.3 Understanding Fixing Methods

Despite numerous attempts by the mainstream container tool communities to address the Pamir vulnerabilities, they continue to persist. In order to identify the shortcomings of these attempted fixes, we utilized the *go-callvis* open-source tool [17] to extract call graphs from both the vulnerable and updated versions of the previously mentioned container tools. Through manual analysis of the relevant call graphs and careful examination of all associated comments on GitHub, we are able to gain a comprehensive understanding of the evolution of these fixes.

Observation 1 - Third-party components of container tools hinder the vulnerabilities from being fixed thoroughly. Fixing vulnerabilities in container tools may not eliminate attack surfaces in their dependent components, which poses a serious risk.

In response to the symlink resolution cheating, a new package ‘secureJoin’ [16] has been developed to replace the Golang package ‘filepath’, with the aim of achieving secure path resolution by verifying if a symlink is resolved into the specified directory (e.g., the container’s root). However, despite significant engineering efforts to implement this fix, the inherent risk of cheating has not been fully

¹The arrows represent that the subsequent vulnerability is caused by the incomplete fix of the previous vulnerability.

eliminated [10]. This is due to a race condition between the resolution checking process and subsequent path-related operations (e.g., copying or mounting), which can be exploited through a TOCTTOU attack to replace a verified path with a malicious symlink. Specifically, this replacement is performed by the attacker within the container after the container tool on the host finishes the resolution checking. This results in a range of vulnerabilities present in various container tools, such as CVE-2018-15664, CVE-2021-30465, CVE-2021-25741, CVE-2022-23648, and CVE-2023-0778.

Despite various attempts to fix the vulnerabilities in container tools, they continue to remain vulnerable. For instance, the Podman community has chosen to pause a container when its filesystem is being accessed by Podman, as a frozen container cannot conduct the race. However, this fix comes with significant overheads (see Section 6), and a weakness has been identified [27] that can bypass it. Specifically, one container can still access the volume and conduct a TOCTTOU attack when Podman interacts with the frozen container if two containers share a volume.

Moreover, the communities of Kubernetes, containerd, and runc have attempted to eradicate the race condition by atomizing the resolution checking and the subsequent path-related operations. However, this atomization can be broken by any third-party executable invoked by these subsequent operations. Specifically, this atomization is achieved by substituting the checked path with the magic link `‘/proc/[pid]/fd/[fd]’` for subsequent operations. This magic link always refers to the checked path, but in the ‘volume’ feature of Kubernetes, the Linux mount tool [24] invoked after the resolution checking resolves the magic link by default and uses the resolution result to call the syscall mount at the end. This resulted in the vulnerability CVE-2021-25741, which can only be fixed by invoking the mount tool with the flag `–no-canonicalize` to disable the resolution.

The Docker community’s fixes aim to eliminate symlink resolution cheating in its ‘copy’ feature but result in the illegal execution issue due to the uncontrollable behaviors of third-party packages. The fixes in the ‘copy’ feature use the syscall `chroot` to enter the container filesystem before accessing a container’s path, ensuring any symlink in this path is always resolved within the container. However, the third-party packages called by the ‘copy’ feature may not realize that the `chroot` leads them into an untrusted environment, e.g., CVE-2019-14271 is caused by the ‘nsswitch’ library loaded unexpectedly in the container (see Section 3.2). The community fixed this vulnerability temporarily by loading all dynamic libraries at the beginning of copying. An update of the GNU C Library (glibc) [2], which is transparent to any container tool, could invalidate this fix. This update enables glibc libraries, including ‘nsswitch’, to be reloaded automatically after such a context switch triggered by the `chroot`.

Observation 2 - OS interfaces-related vulnerabilities are difficult to be fixed by container tools thoroughly. Such vulnerabilities related to pseudo filesystems cannot be eliminated by container tools.

Considering the vulnerability CVE-2019-5736, the fix can only block the attack aftermath rather than eliminate the attack surface in the *Procs*. To be specific, this fix aims to generate the copy of the runc executable to perform each host-container interaction. Even if the vulnerability CVE-2019-5736 could be exploited successfully,

just the copy is overwritten rather than the original. In addition, the vulnerability CVE-2022-0492 related to the *Cgroupfs* interface also cannot be eliminated by the container tools. Their solution is just that reduces the container’s privileges (i.e., disabling privileged user namespaces) to avoid ‘release_agent’ being modified by the container, but this affects the ability to run containers. In the end, a kernel patch enforces access control on the vulnerable interface, fixing this vulnerability thoroughly.

3.4 Insights

The fixes in the userland are impeded by the third-party components in the container tools, which cannot eliminate the underlying cause of ubiquitous Pamir risks. Fundamentally, the Pamir risk emanates from the incomplete kernel-based isolation. Improving this filesystem isolation presents the optimal solution to this issue.

Insight 1 - Fixes in the userland are ineffective and impractical in eradicating the Pamir risk. With respect to the Pamir risk, comprehensive inspections of container tools can only address attack surfaces in the container tool itself and not in its dependent components. For instance, the Kubernetes community added approximately 3,700 lines of code to enforce path resolution checks, which only mitigates vulnerability CVE-2017-1002101, while exposing identical attack surfaces in its dependent components. Vulnerabilities such as CVE-2021-30465 and CVE-2021-25741 can bypass these checks. Container tools use ad-hoc solutions to circumvent risky behaviors in third-party components, but these solutions are fragile and difficult to design thoughtfully. For example, the fix for CVE-2019-14271 (discussed in Section 3.3) is an ad-hoc solution.

To this end, the developers tend to add security inspections in the dependent components, but such proposals are typically not accepted. Such the package ‘secureJoin’ developed by the communities of container tools is the extension of the Golang package ‘filepath’, which adds the path resolution inspecting to the package ‘filepath’ (mentioned in Section 3.3). However, the Golang community declines to merge this package into its standard library. The community believes that this package is tailored to container tools only and is not a generic function that could be utilized elsewhere. Furthermore, the Golang community contends that security inspections should be conducted by the container tools before calling its libraries and does not recognize this as the responsibility of any Golang library [31].

It is a paradox that the third-party components request the container tool to be in charge of the security inspections but potential attack surfaces in them would be not eliminated by these inspections. Re-implementing the risky third-party components might become the last way to solve the Pamir risk, but it requires significant engineering efforts and may not adhere to best practices in software engineering [47, 57]. For example, the Podman community modifies 40,836 lines of code to partially implement the Linux utility ‘nsenter’ within Podman, solely to address CVE-2022-1227 in the ‘podman top’ feature. Additionally, there are 7,498 lines of code modifications to implement fractionated functions of the third-party package ‘archive/tar’ in the ‘docker cp’ feature, which is deprecated to avoid CVE-2019-14271 after the ad-hoc fix is no longer valid.

In reality, our analysis of third-party packages called by container tools underscores the irrationality and unsustainability of re-implementing such components. Specifically, we utilized the Golang built-in package management tool [23] to identify the packages invoked in their vulnerable features and recursively queried the website *pkg.go.dev* [18] to trace the supply chain of these packages. The results demonstrate that presently, over 5,458 Golang packages are relevant to container tools (further information is provided in our website [36]). As container tools incorporate more features over time, the number of third-party components to manage will inevitably increase.

Insight 2 - Kernel-based filesystems isolation between the containers and host should be enhanced to eliminate the end-less Pamir vulnerabilities. The incomplete kernel-based filesystem isolation between the container and host is the underlying cause of the ongoing Pamir vulnerabilities. The mount namespace and *pivot_root* offer some level of isolation, but they do not fully cover all VFS objects such as dentry, which makes it difficult for the kernel to distinguish whether the dentry object belongs to a container or not after resolving a userland path. This incomplete isolation means that a process on the host cannot differentiate the container's filesystem and might treat it as a trusted environment if the security inspections do not restrict the process. Furthermore, the incomplete isolation does not block a container's process from accessing the host's filesystem through illegal channels such as *'/proc/self/exe'* (as seen in the vulnerability CVE-2019-5736). Therefore, enhancing the kernel-based filesystem isolation between containers and the host is crucial to eliminate the Pamir risk.

In summary, we believe that the root cause of the Pamir risk lies in the incomplete filesystem isolation in the kernel. Therefore, to eliminate the risk entirely and elegantly, we must restructure the kernel's filesystem isolation mechanism. Specifically, we need to extend the filesystem isolation to the dentry objects in the VFS and identify whether each dentry belongs to a container filesystem or the host. This extended isolation mechanism can then enforce access control policies based on the process and dentry attributes to prevent illegal access between the containers and host. By implementing these measures, we can address the underlying issue that leads to the Pamir risk and improve the overall security of container environments.

4 PATROL

Here we propose a container filesystem isolation mechanism named *PAth misresolution conTROL* (PATROL) that can restrict the behavior, defending against the Pamir attack. In this section, we elaborate on the design of Patrol and how it can eliminate filesystem-related container-escape issues in a systematic manner.

4.1 Guidelines

To effectively enhance the container filesystem isolation, fortify the mount namespace, and expedite the incorporation of our prototype into practical applications, we expect the following requirements to be met by the design.

Complete protection. To permanently resolve those vulnerabilities in container-escape that are related to filesystems, Patrol must possess the ability to isolate the malicious symlink resolution and

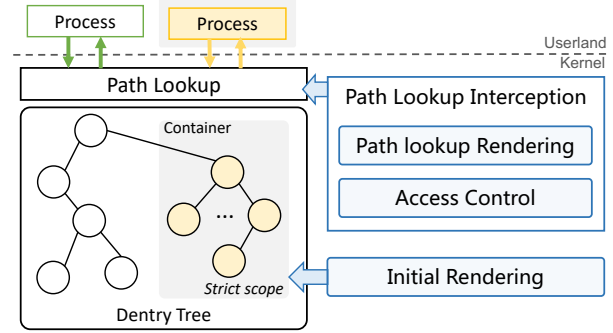


Figure 3: Design overview of PATROL

perform a complete permission check on cross-namespaces file accesses.

Full compatibility. We would like to provide transparent protection for user applications so that there would be NO modifications to the container tools and NO modifications to user-space applications running inside the container. And Patrol should have no compatibility issues with all syscall interfaces.

Minimal performance penalty. In the context of serverless computing or Function as a Service, both cloud service providers and users anticipate swift deployment of serverless functions. Additionally, it is expected that any runtime checks will not significantly compromise performance. Therefore, the implementation of our security-enhanced kernel would be both simple and elegant, while maintaining comparable performance to native Linux.

4.2 Overview

Our overarching goal is to keep the system running smoothly while isolating the container and host filesystems as much as possible. Behind the design of Patrol, the main idea is to enhance the kernel-based container filesystem isolation.

Fig. 3 illustrates the new design as it pertains to our extended filesystem isolation. By placing tags on dentries within each container, Patrol extends the isolation to the VFS objects. We call this procedure “rendering”. It should be noted that a host process can also pass through the container's filesystem and access the host's dentry (mentioned in Section 3.2), so the static rendering at the beginning is not enough. Therefore, Patrol also performs a dynamic *path lookup rendering* and regulates the access to these dentries during path resolution. With the rendering procedures and access control integrated into the Linux path lookup routine, we make the isolation a whole piece. The advantage of doing so is that Patrol can be very lightweight and compact during construction, reducing as much redundant code as possible, and also minimizing performance overhead.

Differentiating dentries between container and host. During container initialization, we need to identify the dentry objects of the container filesystem in the VFS and distinguish between objects in the container filesystem and the host filesystem. This identification is crucial to establish the basis for access control later on. Rendering is an effective way to indicate the scope to which a dentry belongs. Using specific information, such as the root path of a container

filesystem, the operating system can determine the scope of a dentry through the dentry tree. In addition, we combine the static initial rendering and the dynamic path lookup rendering mechanisms to complete the differentiation and isolation mechanism (as detailed in Section 4.3).

However, many existing container tools offer a “shared volume” interface (such as ‘docker run -v’) that allows files to be shared between containers or between the host and containers, making the rendering more complicated. To address this issue, we define multiple security levels when considering real-world container file-accessing scenarios. The target file could be at one of the following three: *within a container filesystem*, *on the host’s filesystem*, or *shared between the host and (multiple) containers*. Firstly, as described in our threat model (Section 3.1), the container is not trustworthy. Adversaries can manipulate everything in its container therefore we must apply the most strict access control policies on each container’s filesystem. To persist the data in shared volumes, the underlying implementation uses “bind mount”¹ to bind the paths in the host to the container. The shared volume could be seen as an extension of the container filesystem but the content can be accessed by different sharing parties. Thus secondly, we should set tags at runtime to distinguish subjects who have access to the shared volume. Finally, the host file has the lowest risk level (i.e., the highest security level) that only can be accessed by host processes.

Access control and policy enforcement. After the rendering, Patrol becomes aware of the respective dentries allocated to each party. This allows for the implementation of access control measures, ultimately thwarting any malicious endeavors. To impede container escape, a straightforward policy would be that any file path from a container cannot be resolved beyond the container filesystem. However, such a simple access control design is not accurate and comprehensive. For example, a host process that belongs to the container tool has access permission to the object outside the container, while this access request should not be induced by the application inside the container. Therefore, we also need to restrict path resolution requests which are from a host process but are meant to have an effect within the container.

Our access control policies, as detailed in Section 4.4, are enforced in the path lookup process as described in Section 4.5. Our policies involve checking path resolution results and file execution permissions. These policies correspond to two types of rules we designed - one relating to object access and the other to control transfer.

To ensure that policy enforcement does not significantly impact system performance and fully mediates access, we intercept all path resolution requests, which is the central concept that underpins the development of Patrol. We have designed Patrol to be compatible with Linux kernel 5.4.1 on x86 architecture, inserting approximately 420 LoCs in total, which is always invoked and small enough. To ensure full compatibility, we have also improved Patrol’s implementation to fully utilize the kernel’s reasoning ability with specific information (in Section 4.6). This means that it can be applied in

¹Bind mount is a mount type provided by Linux. It can mount a file or directory to a new target path. After this kind of mount, the original data can be accessed from both the old and the new paths. Changes to the data from both paths will also take effect, and the original content will be hidden.

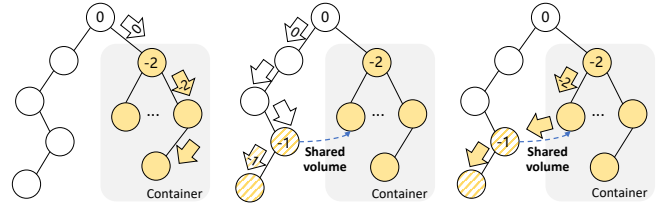


Figure 4: Multi-stage rendering

the kernel without affecting the upper application program. We use a heuristic rule to contain the parsing scope of the symlink.

4.3 Multi-Stage Rendering

From the VFS, the kernel cannot tell whether a directory entry (dentry) object belongs to a container or not. Therefore, we tag the dentries in the VFS to distinguish them. At the same time, in order to identify whether the finally obtained dentry has security risks during the path lookup process, here we combine the static (initial rendering) and the dynamic (path lookup rendering) approaches and use this multi-stage rendering strategy to track the scopes crossed by the path lookup procedure for later access control.

Initial rendering on different scopes. We give the definition of *scope* which is a set of dentries belonging to a container or a shared volume whose functionalities are restricted by our security policies. We extend the PID namespace structure, adding an enum variable ‘security_level’ as the security level flag. The flag has three possible values: strict (-2), shared (-1), and normal (0), which represent three scopes respectively. To mark different scopes, the ‘dentry’ structure is extended, and a PID namespace pointer is added as the tag, which is used to distinguish the namespace that the current dentry belongs to. The security level flag in the PID namespace represents the risky levels of the dentry. Since we do not trust the container, files (referred by dentries) in the container should be tagged with the highest risk level (a.k.a., the lowest security level) - “strict”. As the name suggests, we will assign “shared” and “normal” to dentries in shared volumes and the host respectively. The advantage of this rendering strategy is that the namespace can distinguish different containers naturally and can be used to prevent illegal access later between the host and containers.

During the container startup, a related filesystem is created, and a scope-unique tag is assigned to the container’s dentries. The pivot_root function is used to switch the container process to the root directory of the container rootfs. At this point, we use the destination directory of pivot_root as the starting point to traverse all dentries under it. Dentries are traversed recursively, and if a dentry is the mount point, the dentries belonging to the mounted filesystem will also be tagged. The security level flag of the dentries belonging to the container is set to “strict”. In the case of a shared volume, the filesystem being bind-mounted is shared by the host and the container. A new temporary PID namespace is allocated to tag the dentry in the shared volume, and the security level flag is set to “shared”.

Path lookup rendering. Relying solely on the identity of the dentry based on the initial PID namespace is inadequate for access

control. For example, when a host process accesses and parses a symlink in the container, a dentry of the host's file can be obtained. The path lookup results indicate that the host process has accessed a host file, which seems legitimate. However, the process has actually entered an untrusted container filesystem, and illegal access has been induced to the outside of the container by a malicious symlink. This poses a significant security risk and needs to be controlled. Since the path lookup would cross the namespaces between the host and containers, dynamic path lookup rendering should be performed to determine whether this lookup process has stepped into a namespace with a lower security level.

During the path lookup process, the 'nameidata' object is used to record the intermediate results. So we add a member to this object with a PID namespace tag to keep track of the riskiest scope traversed in each path lookup, and use the PID namespace pointer of the current process to initialize this tag at the beginning of the path lookup. Additionally, we carry out the dynamic rendering by intercepting the kernel function `path_to_nameidata` called at every step of the path lookup to update the rendering result into the 'nameidata' object. The rendering procedure added to this function first compares the current tag in 'nameidata' with the tag in the dentry just discovered, and the tag in the 'nameidata' will be rendered as the one with a lower security level. This rendering is irreversible in each path lookup procedure to ensure that the access from a trusted environment to the untrusted one can be identified. We show examples in Fig. 4: the sub-figures here describe the dynamic rendering in three different types of path lookup procedures, including the access from a host process to the resources in the container and the shared volume, and the access from a container process to the resources in the shared volume. Each arrow in these figures describes the step of finding a dentry and calling the `path_to_nameidata` function, while the color of the arrow shows the security level rendered in the 'nameidata' object. Among these colors, white represents the highest security level ("normal"), shaded yellow represents the second highest level ("shared"), and the lowest security level ("strict") is indicated by solid yellow.

4.4 Access Control Policies

After the rendering, we can enforce the access control determined by certain tag(s). The general access control rules are as follows. Since the container is untrusted, the container process definitely cannot read, write, or execute any file on the host, while the host process cannot execute the files inside the container either. The shared volumes can also be seen as a part of the container filesystem, thus we also need to restrict the host's behavior on those shared volumes.

We have consulted the rule-making guidelines outlined in the Software-based Fault Isolation [54]. In accordance with the principles of data-access and control-flow policy design, two policies have been formulated.

P1 - constraining object-access results. The first rule is to sandbox the path resolution results. As long as the path lookup enters the untrusted scope (no matter whether it starts the lookup from the container or enters the container later), the final return value (result) of the path lookup must be located in this untrusted region.

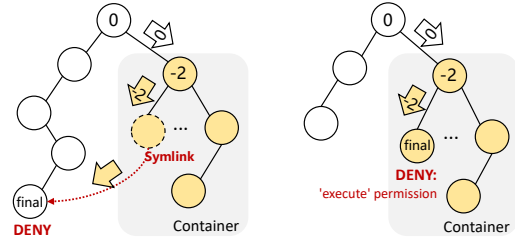


Figure 5: Policy enforcement

P2 - constraining control-transfer behaviors. The second policy is that a control-flow transfer by the untrusted container must stay within the untrusted region. Processes in any trusted regions (which has a higher security level) are not allowed to execute files in untrusted scopes. More specifically, any new processes forked from a container's executable files but actually executing on the host should be blocked since the control has been transferred to the malicious container. The host is identified as a trusted region while all other regions are untrusted to the host, so the files in untrusted container filesystems are not allowed to be executed by the host.

4.5 Path Lookup Interception

Our implementation provides a security monitoring mechanism by inserting certain checks and placing fences in certain kernel functions, making the two policies a whole solution.

We enforce access control policies in the routine of the path lookup, more specifically, in the kernel function `complete_walk`. During the path lookup process, the execution flow must pass through this function. The `complete_walk` is a function to process the final result of path lookup. Therefore, enforcing access control in it can effectively cover the path lookup process without missing any requests.

To enforce P1, Patrol will check whether the tag of the process is consistent with the tag assigned to the dentry in the rendering stage. If the PID namespace pointer *in the process*, the PID namespace tag *in the 'nameidata'*, and the one *in the dentry just obtained* are different, the following fine-grained access-control enforcement is performed in function `complete_walk`. The enforcement first blocks the access if the security level of the process's tag is lower than the tag of the final dentry discovered at the end of the lookup procedure. This could happen when the container process attempts to access a host's file. Further, the access is also blocked when the security level of the final dentry's tag is not shared and is higher than the tag in the 'nameidata' object. In this way, the issue of the symlink resolution cheating (described in Section 3.2) can be prevented by P1. More specifically, the exploit of such vulnerability CVE-2018-15664 will initiate a path lookup to discover the file that should be copied from the container. As shown in Fig. 5 (left), the 'nameidata' object is tagged with the lowest security level after this path lookup enters the container filesystem, while the in-container malicious symlink misleads the path lookup procedure to discover the final dentry which has the highest security level. This outcome aligns with the prohibited situation enforced by P1, which is activated at the end of the path lookup to terminate such illegal access. Further, the accesses from the host to the shared volume but whose result is finally resolved outside the shared volume can be restricted

by P1, since there is an access path from the risky area to the trusted area (possibly via a symlink).

To enforce P2, our defense will perform the execution permission checks of the target file to be opened during the path lookup. To better facilitate the check, we add the access mode flag as a member in the 'nameidata' structure to indicate whether execution would take place after the path lookup. This access mode flag is retrieved and initialized from parameters of the syscall service routines (such as `sys_open` or `sys_execve`) that launch a path lookup. If the PID namespace of the process invoking the syscall service routine and the PID namespace tag in the dentry are different, Patrol starts to check if the P2 is met in `complete_walk`. Once the process has the execution permission and the security level flag of the process is greater than the one in the dentry, access to the dentry is blocked. Fig. 5 (right) depicts how the attack exploiting such vulnerability CVE-2019-14271 can be blocked by P2. During the attack procedure, the Docker process with the highest security level will initiate one path lookup to locate a dynamic-link library that needs to be executed. However, the final dentry discovered has the lowest security level, signifying that this library is within a container. Thus, P2 is enforced at the end of the path lookup to terminate the Docker process that is going to execute a file (e.g., 'nsswitch') with a lower security level.

4.6 Optimizations for Compatibility

To achieve total transparency in the user-space applications, we implement two optimization schemes.

Pivot_root overloading. Since the `pivot_root` syscall is the basic operation to switch into a container's filesystem, Patrol can intercept the 'new_root' parameter [28] from it to indicate the root path of a strict scope. When the syscall is overloaded, the following functions are executed. Starting from the received parameter - the path of the destination root, it recursively traverses the directory entries of all filesystems under it, and uses the strict scope tag (described in Section 4.3) of the current process to mark the related dentry. The above-mentioned overloaded function will take effect only when the process which calls the `pivot_root` syscall is in a different mount namespace than its parent process.

Symlink containing. For better usability and to block the impact of malicious symlinks in our strict scope, Patrol can enforce extra restrictions to make sure all symlinks are resolved under the right root of every strict/shared scope rather than simply block them. The kernel function `trailing_symlink` is also overloaded here for resetting the lookup starting point, which cannot be bypassed since a symlink should and can only be resolved via it.

We add a member 'root_path' in the PID namespace structure, which will store the *root directory* of each scope. When `pivot_root` is called, Patrol sets the root directory of the container rootfs to the 'root_path' variable. In `trailing_symlink`, Patrol obtains the namespace tag in 'nameidata' during the current path lookup process, that is, the pointer to the PID namespace structure; then, resets the *lookup starting point* (described in Section 2.1) according to the 'root_path'. Since the expected goal of our defense is to prevent symlinks from being resolved outside the untrusted scope, we define the following rules: if the symlink is parsed in a container, the lookup starting point is updated to the root path saved according

to the tags; if the symlink in the share volume is parsed from the container, the lookup starting point is still the root path saved; if the symlink in the share volume is parsed from a host process, the starting point is set as the mount point of the share volume.

5 SECURITY ANALYSIS

In this section, we demonstrate that our proposed defense Patrol can enhance the filesystem isolation of containers via formal analysis. Specifically, we model the operations (e.g., syscall service routines [38]) that affect the result of path lookup in VFS as a transition system and leverage Spin (an off-the-shelf model checking tool [35]) to identify the Pamir flaws in the concrete models. Results show that Patrol eliminates all the flaws identified in the current filesystem isolation for containers.

In more specific, the path lookup is the only way to map a path string from user space onto the respective dentry in kernel space by searching the dentry tree in VFS [55]. The start point of this search is decided by the root or current working directory of the process conducting the path lookup. If a dentry is found, the process can access its associated file through the inode pointed by the dentry after the permission check passing. To block illegal accesses between the host and containers, Patrol extends the isolation to the dentries and enforces the access control on the Linux's path resolution routine. To formally evaluate the effectiveness of Patrol, we first establish a general model to describe the state and state transitions that are related to path lookup in VFS (Section 5.1). Then, we implement two concrete models (one with the original path lookup and one with Patrol integrated into the path lookup) based on the general model definition and use Spin to look for flaws in these two models (Section 5.2).

5.1 System Modeling

The state machine model. We model the path lookup-related operations in VFS as a state machine $\mathcal{M} = (\mathcal{V}, \mathcal{A}, \mathcal{S}, \mathcal{T}, s_0)$. Here \mathcal{V} is a set of variables (e.g., *Dentry* see below) that collectively define the state of the system. \mathcal{A} is a set of actions (a.k.a., syscall service routines) that contain a path lookup operation. The execution of each action could change the value(s) of the variable(s) in the variable set \mathcal{V} . \mathcal{S} is a set of states, at each of which certain actions can be formed (e.g., `sys_chroot` and `sys_symlink`). s_0 ($s_0 \in \mathcal{S}$) is the initial state where no action has been performed. \mathcal{T} is a transition function that drives the system from one state to another.

Def. 1. Variable: We define $\mathcal{V} = \{V_{Dtree}, V_{Task}\}$ to record the key VFS status related to path lookup and the status of processes in the system. Specifically, the dentries in the VFS [42], which make up a tree structure (dentry tree), record all files in the system with basic information (e.g., name and hierarchical relation). To model the dentry tree in VFS, we define the variable V_{Dtree} as a set of *Dentries*: $V_{Dtree} = \{Dentry_1, Dentry_2, \dots\}$. Each $Dentry_i$ ($Dentry_i \in V_{Dtree}$) is an abstracted dentry and corresponds to a specific file in the system. Particularly, we use $Dentry_i$ to record the key information of a file, including the name (D_{name}), the hierarchical relation (D_{parent} , D_{child} , and $D_{subdirs}$), and other attributes (D_{flags} , e.g., whether the file is a symlink). As a result, the variable V_{Dtree} could act as the dentry tree to maintain the directory hierarchy and the files' attributes. Moreover, we model the path lookup as simple searches

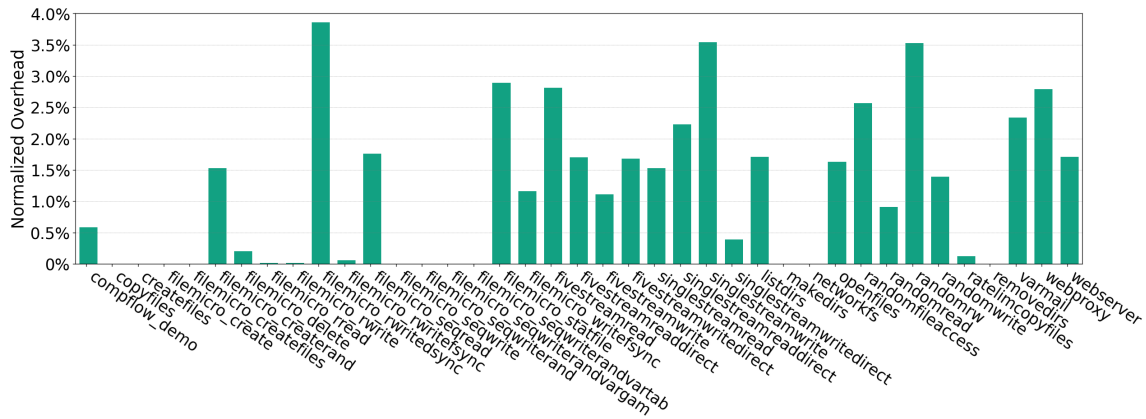


Table 1: Effectiveness results against Pamir attacks

CVE IDs	Crossed scopes	Activated policies
CVE-2017-1002101	0→-1→0	P1
CVE-2018-15664	0→-2→0	P1
CVE-2019-10152	0→-2→0	P1
CVE-2019-14271	0→-2	P2
CVE-2019-18466	0→-2→0	P1
CVE-2019-5736	-2→0	P1
CVE-2021-25741	0→-1→0	P1
CVE-2021-30465	0→-1→0	P1
CVE-2022-0492	0→-1	P2
CVE-2022-1227	0→-2	P2
CVE-2022-23648	0→-2→0	P1
CVE-2023-0778	0→-1→0	P1

should not return any dendry belonging to the host (i.e., $V_{Dtree}.Dentry_j.D_{role} == host$).

- **Security Property 2:** Given any path, any host process (i.e., $V_{Task}.Task_k.T_{role} == host$) should not be able to *execute* any file belonging to the container (i.e., $V_{Dtree}.Dentry_l.D_{role} == host$).
- **Security Property 3:** Under the input of a path mapping to a dendry with the specified role, the path lookup function invoked by a host process (i.e., $V_{Task}.Task_m.T_{role} == host$) should return a dendry having the same role.

Results. The models are implemented in Promela with about 1,500 LoCs. Not surprisingly, Spin identified all the 12 Pamir flaws in the model M_{vul} and reported no flaw in the model M_{SE} after searching all possible states under the constraints of the search depth set for SPIN, 40,000 in our experiment, which shows the effectiveness of our proposed defense.

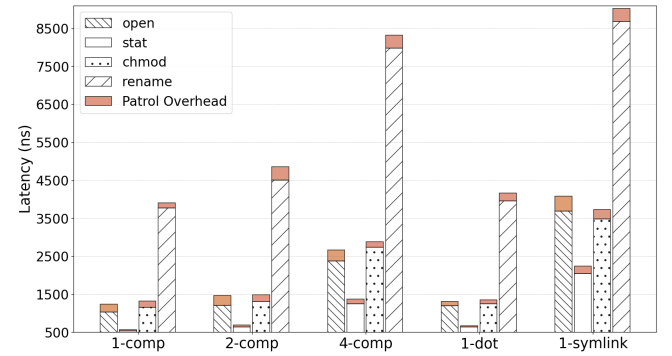
6 EVALUATION

Here we discuss the performance overhead under different level benchmarks Patrol would bear. Our evaluation indicates that Patrol generally exhibits minimal impact on both performance and compatibility and outperforms existing patches.

Settings and references. We evaluated the performance of our prototype with a set of benchmarks and its impacts on the vulnerabilities-related features of the container tools. All our experiments were conducted on Ubuntu 20.04, with the Linux kernel 5.4.1, and the hardware settings include a system with an 8-core 4.50GHz AMD Ryzen 9 7950X CPU and 16GB RAM.

Effectiveness analysis. We first evaluated the effectiveness of Patrol against the attacks that exploit the Pamir vulnerabilities outlined in Section 3.2. In order to observe the response of Patrol to these attacks, we enabled logs in Patrol to monitor which access control policies are activated. Table 1 shows the policies in effect for each Pamir attack and the security scopes crossed by the blocked process during the attacks. Our results demonstrate that Patrol effectively blocks all exploits of the Pamir vulnerabilities. Any process that involves insecure cross-scope accesses will be captured and prevented by Patrol.

Performance of benchmarks. We evaluated the performance impacts of Patrol using Filebench [15] and SPEC 2017 benchmark [34, 45]. The overheads on Filebench and SPEC 2017 are presented in Fig. 6 and Table 2, respectively. Our results indicate that Patrol introduces only minor overhead on file operations in Filebench. Specifically, after running 39 benchmarks on the kernel with Patrol, the overhead on all of them was observed to be below 4.0%, with 30 of them having an overhead below 2.0%. Notably, the overheads on 13 out of 39 benchmarks were almost zero. Additionally, all 43 benchmarks in SPEC 2017 ran successfully on both kernels with and without Patrol, indicating that there was no impact on compatibility. The maximum overhead introduced by Patrol was observed to be 1.7%, with overheads on 38 out of 43 benchmarks being almost zero. The average overhead on these benchmarks is 0.2%, which is negligible.

**Figure 7: Syscall overhead**

Moreover, we use four representative syscalls open, stat, chmod, and rename to evaluate the latency of path lookup [55]. Fig. 7 shows the latencies of these syscalls operating various paths with or without Patrol. These paths have varying lengths ('1-comp', '2-comp', and '4-comp' represent paths /X, /X/Y, and /X/Y/Z/A respectively), or contain a symlink ('1-symlink' represents a symlink /C → /X/Y/Z/C) and parent (dot dot) directory ('1-dot' represents a path /X/Y/. /C). The overheads observed on these syscalls can be primarily attributed to the access controls implemented by Patrol during path lookup. These overheads were found to be less than 0.35 μs and did not exceed 20.7% of the latencies observed for the original syscalls.

Performance of container tools. We next compare the overhead of Patrol to that brought by the vulnerability patches on container tools. We use docker-ce v18.03.1, Kubernetes v1.7.13, and Podman v1.5.1 as the baseline of the evaluation, and the version of runc that is a runtime called by them is v1.0.0-rc5. We selected the earliest versions with known vulnerabilities (discussed in Section 3.2) for these container tools.

According to the design of Patrol, the main overhead is introduced at starting container. The patches for the vulnerabilities CVE-2017-1002101, CVE-2019-5736, CVE-2021-30465, CVE-2021-25741, and CVE-2022-23648 also impact the performance of starting the container along with mounting a shared volume to the container. To evaluate these performance impacts, we sample 120 of

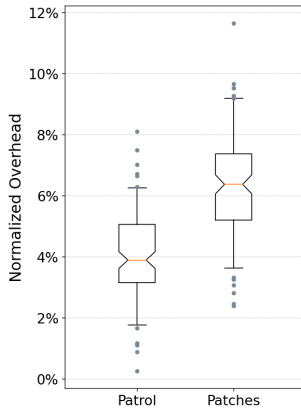
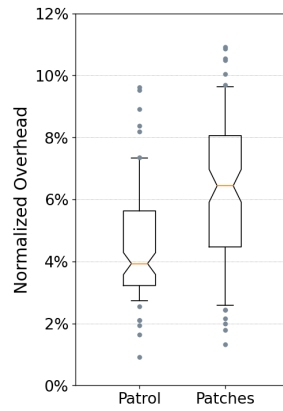
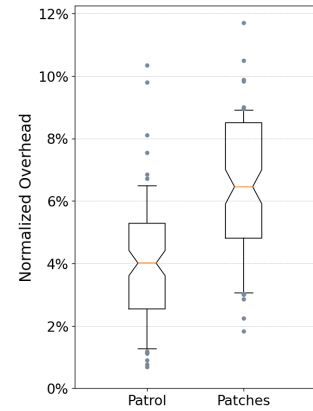
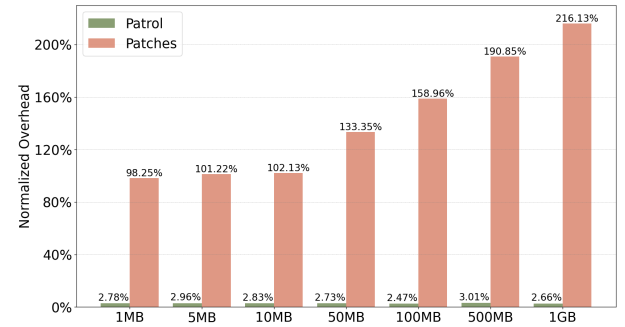
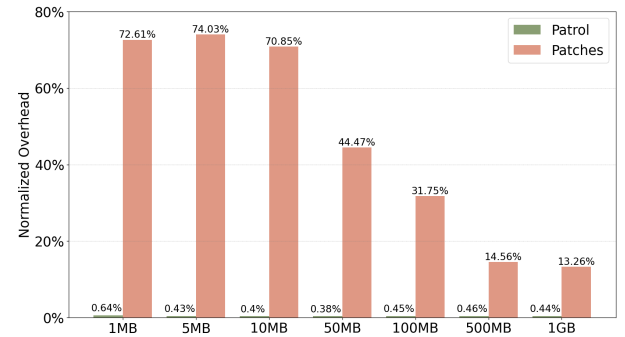
Figure 8: Overhead on *Docker run*Figure 9: Overhead on *Podman run*Figure 10: Overhead on *Kubernetes run*

Table 2: SPEC 2017 overhead

	Benchmarks	Mean	Max	Min
SPEC 2017	43/43	0.2%	1.7%	0.0%

the most prevalent images from the DockerHub [13] and measure the elapsed wall time of starting containers with these images. The starting performed by the container tools uses the ‘volume’ feature, which mounts a shared volume into the container during the starting. Fig. 8, Fig. 9, and Fig. 10 present the distributions of the performance degradation brought by Patrol and patches to Docker, Podman, and Kubernetes, respectively. The lower and upper bounds in the box plots are set to the 5th and 95th percentile in terms of the overheads across running all sampled images, and the dots outside the boxes are outliers. We observe that the median of overheads introduced by Patrol is less than 4% for all these container tools, with the 95th percentile lower than 8%. Among them, Docker meets the lowest overhead with the median at 3.89% and the 95th percentile at 6.26%. In comparison, the overheads caused by the vulnerability patches are higher, with the median being around 6.50%. Specifically, Docker, Podman, and Kubernetes have 95th percentiles of 9.18%, 9.65%, and 8.93%, respectively.

Moreover, the ‘copy’ features of Docker and Podman have been fixed systematically to eliminate such kinds of vulnerabilities, like CVE-2018-15664, CVE-2019-14271, CVE-2019-10152, and CVE-2019-18466. We measure the elapsed wall time of the ‘copy’ features performed by Docker and Podman to evaluate and compare the overheads brought by Patrol and their patches. This test involved copying files of various sizes from the container to the host, with file sizes ranging from 1MB to 1GB. Fig. 11 and Fig. 12 depict the overheads on Docker and Podman, respectively, and demonstrate that Patrol outperforms the existing patches significantly. Specifically, as shown in Fig. 11, the overheads introduced by Patrol are all around 3.0% in copying any size of files, while the overheads from the patches range from 98.25% to 216.13%. The patches to Docker cause the overheads raising with the size of copying files increasing. These overheads come from that Docker has to transmit the copying file out of the container through a pipeline after using sysctl chroot into the container filesystem. Relatively, the patches’ overhead in Podman is fixed, which is brought by pausing and restarting

Figure 11: Overhead on *docker cp*Figure 12: Overhead on *podman cp*

the container during performing copy, but are still far higher than the overheads introduced by Patrol. Specifically, in Fig. 12, we see that the overheads introduced by Patrol are all below 0.7%, which can be considered negligible. In contrast, the overheads introduced by the vulnerability patches range from 13.26% to 72.61%, which is much higher than those introduced by Patrol.

7 DISCUSSION

In this section, we discuss the impacts of the Pamir risk on secure containers and their potential effects on the containers running

on non-Linux platforms, together with the protection a Patrol-like system could provide to them.

Secure containers. Kata and gVisor are two prominent representatives of mainstream secure container technologies: the former utilizes a lightweight *virtual machine* (VM), and the latter relies on a userspace kernel, to enhance the isolation between containers and the host. While the additional isolation layers reduce the attack surface of Pamir vulnerabilities, these vulnerabilities can still be exploited if Kata and gVisor containers share a volume with the host. Specifically, when creating a Kata container, the Kata runtime on the host initially launches a VM that includes a shared volume with the host. Then, the container's rootfs is mounted through this volume by the Kata runtime to enable execution of the container within the VM. However, if an attacker gains control of this VM and replaces the mount point in the shared volume with a malicious symlink, the container's rootfs will be mounted on the host's sensitive path pointed to by the symlink, thus leading to the Pamir risk (CVE-2020-2026). Although gVisor containers typically employ a memory-based virtual filesystem that cannot be accessed directly from the host, the shared volume between the container and the host resides on the host filesystem. As a consequence, the malicious symlink present in this volume would be resolved on the host when the container tool copies this symlink file from the gVisor container to its user, potentially resulting in the disclosure of host files. In addition, we have discovered a new vulnerability of the gVisor container and demonstrated that the Pamir attack can indeed succeed on the container [27].

Generally, the attack surface exploited by the Pamir attacks is the shared volume between the host and these secure containers. Since the shared volume is located on the host filesystem, Patrol on the host can assign a tag with the "shared" security level to the objects within the shared volume when it is created, and enforce access control policies based on these tags. This can preclude the malicious symlink in the shared volume from being resolved outside the "shared" scope, thus eliminating the Pamir risk on Kata and gVisor containers.

Patrol protection beyond Linux. Windows implements its own container technology utilizing kernel features such as *Silo* [6] to isolate processes running on the same kernel, which is similar to the Linux namespace. However, the Pamir vulnerability CVE-2021-24096 is still discovered within the Windows container, which allows a global symlink within the container to be exploited to traverse the host files [37]. We believe that our kernel-based defense has the potential to work on Windows. Unfortunately, given the fact that public documentation on the Windows kernel is limited, a detailed implementation is challenging. Nonetheless, we believe that the high-level designs of Patrol can be extended to the Windows kernel. Specifically, the object manager in Windows is the closest counterpart to the Linux dentry tree, which maintains hierarchical objects (e.g., files, directories, etc.) and permissions. The kernel function `ObpLookupObjectName` handles the pathname resolution, looking up the object in the object manager with a given name. Thus, the Windows kernel can implement the rendering in the object manager by tagging the security level for each object when a container is created. Correspondingly, the path lookup rendering

and the access control could be enforced in the kernel function `ObpLookupObjectName`.

In addition, the kernel of macOS (formerly OS X) does not natively support container technology, so tools like Docker have to launch a Linux guest virtual machine to run the container on macOS. As a result, the Pamir risks are still there and Patrol can provide protection to the containers from the VM kernel on macOS.

8 RELATED WORK

Isolation issues of containers have been studied for a long time. Gao et al. revealed the basic isolation mechanisms of containers are incomplete and vulnerable. Their study [43] reported that *procf*s in containers has incomplete isolation, and many leakage channels from there could be utilized to profile host behaviors. Another study of them [44] disclosed that vulnerable Cgroups isolation allows containers to ignore limitations and exhaust resources in the host. Facing a wave of vulnerabilities breaking container isolation [46, 49], more systematic solutions are expected by academia and industry rather than fixing them individually [51].

The Linux security mechanisms [39] are the first choice to be used to enhance container isolation, but Lin et al. [48] found their interdependence and mutual-influence relationship make configurations hard to be correct. So, Sun et al. [53] implemented a security namespace to virtualize IMA for each container, which can verify the integrity of the container's filesystem independently, but it does not enhance the filesystem isolation and cannot be used to address Pamir. Moreover, secure containers provide strong isolation by using an additional layer(s) of indirection and/or reducing the attack surface. Prominent examples include Kata [20] and gVisor [19]. These approaches, however, still cannot handle the Pamir attacks, even though the attack surfaces have been reduced. Also, rootless containers [33] are proposed by the container community that can be managed without root privilege. But the existing vulnerabilities are still workable for attackers to escape from rootless containers, just the limited privileges could be gained to continue the attack. Fortunately, the community is still working to address the impact of the performance and functionality caused by the rootless containers [41]. Previous work has identified illegal channels inside *Procf*s [40]. To control these channels, sandboxing solutions [52, 56] based on PKU have been proposed, which intercept filesystem-related syscalls. Similar to Patrol, the sandbox analyzes and makes decisions on syscalls to determine whether they should be allowed to execute or blocked. Inspired by these works, we can also initialize our path lookup interceptions through the `prctl` syscall and forward the interception handlers to a userspace monitor according to different policies in our future work.

9 CONCLUSION

In this paper, we report a systematic study on the Pamir risk aroused from incomplete filesystem isolation between the container and host and their interactions. This type of risk poses a significant challenge to mitigation through container tools alone. Our research reveals that the vast obstacle to eliminating the Pamir risk is the third-party components utilized by container tools, but there are no rational solutions to this obstacle in userland. Further, we present a kernel-based filesystem isolation technique called Patrol. The

model checking demonstrates that Patrol eliminates the Pamir risk completely. Our approach only brings a tiny performance penalty to the kernel and container tools, while providing excellent compatibility. Our discoveries and new isolation enhancement have made a step toward better designing the robust isolation for OS-level virtualization.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their insightful comments, and Shepherd for the constructive feedback. Zhi Li is supported by the National Natural Science Foundation of China under Grant No.62202191, and Hai Jin is supported by the National Natural Science Foundation of China under Grant No.62032008.

REFERENCES

- [1] Add continuous integration to your container builds. <https://learn.microsoft.com/en-us/azure-sphere/app-development/continuous-integration>.
- [2] Auto-reloading for /etc/nsswitch.conf. https://sourceware.org/bugzilla/show_bug.cgi?id=12459.
- [3] cgroups(7) — Linux manual page. <https://man7.org/linux/man-pages/man7/cgroups.7.html>.
- [4] chroot(2) — Linux manual page. <https://man7.org/linux/man-pages/man2/chroot.2.html>.
- [5] containerd: An industry-standard container runtime with an emphasis on simplicity, robustness and portability. <https://containerd.io/>.
- [6] Containers vs. virtual machines. <https://learn.microsoft.com/en-us/virtualization/windowscontainers/about/containers-vs-vm>.
- [7] CVE-2017-1002101. <https://kubernetes.io/blog/2018/04/04/fixing-subpath-volume-vulnerability/>.
- [8] CVE-2019-14271 loading of nsswitch based config inside chroot under Glib. <https://github.com/containers/buildah/issues/2740>.
- [9] CVE-2019-5736: Runc uses more memory during start up after the fix. <https://github.com/opencontainers/runc/issues/1980>.
- [10] daemon: pause containers before doing filesystem operations. <https://github.com/moby/moby/pull/39252>.
- [11] Definitions of the actions. <https://sites.google.com/view/container-isolation/paper-appendix/b>.
- [12] Docker: Accelerated, Containerized Application Development. <https://www.docker.com/>.
- [13] Docker Hub. <https://hub.docker.com/>.
- [14] Docker storage drivers. <https://docs.docker.com/storage/storagedriver/select-storage-driver/>.
- [15] FileBench. <http://www.nfsv4bat.org/Documents/nasconf/2004/filebench.pdf>.
- [16] filepath-securejoin. <https://github.com/cyphar/filepath-securejoin>.
- [17] go-callvis. <https://github.com/ofabry/go-callvis>.
- [18] Go Packages. <https://pkg.go.dev/>.
- [19] gVisor: The Container Security Platform. <https://gvisor.dev/>.
- [20] Kata Containers. <https://katacontainers.io/>.
- [21] Kubernetes. <https://kubernetes.io/>.
- [22] Kubernetes Volumes. <https://kubernetes.io/docs/concepts/storage/volumes/>.
- [23] Managing dependencies - The Go Programming Language. <https://go.dev/doc/modules/managing-dependencies>.
- [24] mount(8) — Linux manual page. <https://man7.org/linux/man-pages/man8/mount.8.html>.
- [25] mount_namespaces(7) — Linux manual page. https://man7.org/linux/man-pages/man7/mount_namespaces.7.html.
- [26] opencontainers/runc. <https://github.com/opencontainers/runc>.
- [27] Patrol. <https://github.com/CGCL-codes/Patrol>.
- [28] pivot_root(2) — Linux manual page. https://man7.org/linux/man-pages/man2/pivot_root.2.html.
- [29] Podman. <https://podman.io/>.
- [30] Podman issues about the CVE-2018-15664. <https://github.com/containers/podman/pull/3214>.
- [31] Proposal: path/filepath: addition of SecureJoin helper. <https://github.com/golang/go/issues/20126>.
- [32] Race Condition in crun. <https://security.snyk.io/vuln/SNYK-ORACLE8-CRUN-2585150>.
- [33] Rootless Containers. <https://rootlesscontainers.rs/>.
- [34] SPEC. <http://www.spec.org/index.html>.
- [35] Spin - Formal Verification. <https://spinroot.com/spin/whatispin.html>.
- [36] Third-party dependencies of the container tools. <https://sites.google.com/view/container-isolation/paper-appendix/a>.
- [37] Windows Server Containers Are Open, and Here's How You Can Break Out. <https://unit42.paloaltonetworks.com/windows-server-containers-vulnerabilities/>.
- [38] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. O'Reilly Media, Inc., 2005.
- [39] Thanh Bui. Analysis of docker security. *arXiv preprint arXiv:1501.02967*, 2015.
- [40] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In *Proceedings of 29th USENIX Security Symposium*, pages 1409–1426, 2020.
- [41] Guillaume Everarts de Velp, Etienne Rivière, and Ramin Sadre. Understanding the performance of container execution environments. In *Proceedings of the 6th International Workshop on Container Technologies and Container Clouds*, pages 37–42, 2020.
- [42] Andy Galloway, Gerald Lüttgen, Jan Tobias Mühlberg, and Radu I Siminiceanu. Model-checking the linux virtual file system. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 74–88, 2009.
- [43] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. Containerleaks: Emerging security threats of information leakages in container clouds. In *Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 237–248, 2017.
- [44] Xing Gao, Zhongshu Gu, Zhengfa Li, Hani Jamjoom, and Cong 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*, pages 1073–1086, 2019.
- [45] John L. Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.
- [46] Zhiqiang Jian and Long Chen. A defense method against docker escape attack. In *Proceedings of the 2017 International Conference on Cryptography, Security and Privacy*, pages 142–146, 2017.
- [47] Paul Luo Li, Amy J. Ko, and Jiamin Zhu. What makes a great software engineer? In *Proceedings of the 37th IEEE International Conference on Software Engineering*, pages 700–710, 2015.
- [48] Xin Lin, Lingguang Lei, Yewu Wang, Jiwei Jing, Kun Sun, and Quan Zhou. A measurement study on linux container security: Attacks and countermeasures. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 418–429, 2018.
- [49] Antony Martin, Simone Raponi, Théo Combe, and Roberto Di Pietro. Docker ecosystem-vulnerability analysis. *Computer Communications*, 122:30–43, 2018.
- [50] René Neumann. Using promela in a fully verified executable tlc model checker. In *Proceedings of the 6th International Conference on Verified Software: Theories, Tools and Experiments*, pages 105–114, 2014.
- [51] Elena Reshetova, Janne Karhunen, Thomas Nyman, and N. Asokan. Security of os-level virtualization technologies. In *Proceedings of the 2014 Nordic Conference on Secure IT Systems*, pages 77–93, 2014.
- [52] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. In *Proceedings of the 31st USENIX Security Symposium*, pages 936–952, 2022.
- [53] Yuqiong Sun, David Safford, Mimi Zohar, Dimitrios Pendarakis, Zhongshu Gu, and Trent Jaeger. Security namespace: making linux security frameworks available to containers. In *Proceedings of the 27th USENIX Security Symposium*, pages 1423–1439, 2018.
- [54] Gang Tan. Principles and implementation techniques of software-based fault isolation. *Foundations and Trends in Privacy and Security*, 1(3):137–198, 2017.
- [55] Chia-Che Tsai, Yang Zhan, Jayashree Reddy, Yizheng Jiao, Tao Zhang, and Donald E. Porter. How to get more value from your file system directory cache. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 441–456, 2015.
- [56] Alexios Voulimeas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. You shall not (by) pass! practical, secure, and fast PKU-based sandboxing. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 266–282, 2022.
- [57] Bowen Xu, Le An, Ferdian Thung, Foutse Khomh, and David Lo. Why reinventing the wheels? an empirical study on library reuse and re-implementation. *Empirical Software Engineering*, 25:755–789, 2020.