Journal of Cloud Computing:
Advances, Systems and Applications

RESEARCH

Open Access

Optimus: association-based dynamic system call filtering for container attack surface reduction

Seungyong Yang[1,2], Brent Byunghoon Kang[1] and Jaehyun Nam[3*]

Abstract

While container adoption has witnessed significant growth in facilitating the operation of large-scale applications, this increased attention has also attracted adversaries who exploit numerous vulnerabilities present in contemporary containers. Unfortunately, existing security solutions largely overlooked the need to restrict container access to the shared host kernel, particularly exhibiting critical limitations in enforcing the least privilege for containers during runtime. Hence, we propose Optimus, an automated and comprehensive system that confines container operations and governs their interactions with the host kernel using an association-based system call filtering. Optimus efficiently identifies the essential system calls required by containers and enhances their security posture by dynamically enforcing the minimal set of system calls for each container during runtime. This is achieved through (1) lightweight system call monitoring leveraging eBPF, (2) system call validation via association analysis, and (3) dynamic system call filtering by adopting covert container renewal. Our evaluation shows that Optimus effectively minimizes the necessary system calls for containers while maintaining their serviceability and operational efficiency during runtime.

Keywords Container security, Association analysis, System call, eBPF, Seccomp

Introduction

These days, containers have gained significant traction to operate large-scale applications comprised of microservices effectively. Their widespread adoption by the industry, as evident from the recent survey [13], has solidified their significance in modern computing environments. Container orchestration platforms (e.g., Kubernetes [36] and OpenShift [54]) have further accelerated this trend, enabling seamless automation and scalability of container workloads.

*Correspondence:
Jaehyun Nam
jaehyun.nam@dankook.ac.kr
[1] School of Computing, KAIST, 291 Daehak-ro, Yuseong-gu, Daejeon 34141, Republic of Korea
[2] S2W Inc., 12, Pangyoyeok-ro 192beon-gil, Bundang-gu, Seongnam-si, Gyeonggi-do, Republic of Korea
[3] Department of Computer Engineering, Dankook University, 152, Jukjeon-ro, Suji-gu, Yongin 16890, Republic of Korea

However, the shared kernel-resource model that containers rely on presents significant security challenges, regarding maintaining strong isolation [15] between containers. These concerns are exacerbated by vulnerabilities within legitimate container images [42, 67], providing opportunities for adversaries to exploit weaknesses to breach container isolation. Consequently, unauthorized access to other containers and even the underlying host system becomes possible, posing substantial risks to the overall security posture.

To counter such threats, today's security solutions focus primarily on three aspects: (1) inspecting known vulnerabilities in container images before deployment [24, 53], (2) detecting runtime policy violations within containers [7, 60], and (3) implementing resource restrictions to reduce attack surfaces [34, 55]. However, these solutions often neglect the vital aspect of container interaction with the underlying Linux kernel and leave container environments vulnerable to significant damage,

highlighting the pressing need for a comprehensive security framework that adequately addresses the critical nature of container-to-kernel interactions.

Several studies [19, 30, 68, 70] have explored leveraging Linux's secure computing mode (Seccomp) [63] to accomplish this need by restricting both container-to-kernel interactions and container behavior (filtering invoked system calls). Unfortunately, these approaches also face critical security limitations. First, extracting the minimal sets of system calls required for containers can result in sets that are either too permissive, failing to adequately minimize attack surfaces, or overly restrictive, hindering proper container functionality. Second, the absence of system call validation leaves room for the potential invocation of unauthorized system calls due to adversarial intervention. Lastly, applying the extracted system calls to containers during runtime requires modifications to containerized applications and the underlying host system due to the limitations of Seccomp.

To effectively tackle the security challenges, we propose Optimus, an automated and comprehensive system that performs sustainable identification of the necessary system calls for containers in parallel and continuously enforces the minimal set of required system calls at runtime, ensuring the hardening of containers. Unlike previous approaches that generate a fixed set of allowed system calls for each container, Optimus dynamically adapts to the evolving requirements of containers, maintaining their serviceability while significantly mitigating potential security risks in containers.

Optimus consists of three components: the container manager, the system call monitor, and the profile generator. The container manager oversees the lifecycles of containers and detects any changes in container configurations. The system call monitor tracks all system calls invoked from active containers and accurately segregates the system calls associated with each container by mapping container metadata with relevant system metadata. The profile generator utilizes association analysis on the monitored system calls to generate a tailored set of essential system calls specific to each container, eliminating any ambiguities or uncertain system calls. Finally, the container manager enforces the newly generated system call sets onto the respective containers using covert container renewal. This iterative process ensures comprehensive coverage and support for the operations unexplored within the containers, enhancing the overall security and stability of the containerized environment.

We implement a prototype of Optimus by leveraging the Extended Berkeley Packet Filter (eBPF) and conduct an evaluation using a diverse set of 71 container images obtained from Docker Hub [21]. The results are highly promising: Optimus not only surpasses static analysis-based solutions by effectively filtering out an additional 25% of system calls but also demonstrates impressive resilience, trailing just 1.76% behind dynamic analysis-based solutions. Regarding serviceability, Optimus's covert container renewal leads to only a minimal 1.07% increase in response time, contrasting sharply with Kubernetes's rolling update, which experiences a significant 722% spike in response time when enforcing newly identified system call sets onto containers. Optimus successfully identifies unexplored operations that the dynamic analysis-based solutions fail to capture, highlighting its robustness in detecting previously unknown system call behaviors.

In summary, our contributions include the following:

- Design and implementation of Optimus, an innovative and unified system that continuously monitors system calls invoked from containers and dynamically restricts the available system calls to minimize attack surfaces.
- Development of a novel association-based dynamic system call filtering that validates system calls invoked from containers by analyzing their relationships and filtering out lower-relevance system calls.
- Demonstration of Optimus's capability to reduce the attack surface using real-world container images, along with the identification of system calls for unexplored operations and their dynamic enforcement onto containers at runtime with high serviceability and minimal overhead.

The remainder of this paper is structured as follows: "Background and motivation" section discusses the security challenges in prior work related to attack surface reduction in container security. "Optimus design" and "Implementation" sections introduce our association-based dynamic system call filtering system. The results of security and performance evaluations are presented in "Experimental validation" and "Performance evaluation" sections respectively. "Related work" section reviews related studies, and finally, "Conclusion" section provides the concluding remarks for the paper.

Background and motivation

This section focuses on the key Linux primitives that deliver resource isolation and access control for containers, as well as the security challenges outlined in prior research concerning the reduction of the kernel's attack surface.

The state of container security

Containerization is an operating-system-level virtualization technology that harnesses the powerful isolation

primitives available in the Linux kernel, such as namespaces [64], control groups (cgroups) [65], and capabilities [66]. By leveraging these primitives, containerized applications can operate in virtual environments (i.e., containers) with distinct process trees, file systems, and networking stacks, the illusion of independent and isolated systems, granting applications restricted access to system resources and facilitating highly secure and efficient runtime environments.

Despite the inherent isolation provided by containers, containerized applications are not immune to security vulnerabilities and remain susceptible to targeted attacks [15, 67]. In fact, according to Cloud Native Survey [13], 92% of organizations have employed containers in production, marking a remarkable 300% increase since 2016. However, container security has remained a common area of oversight. The Cloud-Native Security Survey by Aqua Security [5] starkly highlights the lack of awareness among 97% of respondents regarding critical container security principles, leading to misconceptions surrounding default security attributes. These findings emphasize the urgent need to address container security challenges and proactively implement effective measures to reduce attack surfaces.

Containerized applications are legacy applications packaged with containerization techniques, meaning that there is no difference between legacy applications and containerized applications. Any vulnerabilities in legacy applications can exist in the corresponding containers. Lin et al. [42] present that about 56% of the application exploits (e.g., remote code execution and privilege escalation) collected from the Exploit Database [26] are feasible in today's container environments. Furthermore, such vulnerabilities become particularly exploitable within the context of the shared kernel-resource model, which carries profound security implications. These implications are especially acute regarding the capacity of the host OS to sustain isolation when a single container is compromised. For example, container escape attacks [45–48] exploit vulnerabilities in containerized applications to breach the boundaries of container isolation, effectively gaining unauthorized access to the host system.

Attack surface reduction

To protect containers from such attacks, several security solutions have been devised to mitigate the attack surfaces associated with containers. Initial lines of defense include security scanning solutions such as Clair [53] and Docker Scan [24], which proactively scrutinize container images for known vulnerabilities using CVE databases, thus empowering operators to preempt potential threats before container deployment. Complementing this, runtime threat detection solutions (e.g., Tracee [7] and Falco [60]) observe container behavior, identify policy violations during runtime, and conduct anomaly detection, enabling real-time threat mitigation. Additionally, Linux security modules (e.g., AppArmor [34] and SELinux [55]) are deployed to impose further restrictions on containers, managing process executions and file access within containers to fortify container isolation.

However, it is critical to note that these solutions predominantly focus on safeguarding individual containers from a userspace perspective, thereby inadvertently leaving interactions with the host kernel less secured – an aspect capable of causing more comprehensive damage in container environments. Thus, this work emphasizes a crucial yet under-explored aspect of container security: minimizing the attack surface during interactions with the Linux kernel, aiming to contribute towards a more comprehensive security paradigm for containerized environments.

Challenges in attack surface reduction

To diminish attack surfaces, the Linux Secure Computing Mode (Seccomp) [63] emerges as one of the most potent mechanisms widely employed. Seccomp curtails the system calls that applications can initiate, significantly reducing potential attack vectors. The recent Linux kernel offers a vast suite of system calls (approximately 400 syscalls), each capable of becoming an attack vector. However, most applications only engage with a small subset of these system calls. Detecting any unused system calls being invoked could signal a potential compromise, prompting proactive measures to block such actions.

Despite the restrictive power of Seccomp, it necessitates a pre-determined set of system calls that the application is expected to use. In effect, this requires operators to undertake a detailed analysis of the applications, followed by the extraction of the necessary system calls for each, a process that is both intricate and time-consuming. While numerous studies [19, 30, 41] have sought to automate the extraction of system calls, they persistently grapple with substantial limitations in the realm of container security.

False Inference of Necessary System Calls: While extracting necessary system calls to execute containerized applications, a significant challenge presents itself in the form of the false inference problem. The system calls derived from the applications could either prove to be too coarse-grained, thereby not minimizing the attack surfaces of applications as effectively as possible, or too restrictive, potentially hindering the optimal functioning of the applications.

In general, two methodologies can be employed to extract system calls needed for application execution: *static analysis* and *dynamic analysis*. Static analysis-based

approaches [19, 30] examine the source code or binaries of applications, along with all dependencies on external libraries. This allows the collection of all possible system calls that applications might invoke. However, many of these identified system calls may remain unnecessary and unused during runtime, creating potential attack vectors that adversaries can exploit to target the applications and the host kernel. In addition, static analysis does not take into account the specific characteristics of container environments. In particular, it does not adequately cover the process of container creation and initialization before the execution of containerized applications.

On the other hand, dynamic analysis-based approaches [68, 70] identify necessary system calls by monitoring those actually invoked during the rule mining phase, resulting in a significantly reduced set of required system calls for the entire process of container creation, initialization, and execution of applications within containers. However, a potential limitation of dynamic analysis is its dependence on the specific workloads used in the rule mining phase, which may not fully represent the diverse scenarios encountered in a production environment during runtime. Consequently, this approach risks missing critical system calls essential for proper application operation, potentially leading to application failures. Striking the right balance between the advantages of static and dynamic analysis methods is, therefore, a complex and crucial challenge to ensure accurate and reliable system call extraction for containerized environments.

No Validation of Identified System Calls: Another concern arises from the inadvertent detection of unexpected system calls during the extraction of necessary system calls due to the absence of stringent validation measures. This deficiency can result in potentially insecure or unnecessary system calls in the final set, compromising the overall security posture of containerized environments. Adversaries could exploit such vulnerabilities to launch attacks or gain unauthorized access. Thus, it is imperative to implement rigorous validation during the system call extraction.

However, previous studies [19, 30, 68, 70] in system call extraction often lack a comprehensive security mechanism to verify the validity of the identified system calls. This limitation opens the door for adversaries to maliciously manipulate the system call sets to suit their malevolent intentions. For instance, through supply chain attacks [6, 61], attackers could tamper with development environments, coercing developers into using compromised applications or container images. Consequently, when developers seek to identify the necessary system calls for their applications, they may unintentionally include system calls used in container attacks, such as privilege escalation or remote code execution. Moreover, adversaries could directly intervene during the identification phase by deliberately invoking specific system calls necessary for container attacks, as current solutions tend to extract all system calls within containers indiscriminately, regardless of the triggering process. Thus, these malicious system calls could be classified as essential, as the lack of proper validation hinders the accurate identification and exclusion of potentially harmful system calls.

Limitation of Seccomp in profile update: To minimize potential attack surfaces exposed from containerized applications, the initial step involves the application of Seccomp profiles (i.e., the sets of allowable system calls) during container creation. However, a significant challenge arises from the lack of support for modifying and removing already-applied Seccomp profiles during runtime, limiting the adaptability of Seccomp according to evolving security demands.

There are two primary reasons for needing to update existing Seccomp profiles: further restriction and coverage adjustment. When further restriction is required, applications can layer multiple Seccomp profiles, but once attached, these profiles cannot be removed during runtime. Ghavamnia et al. [31] address this limitation by injecting code into applications to stack a new profile on top of the initial one during runtime, achieving the effect of updating the initial profile while imposing additional restrictions on allowable system calls. However, this approach demands access to the application's source code not available in production. Also, it cannot be easily extended to container environments since the modifications of container platforms are not feasible.

On the other hand, when coverage adjustment is needed, Seccomp's inability to modify existing profiles during runtime becomes problematic. Speaker [41] addresses this limitation by introducing a new kernel module into the host kernel, allowing for the replacement of existing Seccomp profiles with updated ones. While effective, this approach requires full privileges on the host system, making it unsuitable for cloud environments with limited privileges. In addition, it may necessitate multiple kernel modules to support heterogeneous host systems, adding complexity to the implementation. As a result, finding a solution that addresses both further restriction and coverage adjustment of Seccomp profiles while considering the constraints of commercial applications and cloud environments remains an ongoing challenge in the pursuit of container security.

Optimus design

This section presents an advanced and automated system, Optimus, for comprehensive container attack surface reduction. In addition, it discusses how Optimus

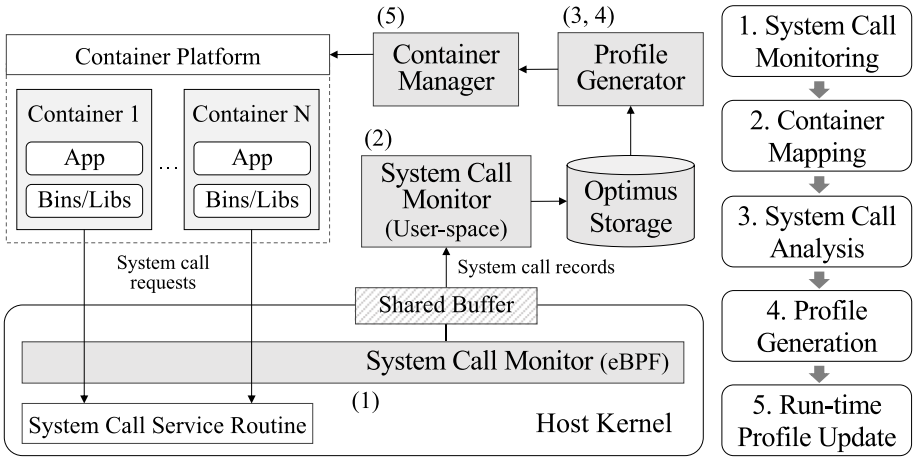

Fig. 1 Overall architecture of Optimus with three main components: the container manager, system call monitor, and profile generator

effectively addresses the limitations outlined in "Challenges in attack surface reduction" section.

Design considerations

To bolster container security and mitigate potential risks as discussed in "Challenges in attack surface reduction" section, three essential requirements have been identified as follows:

R1: *Minimizing attack surfaces in container-to-kernel interactions while not breaking availability.* A system should meticulously determine the necessary system calls for each container, specifically those utilized by the containerized applications, rather than including all possible functionalities. Also, a system should continuously update the set of system calls for each container during runtime to promptly detect any previously undiscovered system calls and maintain an accurate and up-to-date list of essential system calls.

R2: *Validating identified system calls.* To minimize the risk of security breaches, a system should consider the possibility of compromised containerized applications containing unintended system calls and the potential for adversaries to trigger unintended system calls within containers. In the same context, a system should verify the legitimacy of identified system calls.

R3: *Getting through Seccomp limitations.* A system should effectively apply up-to-date Seccomp profiles to containers without requiring container or host system modifications. Also, a system should ensure the stability and availability of containers from the users' perspective while bolstering their security with up-to-date security policies.

Overview

This section presents the overall design of Optimus, an automated and cohesive system aimed at identifying the necessary system calls essential for the seamless operation of each container, including containerized applications. Optimus also enforces a hardened container environment by ensuring a minimal set of required system calls during runtime, all while guaranteeing continuous container serviceability. While Optimus is primarily designed for Kubernetes with Docker as the container runtime, it is not limited to this specific environment. Its design can be applied to various container environments, such as OpenShift [54] with CRI-O [12], expanding its applicability to a broader range of container platforms without compromising effectiveness.

Figure 1 depicts the overall architecture of Optimus, comprising three main components: the container manager, system call monitor, and profile generator. The container manager is responsible for monitoring container-related events, such as creation and removal, and promptly responds by creating or cleaning up corresponding entities in Optimus storage (trace table). Each entity records the system calls associated with a new container, ensuring that all relevant information is efficiently captured. The system call monitor comprises two modules, one in the kernel space and the other in the user space. (1) The kernel-space monitor tracks all system calls triggered from containers, while (2) the user-space monitor periodically fetches these records from the kernel space and stores them in Optimus storage, contextualized with the respective containers. (3) The profile generator then performs an association analysis on the identified system calls for each container and filters out

any abnormal calls that could potentially result from adversarial interventions, ensuring the validation of identified system calls (**R2**). (4) Upon successful validation, the profile generator generates a new Seccomp profile containing an up-to-date set of system calls for the container. (5) The container manager subsequently enforces this new profile to the container, leveraging covert container renewal without requiring modifications to the given containers and host systems (**R3**). To ensure comprehensive system call identification, Optimus repeats the steps outlined above (1-5) to handle potential unexplored system calls (**R1**). The combined operation of these components empowers Optimus to identify necessary system calls sustainably for containerized applications, fortify the container environment with minimal system calls, and maintain container serviceability.

System call monitoring

To accurately identify the essential system calls required by containerized applications (**R1**), rather than including unnecessary functionalities, Optimus's system call monitor takes on the crucial task of collecting all system calls triggered from containers and classifying them according to their respective containers in real-time while ensuring no loss of system calls invoked from them. For this, the system call monitor introduces two pivotal features: lightweight system call monitoring, which prevents any loss of system calls triggered by multiple containers, and container awareness, effectively bridging the gap between system-level and container-level metadata.

Lightweight System Call Monitoring: When it comes to system call monitoring, *strace*[59] and *perf*[62] are popular choices due to their user-friendly interfaces and versatility, particularly in common scenarios such as application debugging. Nevertheless, these tools carry a significant performance overhead as they require frequent transitions between the kernel and user spaces to decode the context of each triggered system call. This inherent drawback makes them less than ideal for continuous monitoring, which represents a fundamental requirement of Optimus (R1). In contrast, Optimus embraces the Extended Berkeley Packet Filter (*eBPF*) [25], a powerful framework enabling the monitoring and tracking of various kernel space activities, including system calls. By harnessing *eBPF*, Optimus efficiently and continuously traces all system calls without imposing significant performance overhead, thus ensuring seamless and effective system call monitoring within the containerized environment.

To identify the system calls triggered by containers, Optimus first employs an *eBPF* program by attaching it at the *raw_syscall* tracepoint, where the control flow of all system calls originates. Then, the *eBPF* program intercepts and captures all system calls, allowing for the extraction of relevant context from the container processes invoking these system calls. As containers are isolated using Linux namespaces, each container process possesses unique IDs within its namespace. Conversely, host processes have a predefined (static) namespace ID (i.e., PROC_PID_INIT_INO). When a system call is invoked, the *eBPF* program captures the event. It then acquires the process context (i.e., PID namespace ID and Process ID) associated with the system call by referencing the *task_struct* structure of the processes, along with the System Call ID. The program then uses the PID namespace ID to determine whether the process invoking the system call belongs to a container. If the system call is not triggered within a container, the program promptly skips further processing for that specific system call. This efficient filtering mechanism ensures that Optimus solely focuses on monitoring and analyzing system calls originating from containers, thus minimizing unnecessary overhead and enhancing the accuracy of system call identification.

As Optimus endeavors to identify a minimal set of necessary system calls essential for container operations, it is crucial to monitor all system calls from the inception of container creation. However, for containers that might already be running before the execution of Optimus, it skips the monitoring of the system calls triggered by those containers since it can only identify a subset of system calls for them. When a new container is created, it undergoes a transition from the default PID namespace to a new PID namespace, resulting in the PID of the initial process becoming 1. Recognizing this characteristic, Optimus utilizes the PID of a process being 1 as a trigger point to commence tracing system calls for new containers. This approach ensures that Optimus efficiently monitors the system calls from newly created containers while avoiding redundant monitoring for existing containers, thereby maintaining the focus on acquiring the complete set of necessary system calls for each container.

Figure 2 illustrates the systematic workflow of Optimus in tracing all system calls for containers. When the initial process (with PID = 1) of a container is launched and invokes a system call, the *eBPF* program initiates the process by registering the PID namespace ID of the container in the *pid_ns* map. Subsequently, whenever a system call is invoked, the *eBPF* program checks the *pid_ns* map to verify if the PID namespace ID associated with the system call exists. If the PID namespace ID is found in the map, indicating that the system call originated from within a container, the *eBPF* program proceeds to record the system call, along with its respective process context, in the shared buffer. This buffer resides between the *eBPF* program and the system call monitor in the

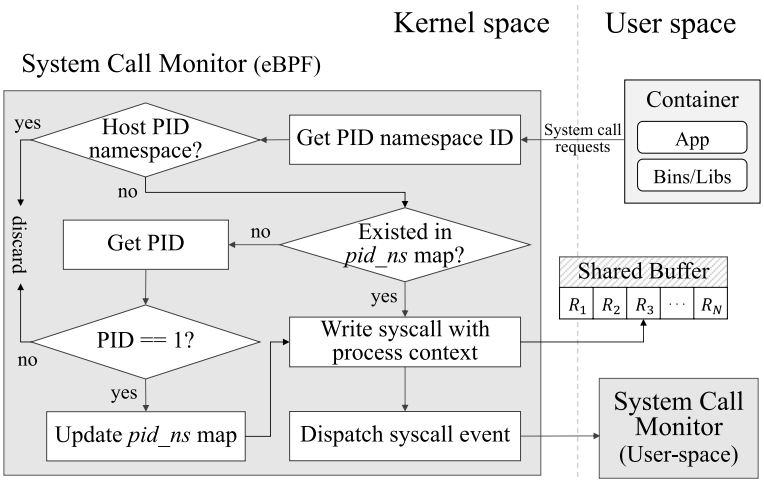

Fig. 2 Overall workflow of lightweight system call monitoring. (1) The system call monitor checks if containers trigger any invoked system calls. (2) It verifies if the invoked system calls need to be monitored based on predefined criteria. (3) The monitor selectively records the required system calls and notifies the update to the user-space monitor

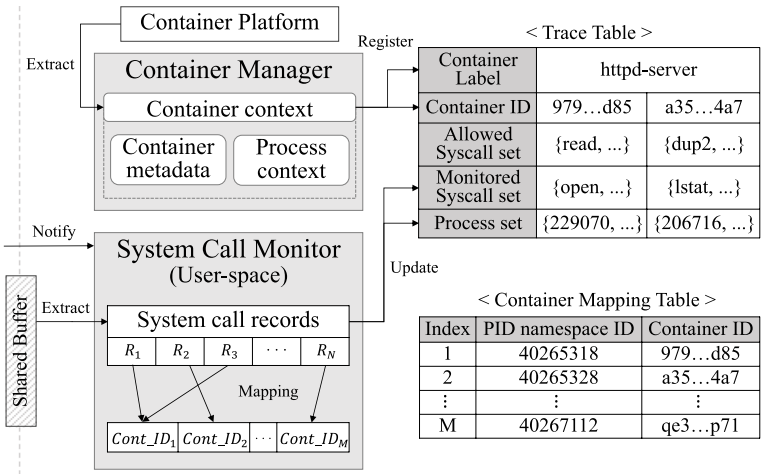

Fig. 3 Overview of container-aware system call recording. (1) The container manager maintains Optimus's trace and container mapping tables by extracting container context from the container platform. (2) The system call monitor updates system call records in the trace table using information obtained from the container mapping table

user space. Conversely, if the PID namespace ID is not found in the *pid_ns* map, signifying that a container did not trigger the system call, the *eBPF* program bypasses monitoring the system call. This approach efficiently ensures that Optimus exclusively traces and captures system calls relevant to containers, not the processes running on the host.

Container Awareness: While the system call monitor can detect system calls made by all container processes, it still has a limited view of which source containers are making these system calls, primarily because container metadata such as Container ID and Container Labels are user-defined and lack inherent context within the host system. This gap between system metadata, including PID namespace ID and Process ID, and container metadata necessitates an additional step to establish a link between the two, effectively bridging this gap and enabling accurate identification and association of system calls with their respective containers.

To bridge the gap between system-level and container-level metadata, as depicted in Fig. 3, the container manager actively monitors container changes within the container platform, maintaining entities for system call records in Optimus's trace table, including context information such as Container ID and Container Label for all active containers. Also, when Optimus detects a new

container, it takes proactive measures to identify the corresponding PID namespace ID for each container by referencing the *proc* file system (specifically, /proc/the PID of the initial process for a container/ns/pid). Then, to effectively associate the system metadata with container metadata, Optimus maintains a container mapping table, which correlates PID namespace IDs with container IDs. Subsequently, when the system call monitor pulls system call records from the shared buffer, it references the container mapping table to find the mapped container IDs associated with the specific PID namespace IDs. This process enables the system call monitor to update the monitored set of system calls accurately for each container in the trace table with the identified system calls, allowing precise profiling of system call activities within individual containers.

Performance Optimization: When monitoring system calls for specific processes, the information on each system call is typically received on a per-system-call basis, as immediate actions or tasks may be required for each call. However, in the containerized environment where numerous containers run on the same host system, there is a significantly higher volume of system calls generated by multiple container processes, even within a short period. Consequently, the communication channel can quickly become overloaded when transferring the information for invoked system calls from the kernel space to the user space through the shared buffer. This can lead to the loss of some system calls due to the lack of space in the shared buffer.

Optimus implements a batch mode to efficiently handle heavy system call invocations while pulling a collection of system call records from the kernel space. As Optimus explicitly requires a set of system calls invoked from each container, it does not need to retain all the information for each system call. By eliminating duplicated system call events already recorded in the shared buffer, Optimus substantially reduces the number of records that need to be transferred to the user space. This intelligent batching approach allows Optimus to pull only a small number of essential system call records and update the trace table without sacrificing the accuracy or completeness of the system call monitoring. The performance benefits of this batch mode and how Optimus effectively resolves the performance issue under heavy system call invocations are elaborated in "Impact on system monitoring" section (Table 3).

System call analysis

As discussed in "Challenges in attack surface reduction" section, the possibility of adversaries intervening in the system call identification process by compromising container images or development environments should be appropriately considered. However, this aspect has received relatively less attention in previous research. Furthermore, determining which system calls should be included or excluded without the context of running containers poses significant challenges, particularly in the containerized environment where multiple tenants deploy diverse container types together. To address the security concerns related to the validation of identified system calls (**R2**), Optimus proposes the implementation of an association-based system call filtering mechanism. This approach is designed to discern and filter out less confident system calls identified within each container, thereby minimizing the potential impact of adversarial interference.

Association Analysis: The primary objective of association analysis [8, 27] is to discover meaningful relationships within given datasets. In the context of Optimus, these relationships manifest as frequent item sets or association rules, representing collections of system calls that frequently co-occur. To validate the relevance of identified system calls for each container independently, Optimus leverages association analysis to uncover relationships among the system calls. By doing so, it identifies patterns of system calls that commonly appear together within the dataset. Subsequently, Optimus filters out system calls that exhibit lower relevance compared to other system calls, effectively reducing the likelihood of including potentially irrelevant or less essential system calls in the final set of necessary system calls. This association-based approach allows Optimus to assess the significance of individual system calls within the context of their co-occurrence patterns, contributing to a more accurate and context-aware system call filtering process.

Association Rule Mining: To uncover relationships among system calls, as illustrated in Fig. 4, the initial step involves establishing association rules for the system calls using the top 100 publicly available container images from Docker Hub [21], deploying them and meticulously collecting the set of system calls invoked by each container process within these deployed containers. Subsequently, the FP-Growth algorithm is employed, a well-known and efficient algorithm [3, 33, 50] for association rule mining that demonstrates effectiveness on large datasets owing to its compact data structure. The generation of association rules is a one-time task, and Optimus leverages the produced rules in the system call filtering process for all containers. By employing this approach, Optimus can effectively discover meaningful relationships among system calls, enabling it to filter out less relevant calls based on their co-occurrence patterns and enhance the accuracy of system call identification for individual containers.

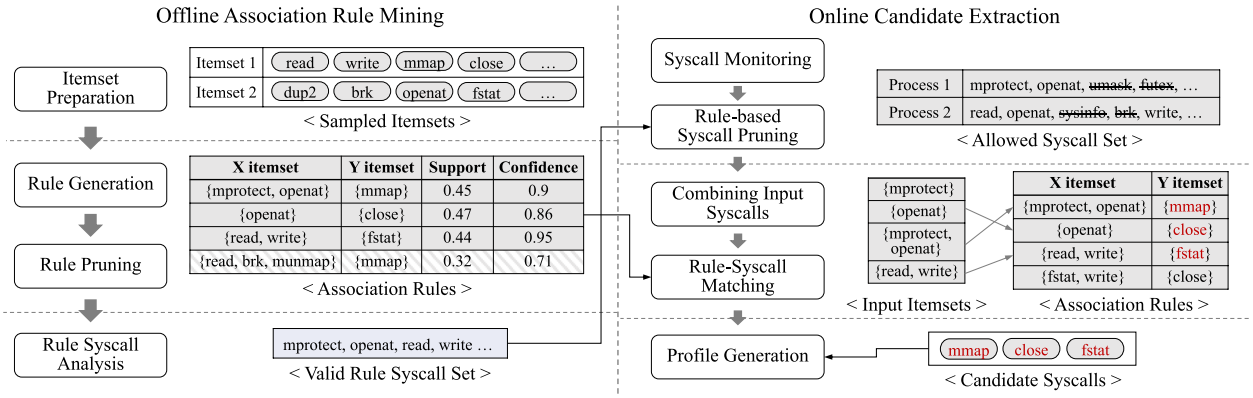

Fig. 4 Procedure to identify highly relevant system calls through association analysis. On the left side, one-time offline processes derive association rules from system calls extracted from diverse container processes, and valid system calls are extracted from these rules. On the right side, the profile generator component utilizes the association rules to identify candidate system calls that exhibit high confidence with the monitored ones

During the process of generating association rules, two crucial parameters, *support* and *confidence*, need to be determined. Support refers to how frequently two system calls, X and Y, appear together in the dataset, while confidence indicates the likelihood of system call X appearing when system call Y is present in the dataset. In this work, the optimal constraints for support (40%) and confidence (80%) are empirically identified, demonstrating a high accuracy score in obtaining sets of highly relevant system calls. Here, the accuracy score represents the measure of correctly identified cases, encompassing true positives and true negatives among all observations. The overall process of determining the optimal parameter set will be elaborated upon in "Effectiveness of association analysis" section, outlining the methodology employed to achieve accurate and reliable system call filtering based on their associations.

The process of generating association rules based on combinations of items in the dataset can lead to a significant increase in rule size, particularly when dealing with diverse items. In this work, since system calls are used as the items for association analysis, in the worst-case scenario, approximately 400 system calls from the recent Linux kernel could be involved. Consequently, the resulting association rules could become excessively large, making it impractical to find relationships among identified system calls for each container in runtime. To tackle this issue, we introduce a novel pruning technique aimed at reducing the number of association rules. An investigation into the accuracy score concerning different numbers of items reveals that the change in accuracy becomes subtle once the number of items exceeds a certain threshold, as depicted in the Fig. 9. Based on this observation, association rules containing more than two system calls are removed, effectively reducing the number of association rules by 94.4%.

Finally, a unique set of system calls included in the association rules is generated. Note that it is possible that some of the system calls identified from containers may not appear in the association rules. In such cases, matching the identified system calls with the association rules would be redundant and consume unnecessary system resources. To optimize this process, a valid set of system calls is created for matching with the association rules, allowing Optimus to filter out identified system calls that cannot be matched during system call validation. This significantly alleviates the workload of profile generation and ensures that Optimus focuses only on relevant and matchable system calls, enhancing the efficiency and effectiveness of the system call filtering.

Seccomp profile generation

Figure 5 illustrates the process of how the profile generator efficiently creates a new Seccomp profile for each container. During this procedure, the profile generator adopts slightly different approaches for the first and subsequent iterations. For the initial creation of the Seccomp profile, the generator derives valid candidate system calls by matching newly monitored system calls with candidates extracted through association analysis. These valid candidates are then combined with the previously allowed system calls to form the basis of the new profile. In subsequent iterations, the profile generator leverages the existing profile and incrementally updates it with newly identified system calls from the association analysis, further refining the profile. This iterative approach optimizes the profile creation process and ensures the

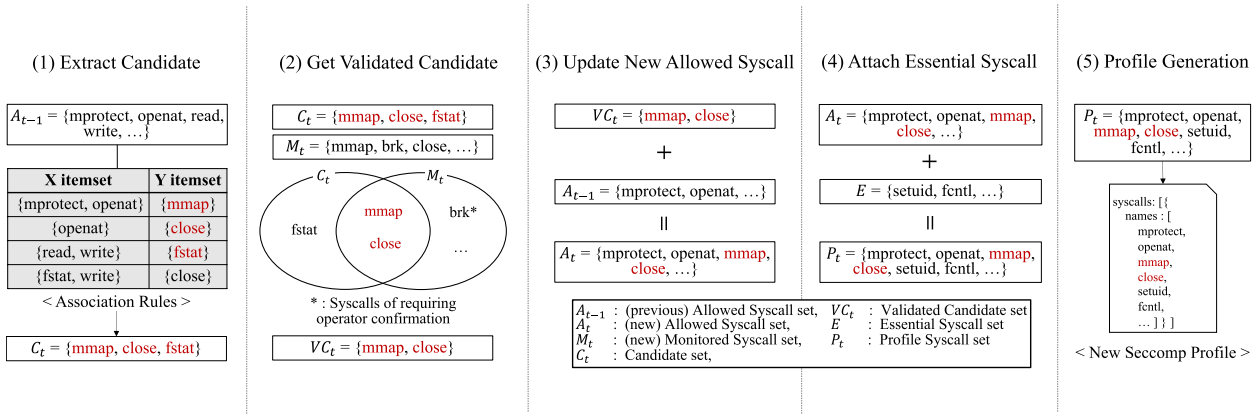

Fig. 5 Process of how Optimus creates new profiles for containers. The profile generator derives valid candidate system calls by matching newly monitored system calls with candidates extracted through association analysis. Combining these valid candidates with previously allowed system calls results in a new set of allowed system calls. The essential system calls required for container initialization are added to this set, culminating in the creation of a new Seccomp profile tailored to the specific container's needs

inclusion of highly relevant and context-aware system calls for each container's specific requirements.

Profile Generation: When operators deploy containers, the containers are initially created with the default Seccomp profile [20], which typically allows most system calls. The profile generator's initial focus lies in the removal of redundant system calls, which are allowed in the default profile but do not actually appear in the container's operations. As the profile generator does not initially possess the system calls required for system call validation, it adopts a pragmatic approach. The generator simply incorporates the system calls monitored within a container into a new Seccomp profile. This strategy effectively reduces the scope of allowed operations for the container, enhancing its security posture.

Once a new Seccomp profile is applied to a container, Seccomp starts restricting unexpected operations with the new profile while allowing most functions necessary for the service of the container. However, it is still possible that certain system calls remain uninvoked until the previous profile generation, potentially representing unexplored operations for the container's service. Therefore, the profile generator shifts its focus to address the addition of undiscovered system calls after the initial profile generation.

Whenever Optimus discovers new system calls in a container, the profile generator is required to update the Seccomp profile applied to the container to accommodate the unexplored operations. To achieve this, the profile generator initiates a validation process for the newly monitored system calls based on the association rules. Specifically, it generates combinations of the system calls already applied to the container while excluding system calls that do not exist in the valid set of system calls derived from the association rules. The profile generator then matches each combination with the association rules, identifying candidate system calls present in the matched rules but not in the set of already-applied system calls. Subsequently, the profile generator checks if each newly monitored system call is among the candidate system calls. If it is, the profile generator deems the newly monitored system call as highly relevant to the system calls invoked until the previous profile generation and automatically includes it in a new Seccomp profile, created with all the system calls previously applied to the container.

However, if a newly monitored system call is not among the candidate system calls, the profile generator temporarily blocks the system call, as it may not be invoked for the container's service. In such cases, the profile generator notifies operators about the unexpected system call and seeks their approval. Unless operators permit the inclusion of this unexpected system call, the profile generator continues to exclude it from the new profile. It is important to note that making decisions in these situations may necessitate the involvement of system security administrators or third-party entities, such as anomaly detection engines. We acknowledge that this aspect goes beyond the scope of our current paper.

Considering the unique behavior of each container image and the various execution scenarios, predicting occurrences of previously unobserved correct executions presents a substantial challenge. Consequently, Optimus consistently engages in decision-making and profile update procedures until the container is terminated, ensuring the availability of the container's intended service. However, if the association analysis fails to identify any candidate system calls with significant associations,

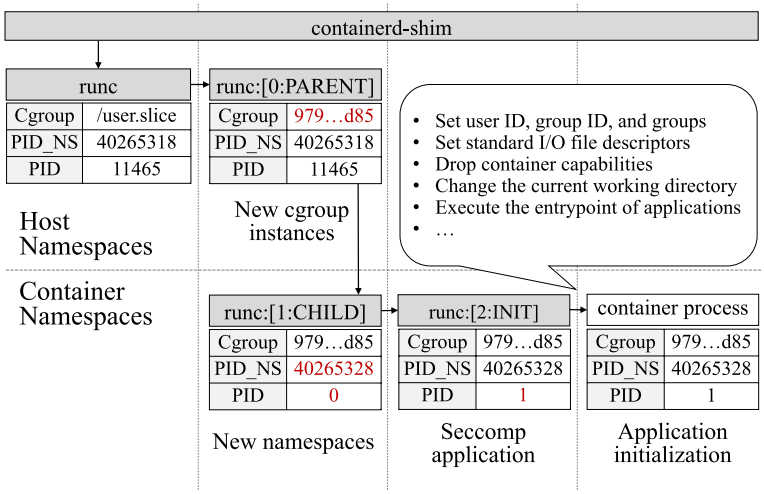

Fig. 6 Workflow of container creation and initialization. Red words highlight the changes in the cgroup, the PID namespace, and the PID

Optimus ceases the process of profile updates. This dynamic and context-aware approach allows the profile generator to adaptively update the profile for each container, ensuring the inclusion of highly relevant system calls while mitigating the potential risks associated with unexplored system call operations.

System Calls for Container Initialization: During the analysis of when a Seccomp profile is applied to a container during the container creation and initialization, it has come to our attention that there exists a hidden space where a Seccomp profile is applied to a container, but the initial process of the container is yet to be executed. This situation poses a challenge as Optimus might miss certain system calls that are required for the proper functioning of running containers. To address this concern and ensure comprehensive coverage, a meticulous manual analysis is conducted using the same container images utilized for association rule generation in "System call analysis" section. This additional analysis allows us to thoroughly identify and incorporate any system calls that might have been overlooked during the initial Seccomp profile generation process, enhancing the accuracy and completeness of Optimus's system call filtering for container environments.

Figure 6 illustrates the container spawning process. When an operator deploys a new container, the container platform initiates the *runc* process, which serves as a low-level container runtime. The *runc* process configures namespaces, cgroups, and capabilities for the container, establishing the container's isolation. Once the container isolation is set up, the *runc* process proceeds to apply a given Seccomp profile (e.g., the default Seccomp profile) to itself. From this point onwards, all operations (i.e., all invoked system calls) within the *runc* process are subject to Seccomp's control. The *runc* process then sets up the container environment, performing tasks like setting user and group IDs, altering the ownership of standard I/O file descriptors, and adjusting the working directory. During this environment setup phase, Optimus might miss particular system calls invoked in this context. Finally, the *runc* process executes the entry point of the container, fully initializing the container environment.

Through our careful analysis, 22 essential system calls[1] required during the container initialization phase have been identified. These essential system calls are fundamental for setting up the container environment and enabling its proper operation. Thus, the profile generator automatically attaches them to the new Seccomp profile for each container at the conclusion of the profile generation process. By including these essential system calls, Optimus ensures the completeness and accuracy of system call filtering, mitigating any gaps that might have arisen during the container initialization.

Covert container renewal

Following the generation of a new Seccomp profile by Optimus for a particular container, it encounters a challenge associated with Seccomp's runtime profile update limitations (**R3**) for active containers, as detailed in "Challenges in attack surface reduction" section. To overcome this constraint, Optimus employs a streamlined approach to dynamically update the Seccomp profile for an active container without necessitating modifications

[1] It is observed that a container invokes setgroups, setuid, setgid, capset, chdir, getdents64, fstat, newfstatat, fstatfs, fcntl, futex, fchown, execve, getppid, prctl, epoll_ctl, epoll_pwait, openat, read, write, close, and nanosleep system calls during the container initialization.

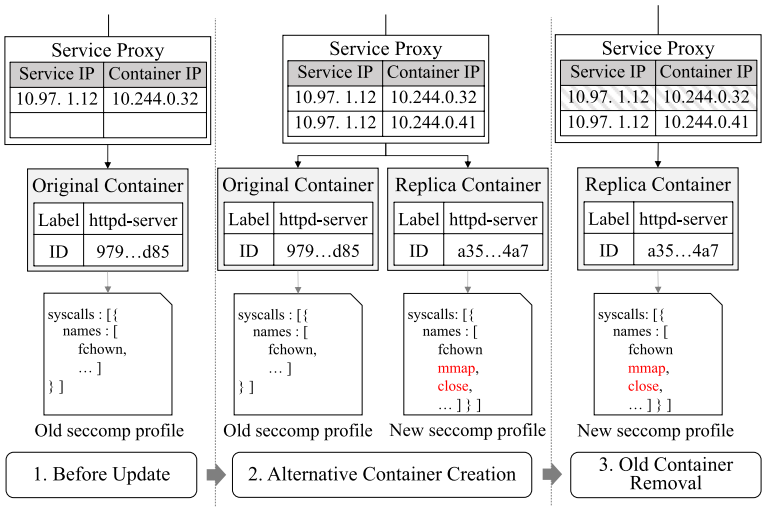

Fig. 7 Workflow of covert container renewal. The container manager creates an identical alternative container to the existing one. Then, it enforces a new Seccomp profile into the alternative container, enhancing its security posture. Lastly, the service proxy seamlessly redirects incoming traffic from the old container to the alternative one

to either the containers or host systems, thereby minimizing disruptions to the container service. This method involves replacing the running container with a new identical container, referred to as an alternative container, which is fortified with the new Seccomp profile. By adopting this approach, Optimus efficiently applies the updated profile to the container while maintaining continuous service availability. The container manager component plays a key role in managing this process, as depicted in Fig. 7.

Alternative Container Creation: Unlike typical applications, containers are designed with scalability in mind, allowing multiple containers (replicas) to serve the same container service. To update the Seccomp profile of a container, Optimus employs an alternative container that is identical to the running container. When the container manager receives the new Seccomp profile for a specific container from the profile generator, it proceeds with the update process by first deploying the new Seccomp profile into the host systems. Additionally, the container manager increases the number of replicas associated with the container service.

The container platform automatically creates a new container, known as the alternative container, equipped with the updated Seccomp profile. By increasing the number of replicas, Optimus ensures the availability of the container service without any interruptions. This approach prevents service downtime that may occur if the container were simply recreated with the new profile. Without adjusting the number of replicas, the container platform might immediately terminate either the original container or the alternative container while attempting to maintain the specified number of replica containers for the container service. This could result in either the container service being unsupported or facing a service interruption until the new container is fully initialized. Therefore, Optimus proactively increases the number of replicas to avoid such potential scenarios and ensures the seamless update of the Seccomp profile with uninterrupted service availability for containers.

Old Container Removal: Upon the readiness of the alternative container, the container platform automatically load-balances incoming traffic between the original and alternative containers. Typically, container platforms group containers based on assigned labels and treat containers with the same labels as endpoints for the same container service.

However, at this stage, it becomes challenging to determine the completion of the container service initialization, as there is no direct context available for the container service. To address this, Optimus relies on the patterns (variations) of invoked system calls from the alternative container. As shown in Fig. 11 ("Managing unexplored operations" section), most containers exhibit a consistent number of system calls once initialization is complete. Conversely, during initialization, the number of system calls generated fluctuates. Optimus capitalizes on this observation by assessing the stability of the container service through the differential of the moving average in the number of system calls over a specific time period. Once Optimus deems the alternative container fully initialized and the container service stable, it proceeds to remove the old container by reducing the number of replicas.

To ensure the efficient termination of the old container, Optimus strategically decreases the priority (deletion cost) of the old container, making it the first selection for termination before modifying the number of replicas. Consequently, the container platform gracefully terminates the old container, directing all incoming traffic to the alternative container, where the updated profile is applied to allow previously unsupported operations. Therefore, Optimus achieves a seamless transition from the old container to the alternative one with the updated Seccomp profile, preserving continuous service availability.

Implementation

The Optimus prototype is implemented using 8.1K lines of Go and C code, running on the Linux 5.4 kernel with Kubernetes v1.21 [14] and Docker v19.03.9 [23].

Container Manager: Optimus leverages Kubernetes's Client-Go [37], which enables interaction with the Kubernetes API server, to monitor container resources across all nodes in the Kubernetes cluster. Using watch events for each pod, Optimus can track container creation, modification, and deletion. In covert container renewal, Optimus utilizes *labels* to categorize containers for the same logical stream, as Kubernetes lacks an inherent method to represent the relationship between containers with different Seccomp profiles.

System Call Monitor: The system call monitor comprises two main components: the system monitor in the user space and the *eBPF* program in the kernel space. Optimus employs the BPF Compiler Collection (BCC) [35] to install and retrieve data from the *eBPF* program in the kernel space. The *eBPF* program is attached to the *raw_syscalls* tracepoint to monitor all system calls. Optimus also utilizes two types of *eBPF maps*: BPF_HASH (pid_ns map) to internally store the PID namespace IDs for the containers and BPF_ARRAY (shared buffer) to store system call records.

Profile Generator: To generate Seccomp profiles for containers in the Kubernetes cluster, the profile generator constructs profiles with three essential fields: *syscalls*, *architectures*, and *defaultAction*. It enumerates the set of derived system calls into *syscalls* with the ALLOW return action, specifies the corresponding *architectures* since system calls can differ by architecture, and configures the *defaultAction* as BLOCK, indicating a whitelist-based Seccomp profile.

Experimental validation

In this section, we assess the effectiveness of Optimus in accurately profiling and extracting the minimum necessary system calls for containers while maintaining support for required operations. The evaluations consist of three main types: association analysis validation, attack surface reduction, and management of unexplored operations. For the experiments, three machines were utilized to create a Kubernetes cluster with Flannel [28]. Each machine was equipped with an Intel E5-2678 CPU (12 cores, 2.5GHz) and 16 GB of RAM.

Effectiveness of association analysis

This section presents the accuracy of Optimus in extracting candidate system calls that are highly relevant and provides experimental results that showcase the identification of appropriate association rules to maximize the performance of association analysis. Also, it verifies the impact of association rules optimized for efficient association analysis.

Optimal Constraints on Association Rule: To attain optimal results in extracting highly relevant candidate system calls, we conducted a comprehensive evaluation of association analysis, employing different metric values for support and confidence with the accuracy score as our performance measure. The accuracy score allowed us to gauge the quality of association rules, assessing the precise identification of cases, including both true positives (where predicted system call occurrences match actual appearances) and true negatives (where predicted system calls, expected to be absent, do not occur). Ensuring the robustness and credibility of our experimental results, we systematically tested each generated association rule, considering various combinations of constraints, across the dataset comprising 71 carefully selected container images among the top 100 publicly available container images from Docker Hub [21], repeating the process 10 times. Note that we excluded 29 container images (e.g., OpenJDK [16], Sentry [17], and Apache Storm [18]) from this dataset. This decision was made to individually configure container images, ensuring they represented a diverse range of workloads independently.

Figure 8 illustrates the accuracy scores of association rules generated through the combination of support constraints ranging from 30% to 50% and confidence constraints from 70% to 90%. Upon examination of the generated association rules under the same support constraint, it is evident that accuracy scores decline for confidence constraints below 80%, attributable to diminished association strength. Interestingly, accuracy scores also decrease when the confidence constraint exceeds 80%, indicating a reduction in the size of extracted candidate system calls. Conversely, under the same confidence constraint level, the analysis reveals that the accuracy score reaches its peak at a 40% support constraint, indicating that the accuracy scores do not exhibit a proportional increase as the constraint level escalates, which can be attributed to the sparser extraction of candidate rules.

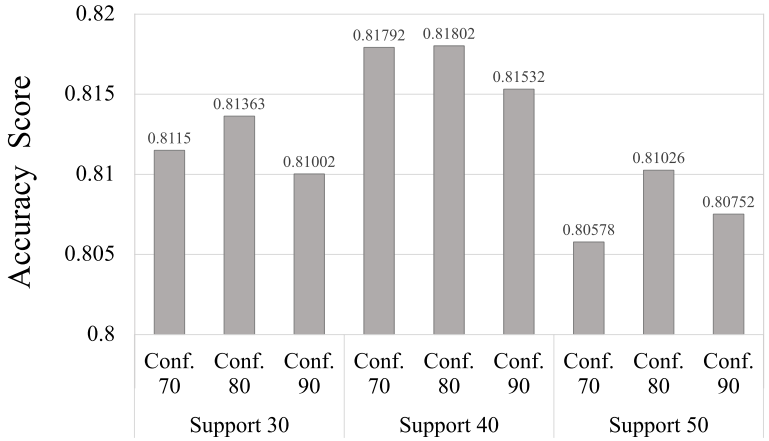

Fig. 8 Accuracy scores of the association rule set extracted under varying support (30% to 50%) and confidence (70% to 90%) constraints

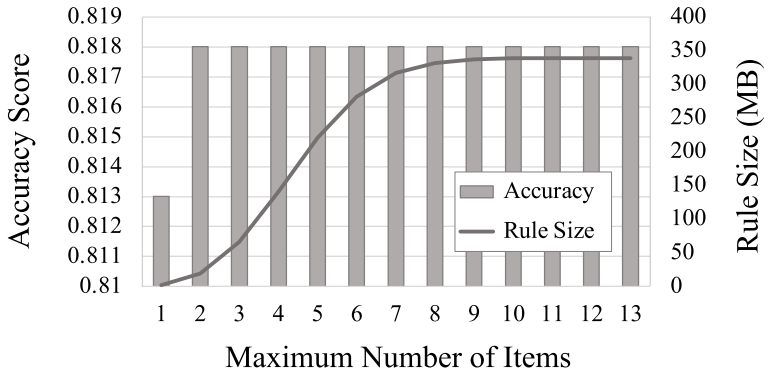

Fig. 9 The accuracy score and rule size of the association rule sets with the different number of items per rule

Performance of Association Analysis: Using the optimal set of association rules, a comprehensive experiment was conducted with 71 container images to assess the performance of our association analysis model in extracting candidate system calls that actually occur during the container's service runtime. The results showed that, on average, 10 out of 11 candidate system calls were observed to occur, representing true positive observations. Conversely, out of 325 system calls that were excluded from the candidates due to their low relevance, 265 system calls did not occur as predicted, indicating true negative observations. This demonstrates that Optimus can accurately predict the occurrence and non-occurrence of system calls with high correlation and low correlation, respectively, achieving an accuracy of 81.8%.

Optimization for Association Analysis: To evaluate the performance of the filtered association rules, we conducted experiments with various rule sets by pruning the original rules based on the number of items in each rule.

Figure 9 shows that as the maximum number of items in a rule increases, the file size of the rule set significantly increases from 1.8 MB to 339 MB. However, the accuracy remains constant at 81.8% for rule sets containing up to 2 items. Notably, the file size of the rule set with less than or equal to 2 items is only 19 MB, which is a substantial reduction of 94.4% compared to the original set's size of 339 MB, while maintaining the same accuracy score. This significant reduction in file size greatly alleviates the workload of matching input system call combinations to the rules, improving efficiency during system call validation.

System call filtering

To evaluate the attack surface reduction achieved by Optimus, we measured the number of system calls disabled by Optimus. Figure 10 displays the results of this assessment for various container images, including MySQL, PostgreSQL, MongoDB, and Redis, which are widely used in practice. The experiments were conducted on a total of 71 container images; nevertheless, the figure exclusively showcases the outcomes for images that were mutually selected from those evaluations in other works. To enhance the reliability of these experiments, we conducted 10 iterations for all association

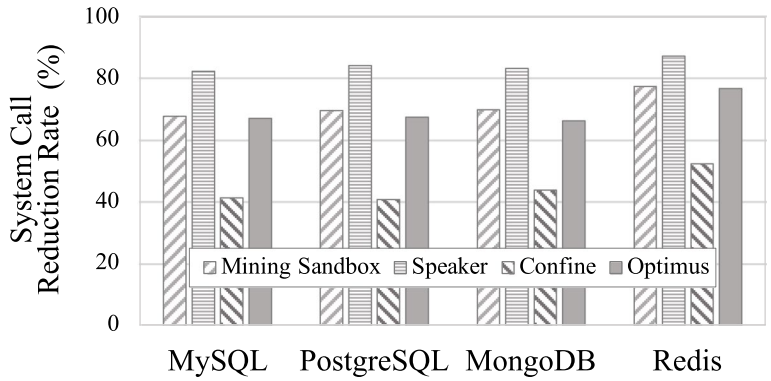

Fig. 10 System call reduction rate achieved by different profiling methods

rules filtered based on the maximum number of items across the 71 selected images.

In the evaluation of attack surface reduction, Optimus demonstrated a significant reduction in the number of allowable system calls for common container images. Specifically, Optimus achieved a system call reduction rate of 69.4%, which is about 25% higher than Confine [30], a static analysis-based approach that extracts allowable system calls by statically analyzing the library functions observed at the beginning of execution. While Mining Sandbox [70], a dynamic analysis-based method that derives the set of allowable system calls through offline training, achieved the highest reduction rate of 71.16%, Optimus's result is only 1.76% lower, making it a competitive solution for minimizing the attack surface. Moreover, Speaker [41], another dynamic analysis-based approach, presented the highest reduction rate of 84.12% since it removes the system calls, which are only required for the booting phase at the start of the running phase. However, Optimus provides an essential advantage over these dynamic analysis-based approaches by supporting the reliable execution of containers even in the case of unexplored operations. This is made possible through Optimus's covert container renewal mechanism, which allows for seamless updates of Seccomp profiles without disrupting container service. Notably, Optimus's evaluation focuses on container-native environments, differentiating it from static analysis-based approaches designed for host-native applications.

To validate the accuracy and reliability of the profiles generated by Optimus, we conducted inspections on the status of containers and application logs for all 71 containers that were applied with profiles corrected by Optimus. The results revealed that all containers were successfully launched without encountering any issues or disruptions. This demonstrates that Optimus effectively creates valid profiles that do not interfere with the essential functionality required for the normal execution of containers. The thorough validation process ensures that the profiles derived by Optimus are robust and dependable, providing an added layer of confidence in the security and stability of containerized applications.

Managing unexplored operations

To evaluate Optimus's capability in handling unexplored operations and exceptional cases, we verified the validity of the profiles generated by Optimus after correcting them to accommodate these intended operations. Additionally, we analyzed the occurrence patterns of system calls to ensure the proper initialization of containerized applications.

Unexplored Operation Discovery: Table 1 provides a detailed description of the exceptional cases introduced to test Optimus's ability to handle unexplored operations.

- **Memory Buffer Bloating.** PostgreSQL encountered memory buffer problems when subjected to significant stress. These issues were traced back to the execution of a large number of insert queries using the *sysbench* utility, which was running within the client-side container. In response to this situation, PostgreSQL dynamically increased the memory buffer size by making use of the *sys_mremap* system call to extend an existing block of virtual memory.
- **Configuration Reload.** Nginx performed configuration reloads, typically triggered by rare events such as SSL certificate updates, the implementation of redirect rules, or adjustments to rate limits. To assess Nginx's configuration reload process, we initiated it by sending a *SIGHUP* signal. The master process conducted checks on the new configuration file's status using the *sys_lstat* system call. Additionally, it employed the *sys_umask* system call to set file creation permissions for log files and new sockets. When

Table 1 Example cases of unexplored operations that do not belong to normal execution paths. Each case exceptionally appears under specific conditions, not simple benchmarks or training. The above system calls are required to perform the required operation successfully

Container Image	Operation that rarely occurs	Required system calls	Reason to invoke system calls
PostgreSQL	Memory Buffer Bloating	mremap	To resize memory space for transaction logs and caching
Nginx	Configuration Reload	lstat	To confirm and parsing the new configuration file
		umask	To open log files and new sockets
		getpgrp	To obtain the PGID of the old worker processes
		kill	To send the SIGTERM signal to the old worker processes
Nginx	Cache Purging	unlink	To delete the old cached files
Apache Httpd	Server Reconfiguration	dup2	To duplicate the file descriptor of the dummy socket
		sysinfo	To get available memory/swap space size
		getpgrp	To obtain the PGID of the old worker processes
		kill, tgkill	To send the SIGTERM signal to the old worker processes

replacing the worker process, the master process utilized the *sys_getpgrp* and *sys_kill* system calls to send a *SIGTERM* signal.

- **Cache Purging.** In the case of Nginx, we also conducted an examination of cache content management in exceptional scenarios. This involved saturating the cache memory by utilizing the *ab* and *curl* utilities. In cases where the requested content was not found in the cache, the cache manager performed a flush operation on the most recently accessed cached content, which in turn triggered the use of the *sys_unlink* system call to remove the cached data.
- **Server Reconfiguration.** Exploration of server reconfiguration, a rarely executed operation by web daemons, was also investigated for Apache httpd. A graceful reload of Apache Httpd was conducted while actively handling traffic generated via the *httperf* utility. Throughout this process, the parent process duplicated the file descriptor of the dummy socket using the *sys_dup2* system call and leveraged the *sys_info* system call to assess the current server memory statistics. To signal the termination of worker processes, the *sys_getpgrp*, *sys_kill*, and *sys_tgkill* system calls were employed. These exceptional scenarios exemplify potential occurrences during runtime.

To validate the effectiveness of the profiles generated by Optimus, we thoroughly examined the status of containers and the application logs when the containers, equipped with the derived profiles, encountered the exceptional cases introduced earlier. Remarkably, all containers fortified with the corrected profiles exhibited flawless performance without encountering any issues or errors during application initialization and when dealing with previously unexplored operations. Here, we confirm that Optimus adeptly manages the occurrence of unexplored operations while effectively minimizing the attack surface exposed to the kernel.

System Call Invocation Pattern: To assess the completion of application initialization, we conducted an in-depth analysis of system call invocation patterns on 71 container images that were subjected to Optimus. Figure 11 illustrates the analysis results for the *plone* container, chosen as a representative case. The column bar graph depicts the number of invoked system calls, measured at 500 ms intervals from the beginning of the container deployment. Notably, there were significant fluctuations in the number of system calls during the first 12 seconds of container deployment. After careful examination of the application logs, we concluded that the containerized application completes its initialization after these fluctuations have subsided. A similar pattern was observed across most container images where Optimus was applied, where either no system calls were issued or a stable number of system calls were intermittently repeated after initialization was completed. These findings affirm Optimus's ability to identify the optimal timing for container renewal, ensuring seamless and reliable execution.

To achieve a comprehensive understanding of variations in the number of system call occurrences during application initialization, a moving average approach was employed. As observed in Fig. 11, this differential value gradually approaches zero after the application's initialization phase. By leveraging moving averages, Optimus was able to ascertain the moment when the containerized application achieved a stable state, fully initialized, and capable of serving traffic.

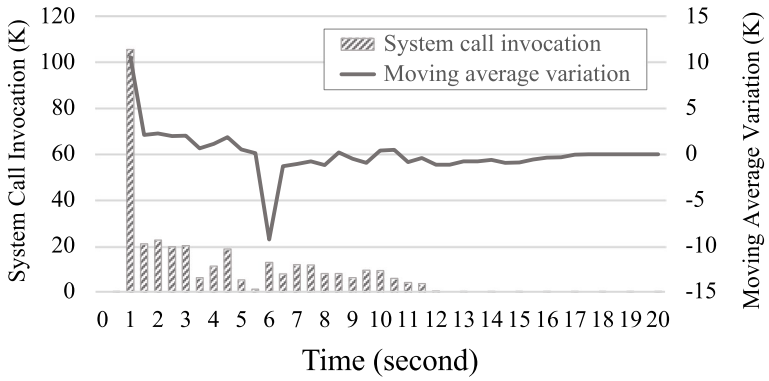

Fig. 11 Measurements on the number of system call invocations and moving averages for estimating the completion of container initialization

Table 2 Measurements of the elapsed time in handling system calls, depending on monitoring schemes: *strace* and *eBPF*

	Processing time (μs)	Increment rate (%)
Base	15.07	-
Strace	18.45 (+3.38)	22.39
eBPF	15.29 (+0.22)	1.41

Table 3 A performance comparison between classic and batch modes of *eBPF* in monitoring system call events. The total events indicate the number of events transferred to the *eBPF map*, whereas the lost events represent what was not processed and wasted occurrences

	Total events	Lost events
Classic Mode	2,579,794	853,583 (33.1%)
Batch Mode	95	0 (0%)

Performance evaluation

Unlike previous studies that rely on offline analysis, Optimus continuously monitors system calls while the container is running, which may impact the performance of the host system and containerized applications. To assess the influence of Optimus on the performance of the host system, we conducted several measurements to evaluate its effect.

Impact on system monitoring

Processing Time by Monitoring Mechanisms: To assess the performance degradation caused by the monitoring mechanism utilized by Optimus, we conducted a series of measurements to measure the processing time for a set of system calls. The measurements were performed 30 times for three different scenarios: the absence of monitoring (Base), the use of the *strace* utility, and the application of *eBPF*. These scenarios were chosen to evaluate the suitability and effectiveness of Optimus.

As shown in Table 2, the *strace* utility exhibited a 22.39% increase in processing time compared to the baseline scenario, where no monitoring mechanisms for system call invocations were employed. This significant increase in processing time is attributed to the need for frequent transitions between user and kernel space for capturing and decoding system calls. The high number of context switches between user and kernel spaces leads to considerable performance degradation on the host system. On the other hand, the *eBPF* approach showed only a 1.41% increase in processing time, which is a significantly more reasonable overhead compared to *strace*. The *eBPF* mechanism operates within the kernel and facilitates interactions with user space through the *eBPF map*, which is accessible from both spaces. As a result, there is no need for frequent context switching between spaces to trace system calls. Based on the system call processing time measurement results, we conclude that *eBPF* is a suitable and high-performance approach for system call monitoring.

Effectiveness of Batch Monitoring: In scenarios where the container execution involves a vast number of system calls, a new challenge arises in accurately identifying and handling these system calls. This is due to the events getting heavily stacked in the *eBPF map* and the non-atomic handling of events. We have devised an optimized batch mode to overcome this issue and reduce the performance degradation caused by the basic process of the classic *eBPF*.

We measured the number of (monitored and lost) system call events in containers to assess the batch mode's performance benefits. Table 3 displays the results of event invocations when the classic and batch modes were

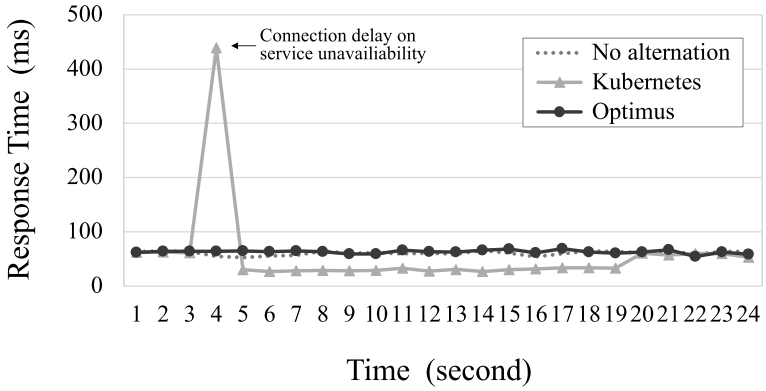

Fig. 12 Time series of response time variations during the Kubernetes-native Rolling Update and Covert Container Renewal

applied, respectively. The classic *eBPF* generated 2.5 million events that had to be processed in user space. In contrast, the batch version of *eBPF* significantly reduced the number of events by 99.99%, resulting in only 95 events. Moreover, 33.1% of all events in the classic *eBPF* were discarded without processing due to the *eBPF map* being already full of unprocessed events. In contrast, no events were lost in the batch mode, thanks to the significant reduction in total events. The dramatic reduction in the scale of events in user space significantly mitigated performance degradation, addressing potential strain on host system resources, particularly in environments with numerous active containers.

Impact on container applications

To assess the impact of the Covert Container Renewal on the containerized application's service availability, we measured the response time and the rate of failed requests. The goal was to evaluate whether the renewal process has any noticeable effect on the performance and reliability of the containerized application's service.

Response Time on Container Renewal: We deployed Apache Httpd with accessible web pages on the server-side container and used another container as a client to send requests using *ab* [4]. We then recorded the response time while the server-side container was replaced with a new one through different container alternation approaches. Response time refers to the duration it takes for a system to process and respond to a client's request, encompassing the round-trip time from client to server and back.

Figure 12 shows the response time of requests through the respective container alternation approach. When there was no container alternation, the average response time was 60.83 ms. However, with the Kubernetes-native Rolling Update approach, the response time spiked to 439.14 ms at the start of the container alternation (at 4 seconds). This increase in latency was due to the abrupt termination of the server-side container before the new container's application had fully initialized, leading to connection establishment issues and delays in responding to requests. Additionally, during the period between 5 and 19 seconds, the requests experienced lower response time than the container with no alternation because the server-side container was busy with initialization and unable to respond promptly.

On the other hand, Optimus's Covert Container Renewal approach yielded an average response time of 61.48 ms, only a 1.07% increase compared to the container with no alternation, and displayed a similar response time pattern. This is because Optimus keeps the old container on the server side until the new container's application completes its initialization process, ensuring that the service remains available during the renewal process, unlike the Kubernetes-native Rolling Update, which causes disruptions in service availability during the container alternation.

Request Failure Rate: In the evaluation of request failure rates during container alternation at various request concurrency levels, Fig. 13 shows the results for both the Kubernetes Rolling Update and Optimus's Covert Container Renewal. In this context, the request failure rate denotes the proportion of requests sent to a containerized application that either lacks a valid response or receives an error response. At a concurrency level of 4,000, the Kubernetes Rolling Update exhibited a loss rate of 0.43%, indicating the highest increase of up to 1,076% compared to the container without alternations. On average, the request failure rate during the Kubernetes Rolling Update was 797.69% higher than that of the container without alternations.

In contrast, Optimus's Covert Container Renewal displayed a loss rate of 0.05% at the 2,500 concurrency level, resulting in a decrease of -27.98% compared to the

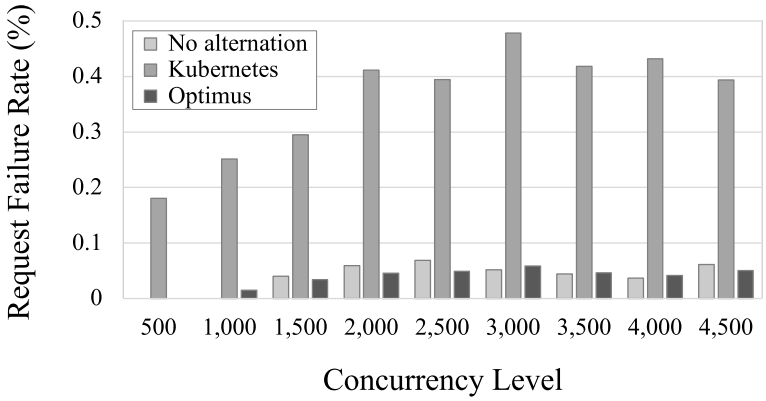

Fig. 13 Request failure variations during container alternation with an increment in concurrent connections

container without alternations. On average, Optimus achieved a slightly lower request failure rate of 5.94% compared to the container without alternations. This improvement is attributed to the divided burden on the server during the coexistence period of the old and alternative containers, which helps alleviate the request failure rate when the server is under heavy load.

With these results, Optimus effectively updates the applied profile during container execution while ensuring the high availability of the application within the container. The Covert Container Renewal approach mitigates the service disruption issues observed in the Kubernetes-native Rolling Update, allowing for smoother and more reliable container alternation without breaking the application's availability.

Related work

Application debloating: One approach to reduce attack surfaces is removing the unused parts of code from the application memory space. They achieve this through techniques such as library specialization [49, 52], function call graph analysis [2], data dependency analysis [51, 58], argument-level specialization [44], and user-defined feature analysis [56]. These approaches reorganize programs or libraries to minimize the code that is loaded and executed, thereby reducing potential attack vectors. In contrast, Optimus addresses the security of containerized applications by restricting the interactions with the Linux kernel through dynamic and association-based system call filtering. Rather than modifying the application code itself, Optimus monitors system calls at runtime and dynamically enforces a tailored Seccomp profile to limit the available system calls, effectively minimizing the exposed attack surface to the Linux kernel.

Kernel debloating: Several works [1, 32, 38–40, 73] have focused on reducing the attack surface of applications by minimizing kernel memory space. These approaches involve tailoring the Linux kernel to specific workloads [40], instrumenting kernel functions to identify and remove unused code sections [38, 39], generating customized kernel profiles for individual applications [32, 73], and conducting dynamic switching of in-memory kernel code based on application profiles [1]. While these methods share the goal of minimizing the kernel's exposure to potential attacks, our approach in this research centers on securing interactions with the Linux kernel by restricting access to system calls, rather than customizing the kernel for individual containers.

System call restriction: Several studies have explored the use of system call restriction for reducing the attack surface of applications.

Static analysis-based approaches. Sysfilter [19] employs static binary analysis to identify necessary system calls from library functions that have a dependency on a given application, creating a tailored Seccomp profile for system call restriction. TAILOR [71] also determines the required system calls for applications by conducting a thorough static analysis of the standard library at the source code level. Saphire [10] identifies API functions of the PHP interpreter and captures the system calls invoked from these functions to create a restricted profile. While research employing static analysis techniques has produced a comprehensive whitelist of system calls, Optimus diverges by tracking system calls during actual runtime. This approach minimizes potentially risky or unnecessary system calls, thus reducing the superfluous attack surface.

Dynamic analysis-based approaches. Wan et al. [68, 70] utilize test suites to train the execution of containerized applications and dynamically record accessed system calls using the *sysdig* [9] trace logs. DockerSlim [22] is a dynamic analysis tool that optimizes containers by removing unnecessary parts from container images and automatically generates a custom Seccomp

profile for the container. However, in dynamic analysis-based research, the generation of system call profiles relies on specific workloads, limiting its ability to adapt to various scenarios encountered throughout the entire execution of a container. In contrast, our system can support the additional system calls required by container applications during actual runtime through covert container renewal.

Hybrid approaches. Confine [30] introduces a dynamic method to capture executables launched during the initial configuration time and performs static analysis on these captured executables to reduce unnecessary system calls. Similarly, RSDS [69] obtains the executed ELF files by monitoring events on the host file system corresponding to each layer of a container image using *inotify* [43]. Canella et al. [11] employ optional dynamic analysis to complement system calls missed from static analysis. Nimos [57] utilizes both static and dynamic analysis techniques to scrutinize the sequence of system call occurrences in kernel exploit codes, utilizing them as attack patterns. However, attempts to merge static and dynamic analysis methods in such research endeavors have encountered pitfalls inherent in each approach, such as erroneous inferences about necessary system calls. In contrast, our system dynamically alters profiles at runtime, thereby minimizing the exposed attack surface while supporting the necessary system calls for the proper execution of container applications.

Temporal approaches. Some studies use temporal separation to apply different policies to applications at different times. Ghavamnia et al. [31] identify distinct initialization and serving phases during an application's execution time and enforce different system call policies for each phase based on static analysis. Speaker [41] presents a similar approach but utilizes dynamic analysis to identify the required system calls for the initialization and run-time phases. Similarly, Yunlong et al. [72] propose a methodology for partitioning the container execution lifecycle into three distinct phases: booting, running, and shutdown. Subsequently, profiles, crafted through dynamic analysis, are strategically applied to the respective phase. SysXCHG [29] introduces an innovative system call filter model by augmenting the existing seccomp-BPF and integrating it into individual ELF binaries. This approach enables the refinement of the permitted syscall set dynamically at runtime, precisely at the point of execution of execve. Research utilizing temporal separation of profile application, similar to Optimus, has innovated new approaches to system call policy enforcement, enabling dynamic replacement with appropriate system call policies at runtime. However, inevitable modifications to the target binaries or kernel pose compatibility issues with existing operating systems.

Conclusion

In the realm of securing containerized applications, there has been a lack of focus on restricting container access to the shared kernel of the host system. Existing approaches to limit container operations and interactions with the host kernel have encountered significant security challenges. To address this gap, we propose Optimus, an automated and unified system that employs association-based dynamic system call filtering in container environments. Optimus utilizes eBPF to monitor all system calls invoked from containers at the kernel level, applies association analysis to filter out irrelevant system calls for each container, and enforces runtime restrictions on available system calls. Through evaluations with real-world container images, we demonstrate that Optimus effectively reduces necessary system calls for containers during runtime, while ensuring continuous container serviceability.

Authors' contributions
S.Y. and J.N. conceived the presented idea and designed the system. S.Y. carried out the implementation and performed the experiments. S.Y. and J.N. interpreted the results and wrote the manuscript. S.Y., B.K., and J.N. reviewed and contributed to the final manuscript. J.N. supervised the work.

Funding
This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ICAN (ICT Challenge and Advanced Network of HRD) support program (IITP-2024-RS-2023-00259867) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation).

Availability of data and materials
Not applicable.

Declarations

Ethics approval and consent to participate
Not applicable.

Competing interests
The authors declare no competing interests.

Received: 10 October 2023 Accepted: 16 March 2024
Published online: 23 March 2024

References

1. Abubakar M, Ahmad A, Fonseca P, Xu D (2021) Shard: fine-grained kernel specialization with context-aware hardening. In: Proceedings of the Security Symposium, USENIX
2. Agadakos I, Jin D, Williams-King D, Kemerlis VP, Portokalidis G (2019) Nibbler: Debloating binary shared libraries. In: Proceedings of the Annual Computer Security Applications Conference, ACM
3. Agrawal R, Srikant R, et al (1994) Fast algorithms for mining association rules. In: Proceedings of the International Conference on Very Large Data Bases, Citeseer
4. Apache Group (1997) Apache http benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html. Accessed 03/2021.
5. Aqua Security. 2021 cloud native security survey reveals runtime knowledge gap. https://www.aquasec.com/news/cloud-native-runtime-security-survey. Accessed 05/2021.

6. Aqua Security (2019) Supply chain attacks using container images. https://blog.aquasec.com/supply-chain-threats-using-container-images. Accessed 06/2021.
7. Aqua Security (2019) Tracee: Runtime security and forensics using ebpf. https://github.com/aquasecurity/tracee. Accessed 10/2020.
8. Baralis E, Cagliero L, Cerquitelli T, Garza P (2012) Generalized association rule mining with constraints. Inf Sci 194:68–84
9. Borello G (2015) System and application monitoring and troubleshooting with sysdig. In: Proceedings of the Conference on Large Installation System Administration, USENIX
10. Bulekov A, Jahanshahi R, Egele M (2021) Saphire: Sandboxing PHP Applications with Tailored System Call Allowlists. In: Proceedings of the Security Symposium, USENIX
11. Canella C, Werner M, Gruss D, Schwarz M (2021) Automating seccomp filter generation for linux applications. In: Proceedings of the Workshop on Cloud Computing Security, ACM
12. CNCF (2016) CRI-O. https://github.com/cri-o/cri-o. Accessed 02/2022.
13. CNCF (2020) Cloud native survey 2020 Containers in production jump 300% from our first survey. https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf. Accessed 06/2022.
14. CNCF (2021) Kubernetes v1.21. https://github.com/kubernetes/kubernetes/tree/v1.21.0. Accessed 07/2022.
15. Combe T, Martin A, Di Pietro R (2016) To docker or not to docker: a security perspective. IEEE Cloud Comput 3(5):54–62
16. Community TD. Openjdk. https://hub.docker.com/_/openjdk. Accessed 11/2021.
17. Community TD. Sentry. https://hub.docker.com/_/sentry. Accessed 12/2021.
18. Community TD. Apache storm. https://hub.docker.com/_/storm. Accessed 11/2021.
19. DeMarinis N, Williams-King K, Jin D, Fonseca R, Kemerlis VP (2020) Sysfilter: Automated system call filtering for commodity software. In: Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses
20. Docker. Default seccomp profile. https://docs.docker.com/engine/security/seccomp/. Accessed 10/2020.
21. Docker (2014) Docker hub. https://hub.docker.com/. Accessed 01/2021.
22. Docker (2016) DockerSlim. https://github.com/docker-slim/docker-slim. Accessed 01/2021.
23. Docker (2019) Docker v19.03.9. https://github.com/moby/moby/tree/v19.03.9. Accessed 02/2021.
24. Docker (2020) Docker scan. https://github.com/docker/scan-cli-plugin. Accessed 02/2021.
25. eBPF Foundation. What is ebpf? https://ebpf.foundation/what-is-ebpf/. Accessed 09/2020.
26. Exploit Database (2010) About the exploit database. https://www.exploit-db.com/about-exploit-db. Accessed 03/2022.
27. Fayyad U, Piatetsky-Shapiro G, Smyth P (1996) From data mining to knowledge discovery in databases. AI Mag 17(3):37
28. Flannel Maintainer Community (2016) Flannel overlay network. https://github.com/flannel-io/flannel. Accessed 06/2021.
29. Gaidis AJ, Atlidakis V, Kemerlis VP (2023) Sysxchg: Refining privilege with adaptive system call filters. In: Proceedings of the Conference on Computer and Communications Security, ACM, p 1964–1978
30. Ghavamnia S, Palit T, Benameur A, Polychronakis M (2020) Confine: Automated system call policy generation for container attack surface reduction. In: Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses
31. Ghavamnia S, Palit T, Mishra S, Polychronakis M (2020) Temporal system call specialization for attack surface reduction. In: Proceedings of the Security Symposium, USENIX
32. Gu Z, Saltaformaggio B, Zhang X, Xu D (2014) Face-change: Application-driven dynamic kernel view switching in a virtual machine. In: Proceedings of the International Conference on Dependable Systems and Networks, IEEE
33. Han J, Pei J, Yin Y, Mao R (2004) Mining frequent patterns without candidate generation: A frequent-pattern tree approach. Data Min Knowl Disc 8(1):53–87
34. Immunix (1998) AppArmor. https://gitlab.com/apparmor/apparmor. Accessed 09/2021.
35. IO Visor Project (2016) Bpf compiler collection (bcc). https://github.com/iovisor/bcc. Accessed 07/2021.
36. Kubernetes. Production-Grade Container Orchestration. https://kubernetes.io. Accessed 11/2021.
37. Kubernetes (2017) Client-go: official client library for interacting with a kubernetes cluster. https://github.com/kubernetes/client-go. Accessed 12/2020.
38. Kurmus A, Dechand S, Kapitza R (2014) Quantifiable run-time kernel attack surface reduction. In: Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Springer
39. Kurmus A, Sorniotti A, Kapitza R (2011) Attack surface reduction for commodity os kernels: trimmed garden plants may attract less bugs. In: Proceedings of the European Workshop on System Security, ACM
40. Kurmus A, Tartler R, Dorneanu D, Heinloth B, Rothberg V, Ruprecht A, Schröder-Preikschat W, Lohmann D, Kapitza R (2013) Attack surface metrics and automated compile-time os kernel tailoring. In: Proceedings of The Network and Distributed System Security Symposium
41. Lei L, Sun J, Sun K, Shenefiel C, Ma R, Wang Y, Li Q (2017) Speaker: Split-phase execution of application containers. In: Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Springer
42. Lin X, Lei L, Wang Y, Jing J, Sun K, Zhou Q (2018) A measurement study on linux container security: Attacks and countermeasures. In: Proceedings of the Annual Computer Security Applications Conference, ACM
43. Love R (2005) Kernel korner: Intro to inotify. Linux J 2005(139):8
44. Mishra S, Polychronakis M (2018) Shredder: Breaking exploits through api specialization. In: Proceedings of the Annual Computer Security Applications Conference, ACM
45. MITRE Corporation (2017) Cve-2017-7308. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7308. Accessed 01/2022.
46. MITRE Corporation (2019) Cve-2019-14271. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-14271. Accessed 01/2022.
47. MITRE Corporation (2019) Cve-2019-5736. https://nvd.nist.gov/vuln/detail/CVE-2019-5736. Accessed 01/2022.
48. MITRE Corporation (2020) Cve-2020-14386. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-14386. Accessed 01/2022.
49. Mulliner C, Neugschwandtner M (2015) Breaking payloads with runtime code stripping and image freezing. In: Proceedings of the International Conference, Black Hat USA
50. Pearce C (2018) An implementation of FP-Growth algorithm. https://github.com/cpearce/arm-go. Accessed 02/2022.
51. Porter C, Mururu G, Barua P, Pande S (2020) Blankit library debloating: Getting what you want instead of cutting what you don't. In: Proceedings of the Conference on Programming Language Design and Implementation, ACM
52. Quach A, Prakash A, Yan L (2018) Debloating software through piece-wise compilation and loading. In: Proceedings of the Security Symposium, USENIX
53. Quay (2020) Clair. https://github.com/quay/clair. Accessed 04/2021.
54. RedHat. Openshift. https://github.com/openshift. Accessed 08/2021.
55. RedHat (2000) SELinux. https://github.com/SELinuxProject/selinux. Accessed 09/2021.
56. Sharif H, Abubakar M, Gehani A, Zaffar F (2018) Trimmer: application specialization for code debloating. In: Proceedings of the International Conference on Automated Software Engineering, ACM/IEEE
57. Song S, Suneja S, Le MV, Tak B (2023) On the value of sequence-based system call filtering for container security. In: Proceedings of International Conference on Cloud Computing, IEEE, p 296–307
58. Song L, Xing X (2018) Fine-grained library customization. In: arXiv preprint arXiv:1810.11128
59. Strace Project. Strace - linux syscall tracer. https://strace.io. Accessed 11/2020.
60. Sysdig (2016) Falco: Cloud Native Runtime Security. https://github.com/falcosecurity/falco. Accessed 08/2021.
61. TechTarget (2019) Container vulnerability opens door for supply chain attacks. https://www.techtarget.com/searchsecurity/news/252514659/Container-vulnerability-opens-door-for-supply-chain-attacks. Accessed 04/2021.
62. The kernel development community. Perf - linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page. Accessed 03/2021.

63. The kernel development community (2017) Seccomp BPF. https://www.kernel.org/doc/html/v4.16/userspace-api/seccomp_filter.html. Accessed 03/2021.
64. The kernel development community. Namespaces. https://www.kernel.org/doc/html/latest/admin-guide/namespaces/index.html. Accessed 10/2020.
65. The kernel development community. Cgroups. https://www.kernel.org/doc/Documentation/cgroup-v2.txt. Accessed 10/2020.
66. The kernel development community. Capabilities(7) - the linux man-pages project. https://man7.org/linux/man-pages/man7/capabilities.7.html. Accessed 10/2020.
67. Tunde-Onadele O, He J, Dai T, Gu X (2019) A study on container vulnerability exploit detection. In: Proceedings of International Conference on Cloud Engineering, IEEE
68. Wan Z, Lo D, Xia X, Cai L (2019) Practical and effective sandboxing for linux containers. Empir Softw Eng 24(6):4034–4070
69. Wang X, Shen Q, Luo W, Wu P (2020) Rsds: Getting system call whitelist for container through dynamic and static analysis. In: Proceedings of the International Conference on Cloud Computing, IEEE
70. Wan Z, Lo D, Xia X, Cai L, Li S (2017) Mining sandboxes for linux containers. In: Proceedings of the International Conference on Software Testing, Verification and Validation, IEEE
71. Xing Y, Cao J, Sun K, Yan F, Wan S (2022) The devil is in the detail: generating system call whitelist for Linux seccomp. Futur Gener Comput Syst 135:105–113
72. Xing Y, Wang X, Torabi S, Zhang Z, Lei L, Sun K (2023) A hybrid system call profiling approach for container protection. IEEE Transacions on Dependable and Secure Computing p1–p15, PrePrint
73. Zhang Z, Cheng Y, Nepal S, Liu D, Shen Q, Rabhi F (2018) KASR: a reliable and practical approach to attack surface reduction of commodity OS kernels. In: Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses

Publisher's Note