Securing Container-based Clouds with Syscall-aware Scheduling

Michael V. Le
mvle@us.ibm.com
IBM Research
Yorktown Heights, NY, USA

Salman Ahmed
sahmed@ibm.com
IBM Research
Yorktown Heights, NY, USA

Dan Williams
djwillia@vt.edu
Virginia Tech, IBM Research
Blacksburg, VA, USA

Hani Jamjoom
jamjoom@us.ibm.com
IBM Research
Yorktown Heights, NY, USA

ABSTRACT

Container-based clouds—in which containers are the basic unit of isolation—face security concerns because, unlike Virtual Machines, containers directly interface with the underlying highly privileged kernel through the wide and vulnerable system call interface. Regardless of whether a container itself requires dangerous system calls, a compromised or malicious container sharing the host (a *bad neighbor*) can compromise the host kernel using a vulnerable syscall, thereby compromising all other containers sharing the host.

In this paper, rather than attempting to eliminate host compromise, we limit the effectiveness of attacks by bad neighbors to a subset of the cluster. To do this, we propose a new metric dubbed *Extraneous System call Exposure (ExS)*. Scheduling containers to minimize ExS reduces the number of nodes that expose a vulnerable system call and as a result the number of affected containers in the cluster. Experimenting with 42 popular containers on *SySched*, our greedy scheduler implementation in Kubernetes, we demonstrate that *SySched* can reduce up to 46% more victim nodes and up to 48% more victim containers compared to the Kubernetes default scheduling while also reducing overall host attack surface by 20%.

CCS CONCEPTS

• **Security and privacy** → **Systems security**; **Software and application security**.

KEYWORDS

container, scheduling, system call, seccomp, co-location

ACM Reference Format:
Michael V. Le, Salman Ahmed, Dan Williams, and Hani Jamjoom. 2023. Securing Container-based Clouds with Syscall-aware Scheduling. In *ACM ASIA Conference on Computer and Communications Security (ASIA CCS '23), July 10–14, 2023, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3579856.3582835

ASIA CCS '23, July 10–14, 2023, Melbourne, VIC, Australia
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0098-9/23/07.
https://doi.org/10.1145/3579856.3582835

1 INTRODUCTION

In the ever expanding container cloud ecosystem, the inherent interaction model between the containers and the underlying system remains a security concern. Since containers are composed of processes, they interact directly with the highly privileged host kernel through the system call interface. The system call interface is wide: to date, Linux includes over 340 system calls. Any system call that contains a vulnerability could potentially be used to escalate privileges or otherwise compromise the system. Some recent examples of such exploit include the infamous Dirty COW vulnerability [12] (CVE-2016-5195) or the recent Dirty Pipe vulnerability [37] (CVE-2022-0847), Integer underflow vulnerability (CVE-2022-0185), and the Waitid vulnerability (CVE-2017-5123). Thus, the security of a container is intimately linked with the security of neighboring containers on a host and what system calls those containers can access and potentially exploit. Yet cloud providers strive to increase container density to improve host utilization by multiplexing workloads from different users and tenants on the same set of physical hosts, providing ample opportunities for distrusting containers to become *"bad neighbors."*

One approach to improving the security of the host is to limit the system call interface, and thus its attack surface. Mechanisms that perform per-process system call filtering using seccomp [19] or software specialization like Chisel [24] and FaceChange [21] can restrict access to the kernel for a specific process. Similarly, host or system-wide specialization approaches like kRazor [34] restrict access to system calls used by containers on a host. However, they are inadequate on their own as they do nothing to *prevent* bad neighbor relationships from occurring in the first place.

In this work, we focus on exploiting the relationship between container scheduling and system call profiles to *practically* improve the overall security of container-based clouds. Specifically, we propose a greedy *system call-aware* container scheduler called ***SySched*** that places containers based on their system call usage profile in such a way that can minimize each container's exposure to extraneous system calls on a host—system calls a container does not use itself but can potentially be exploited to its own detriment. The key to achieving this is the introduction of a new *Extraneous System call Exposure* (*ExS*) metric that quantifies the amount of extraneous system calls a container is exposed to. Rather than attempting to eliminate system call-based compromises, a great challenge in itself, our approach makes use of this metric to inform container placement decisions to contain and limit the impact of such exploits.

To determine the feasibility of our approach, we analyzed system call usage captured by *sysdig* from a representative set of containerized workloads that consists of 42 popular and highly-downloaded container applications across 11 categories from Docker Hub. Our analysis uncovers two key insights: *(i)* most applications only use a fraction of system calls (9% - 37%) available on the host and *(ii)* as little as 23% or as much as 99% of system calls are common between different applications. These insights suggest that applications that have similar system call profiles will have very few extraneous system calls among them. If they can be co-located, then there is opportunity for limiting the effectiveness of bad neighbors that may exploit vulnerabilities in these extraneous system calls to a subset of the cluster.

System call-based container co-location intuitively raises two concerns: i) impact on performance and ii) facilitation of targeted co-location attacks [15]. With respect to performance, we do not envision our scheme being used solely for dictating container placement as there are often many considerations such as performance and utilization that must be taken into account when scheduling workloads in the cloud. Our goal is to introduce a new security dimension based on system calls to the existing container scheduling schemes. How prominent SySched plays in the overall placement decision is based on the priorities and scheduling goals of the workload owner and cloud provider. As such, like other scheduling schemes, SySched can be configured to have a range of impact on other scheduling dimensions, from being the sole arbiter of container placement decisions to taking over only *after* all other placement constraints have been met. The trade-offs associated with different policies must be considered by the cloud providers.

Targeted co-location attacks can be used to select victims in order to launch additional attacks. Our scheme can potentially make such attacks easier but they can be mitigated in certain situations as we describe later in the paper (see Section 5). However, it should be noted that existing widely deployed container schedulers are already highly susceptible to such targeted co-location attacks [15] and will require additional mechanisms to completely mitigate which we leave for future work.

To evaluate our approach, we implemented SySched in Kubernetes, an open-source and widely deployed container orchestration engine. We make the case that there is ample opportunity for reducing ExS, thus improving overall security in a cluster, by showing that there can be up to 4x difference in ExS between the default Kubernetes scheduler and a near-optimal, albeit unrealistic, offline oracle scheduler. Our experimental evaluation demonstrates that SySched can greatly improve the situation by reducing ExS by up to 2x compared to the default Kubernetes scheduler. We show that *SySched* can reduce up to 46% more victim nodes and up to 48% more victim containers compared to the Kubernetes default scheduling. In addition, we are able to reduce the host attack surface of a cluster by approximately 20% (20% less system calls are needed). Furthermore, we empirically show that while scheduling for security can impact performance, it is not always the case and heavily depends on the workload.

Container schedulers in container orchestration engines such as Kubernetes already make placement decisions based on resource availability, high availability, and user-specified co-location constraints, but do not take into account the security implications of

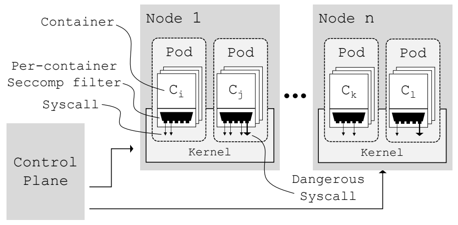


Figure 1: Anatomy of a typical container deployment

the resulting placement decisions. Of course, scheduling has been used to separate tenants (*e.g.*, Coke and Pepsi) on physically separate hosts and scheduling applications on different cores to thwart certain kinds of side-channel attacks. Academic research in the area of security-aware schedulers has so far focused on VM scheduling and use CVSS as the single risk metric to globally reduce [22, 35, 54]. Our work is the first to discuss and explore the opportunistic interplay between system call exposures and container scheduling.

In summary, this paper makes the following contributions:

- We identify limitations of existing system call filtering schemes due to the bad neighbor problem and propose the use of container scheduling to address them.
- We introduce a new container scheduling metric based on extraneous system call exposures (ExS) and utilize it to design our container scheduler scheme SySched.
- We implement a prototype of SySched as a scheduler plugin in Kubernetes and evaluate its impact on security and performance.

2 THE EFFECT OF SYSTEM CALLS ON SECURITY OF CONTAINER-BASED CLOUDS

In Linux, containers are processes or groups of processes that experience a private view of system resources via kernel mechanisms including namespaces and cgroups, despite potentially sharing the kernel with other containers. In a container-based cloud, the compute abstraction provided to customers are containers. Figure 1 depicts a typical, and our assumed, container-based cloud environment, such as one managed by Kubernetes. The cloud manages a number of worker *nodes*, which could be physical or virtual machines. The control plane schedules user-supplied containers to run on worker nodes. The unit of scheduling is typically a *pod*, which consists of one or more related containers. In this work, we limit our discussion to pods with one container and thus use the terms *pods* and *containers* interchangeably.[1] Importantly, each node runs one instance of a kernel, so every container on a node—even from mutually distrusting tenants—shares the kernel.

2.1 Sandboxing Containers

Mutually distrusting containers—either from one or multiple organizations—can share a host, and therefore a common system call interface into the kernel. Concerns over neighboring containers exploiting a vulnerable system call to perform privilege escalation or

[1]Further discussion on the implications of multi-container pods appears in Section 5.

container breakout have led the industry to consider container sandboxing techniques. For example, gVisor[48] and VM-based techniques such as Firecracker microVMs[1] and Kata containers [27] are being discussed as a possible replacement for containers due to the promise of better isolation guarantees through specialized application-level kernels or hardware-based mechanisms while maintaining an equivalent level of performance.

Despite isolation improvements, these virtualization and user-level kernel approaches have drawbacks such as application incompatibilities or requirements for separately maintained guest kernels running on a limited set of supported CPUs and are ill-suited to already-virtualized environments. We believe that to unlock the full potential of container-based clouds, clouds where containers run directly atop the host OS, will require overcoming security concerns, and in this work, we offer ways to practically improve on the state-of-the-art, which centers around system call filtering.

In particular, recent work has suggested that the kernel can be protected from containers by reducing access to vulnerabilities through system calls [51, 52]. As depicted in Figure 1, on a per-container basis, mechanisms like `seccomp` can limit the system call interface. Default per-container system call filters are typically too permissive for fear of causing the containers to fail. Recent work has investigated automatically deriving tighter, more application-specific filters [8, 11, 18, 19]. Unfortunately, as we will describe next, application-specific approaches are fundamentally unable to produce a safe container cloud due to the existence of bad neighbor(s).

2.2 Threat Model and Goals

We assume a large single or multi-tenant container-based cloud environment as described above, in which isolation of containers is desirable. We assume the cloud operator(s) and administrator(s) are trustworthy and that application-specific system call filtering is specified in advance for containers and enforced by the host kernel.

Further, we assume that some containers can be malicious by deliberately exercising vulnerable system calls to exploit the host kernel or can become malicious after being compromised by an attacker through vulnerabilities that may exist in those containers. These containers then may attempt to break out of containment by using one or more vulnerable system calls, i.e., these containers become bad neighbors. Note that we assume a malicious container can specify any system call profile it chooses ahead of time; any escape by a container must only use system calls in its profile. We place other attack vectors such as vulnerabilities in a container's own code, including micro-architectural side channels, out-of-scope.

We assume a container that has compromised the kernel trivially compromises all other containers on its host. However, we assume that a compromise of the kernel on one node does not imply compromise of kernels or containers on other nodes. Compromise of another node must come from a malicious container on that node through a vulnerable system call. Likewise, vulnerabilities in container-based cloud platforms, including code/image repositories, container runtime, and orchestration engines are out of scope.

Given this threat model, we identify three reasons that an application-specific system call filtering approach alone cannot lead to a safe container-based cloud:

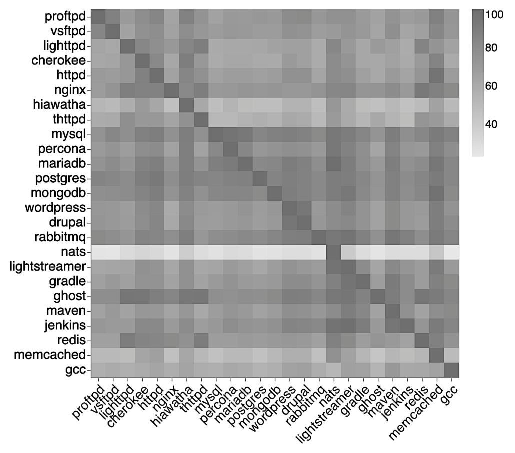


Figure 2: System call similarity (%) between two containers w.r.t. the container on the y-axis. Darker means more similar.

- **Diverse system call usage:** There is no one-size-fits-all system call profile that is also tight or a comprehensive list of dangerous system calls that all applications are able to adhere to without compromising their correct execution. Thus some tenants may run containers that use system calls that (from another tenant's perspective) jeopardize the host.
- **Sharing of the privileged kernel:** A compromised kernel implies compromise of all containers on the host. Containers that may have different system call profiles and may not agree on what system calls should be avoided must implicitly trust each other not to compromise the kernel through the system calls available to them.
- **Lack of control:** Even if a user's containers adhere to a per-container filter to avoid a particularly risky system call (e.g., one with a known CVE or history of CVEs), there is no guarantee that other containers have adhered to the policy, especially in a multi-tenant, container environment where placement is completely abstracted away from users.

Since malicious containers have the ability to use vulnerable system calls (as long as they are declared ahead of time) in our threat model, *our goal is not to prevent attacks*, i.e., bad neighbors may still compromise the kernel. Instead we take a holistic cluster-wide approach and seek to ***contain*** the damage of attackers. To this end, we articulate the following goals:

- (G1) Minimize the number of *hosts* in the cluster affected by a malicious container escape via one or more vulnerable system calls.
- (G2) Minimize the number of *containers* in the cluster affected by a malicious container escape via one or more vulnerable system calls.

2.3 Better Security through Scheduling

To better understand the limitations of an application-specific approach to system call filtering and motivate a cluster-wide scheduling, we examine the commonalities and differences in system calls across different applications.

We analyze the system call invocations of 42 popular and highly-downloaded containers across seven categories from Docker Hub. To gather the system call data, we ran a variety of workloads against the containers and captured system calls using `sysdig`. Across all 42 containers, we found they used between 9% and 37% of the full Linux system call interface of 335 calls. Our findings are in line with related work that utilizes source-level static analysis to determine system call usage (between 25% and 29% for the serving phase of popular applications) [19] and a large scale binary analysis (30,398 binaries) showing a median of 26% [11].

We further analyze the system call similarity across the 42 containers, shown in Figure 2 (only shown for 25 applications for readability. Each cell shows the commonality of system calls between two containers, represented by a percentage of common system calls from the perspective of the container on the y axis (i.e., w.r.t. the containers on the rows). In other words, the percentage is computed as the size of the intersection of the system call sets of the two containers out of the size of the system call set for the container on the y axis. For all containers, between any two, we observe as little as 23% similarity to as much as 99% similarity with another container. In addition, we also observe diversity in system calls associated with kernel CVEs and the use of those system calls in the 42 containers. For example, we analyzed a collection of 50 kernel-based CVEs (2008-2018) and found they are associated with 59 unique system calls and some of these system calls are used by almost all of the containers while others are only used by a few.

The diversity of system calls confirms that tight, application-specific system call filters will vary from application to application and that access to/avoidance of vulnerable system calls will also vary between them. Furthermore, when considering the impacts of malicious containers in our threat model, placement matters. Strictly consolidating containers that require a vulnerable system call on fewer nodes will reduce the number of nodes at risk for compromise (G1). Similarly, given our threat model, strictly consolidating containers that *do not* require a vulnerable system call on other nodes protects them from the risks of a shared kernel (G2). Additionally, system call usage-based container placements lead to an overall attack surface reduction on hosts and improved isolation of the container cloud (see Section 4).

3 SYSCALL-AWARE SCHEDULER DESIGN

Given our threat model, malicious containers may compromise the kernel and other containers on the host through one or more vulnerable system calls. As cloud providers implement container scheduling, we do not know which system calls may be vulnerable, nor which containers may be malicious. But the scheduler can still reduce the aggregate system call attack surface on each host.

Our key insight is that broad system call attack surfaces are a tragedy of the commons: container developers face no consequences for utilizing a broad set of potentially vulnerable system calls or specifying them in a syscall filtering policy, nor do they gain rewards for minimizing their system call usage. Simply scheduling to minimize system call attack surface on the host does not address this concern.

Instead, we introduce a concrete incentive to container developers, namely that they implicitly agree that system calls they themselves use are low risk: they are deemed unlikely to contain vulnerabilities now *or in the future*. It follows that such a container will have no safety concerns being co-located with other containers that use those (or a subset of those) system calls. Containers that use fewer system calls or those less likely to encounter vulnerabilities will be co-located with others that have made similar assessments of system call safety and adhered to a corresponding system call filtering policy.

To capture this incentive in scheduling, we introduce a new single-valued metric used to quantify the **Ex**traneous **S**ystem call exposure (*ExS*) of a container on a node. Informally, *ExS* captures the additional system calls used by other containers on a node that the container itself does not use and thus has not implicitly agreed to be low risk. In this way, *ExS* reflects the potential "danger" a container must face when scheduled on a node.

3.1 Scheduling Metrics

In this work, we assume that the system call profiles of pods have explicitly been made available to the scheduler, possibly automatically pre-generated through analysis (*e.g.*, dynamic such as via the Security Profiles Operator in Kubernetes) and/or static analysis [8, 11, 19], and explicitly specified in a configuration file of a pod (*e.g.*, a `seccomp` profile).

We assume that all nodes are running identical host operating systems with the same system calls on each node, numbered $1, ..., M$. Let $\boldsymbol{S}_i^n$ represent the binary vector of enabled system calls for container i on node n: $\boldsymbol{S}_i^n = [s_1, s_2, ..., s_M]$, where $s_k = 1$ when the k-th system call is enabled, and $s_k = 0$ otherwise. Practically, $\boldsymbol{S}_i^n$ mirrors a typical `seccomp` policy for the corresponding container.

To find the systems calls that are enabled a given node n, we perform a logical *or* (union) of the enabled system calls across all containers within that node:

$$\boldsymbol{S}^n = \bigcup_i \boldsymbol{S}_i^n$$

We can now compute the vector, $\boldsymbol{E}_i^n = [e_1^n, e_2^n, ..., e_M^n]$, which represents the extraneous systems calls for container i on node n:

$$\boldsymbol{E}_i^n = \boldsymbol{S}_i^n \oplus \boldsymbol{S}^n$$

Some system calls may be "riskier" than others such as system calls that are known to be critical in developing exploits or are themselves associated with CVEs that have since been fixed, hence historically vulnerable [18, 19, 26]. The presence of these system calls does not mean imminent danger but may justify stronger isolation measures. Weights can be used to inflate the ExS scores when these risky system calls are encountered, thereby, providing a mechanism to control the degree of isolation. Increasing weights on some system calls can result in more aggressively co-locating workloads using those system calls together while pushing others away to curtail the impact in the event those system calls get exploited. For system calls that are known to be imminently vulnerable, direct quarantining measures of the associated workload such as using existing affinity/anti-affinity mechanisms may be the better choice.

Let $\boldsymbol{W} = [w_1, w_2, ..., w_M]$ be the set of *riskiness* weights associated with each system call (more below). As mentioned earlier, we are interested in computing a score, *ExS*, that reflects the extraneous system call exposure. For container i on node n, this score can then be computed as the dot product between $\boldsymbol{E}_i^n$ and $\boldsymbol{W}$:

$$ExS_i^n = \boldsymbol{E}_i^n \cdot \boldsymbol{W} = e_1^n \times w_1 + e_2^n \times w_2 + ... + e_M^n \times w_M$$

When scheduling, we use the ExS_i^n score as one of the basis on which to decide which candidate hosts to place an incoming container. By using the ExS_i^n score, it allows us to answer a key question during scheduling: how does the placement of a new container on a host impact that container's ExS_i^n score as well as the scores of all current running containers on that node? This node-wide ExS^n score can be calculated by first assuming the new incoming container is placed on the target node and then calculating the ExS^n score for each container on that node and adding those scores together:

$$ExS^n = \sum_i ExS_i^n$$

It is useful to also calculate the ExS score for an entire cluster as a way to evaluate how well syscall-aware scheduling can reduce the extraneous system call exposure of all the nodes in that cluster. The cluster-wide ExS score for a cluster with N nodes is the summation of the ExS^n score across all nodes:

$$ExS = \sum_n ExS^n$$

Weights: Each weight $w_1...w_M$ in W is a composite function:

$$w_k = f(\overline{w_k}) = \begin{cases} 0, & \text{if } \overline{w_k} = -1 \\ 1, & \text{if } \overline{w_k} = 0 \\ \overline{w_k} * s, & \text{if } \overline{w_k} > 0 \end{cases}$$

$\overline{w_k}$ is a user-provided value representing the riskiness of the k-th system call. When $\overline{w_k}$ is -1, the k-th system call is not considered as extraneous, hence weight $w_k = 0$. $\overline{w_k} = 0$ means no special consideration for the k-th system call. If all system calls are treated equally, then $\boldsymbol{W} = [1, 1, ..., 1]$. To allow the risk values to affect placement decisions, s is provided as a system maintained average of the differences of ExS^n (node-wide score) among nodes in the cluster. This value ensures the risk values actually affect the resulting ExS^n score enough to change the ranking order of candidate nodes during scheduling. Hence, $\overline{w_k} > 0$ expresses the level of aggressiveness of the isolation of the k-th system call based on its risk factor. For example, $\overline{w_k} = 2$ would suggest that the presence of the k-th system call would result in a relatively large ExS^n score, thereby making nodes hosting containers utilizing the k-th system call to be unlikely a top candidate node. In Section 4, we demonstrate how weights can be used to better isolate containers with known-risky system calls. We also discuss approaches to determine the weights of risky system calls in Section 5.

3.2 Scheduling Scheme

We explain how the ExS metric fits into the Kubernetes scheduling scheme below. For a given incoming container, the Kubernetes scheduler first finds the set of feasible nodes by filtering the available nodes based on scheduling requirements and policies (e.g., resource constraints, affinity, etc.). After the filtering phase, the set of feasible nodes needs to be ranked. This is done by passing the incoming container along with a list of feasible nodes as inputs to different scoring mechanisms, SySched being one among potentially many. The cloud administrator can set weights to these mechanisms to underscore the impact of one mechanism over another. All these scoring mechanisms work together to compute the final scores to rank feasible nodes for the incoming container and works as follows. Each mechanism computes its own score for each feasible node. Then, the scores gets normalized and is combined based on the mechanism's weight to produce a final score w.r.t a feasible node. In the case of SySched, it computes its score using the ExS^n metric for each feasible node and returns the reverse normalized score, i.e., a lower ExS score yields a higher normalized score. The Kubernetes scheduler then sorts the feasible nodes based on the final scores, and selects the node with the highest score. If there exist multiple nodes with the same high score, the scheduler randomly selects a node from these same-scored nodes. If no feasible node is available, the scheduler places the incoming pod in a queue to wait for feasible nodes to be available.

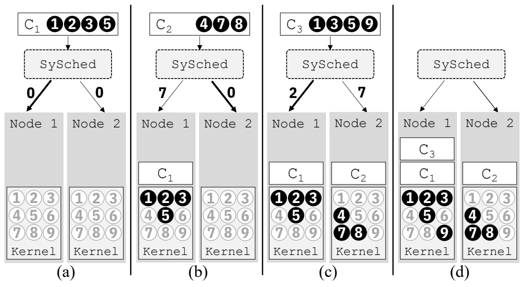


Figure 3: Step-by-step illustration of the syscall-aware scheduling using ExS. Numbers on the arrows indicate the ExS^n score. The numbers inside each gray box below a host indicate the union of system calls used by all containers on that host. The numbers inside rectangular boxes of containers (`C1`, `C2`, and `C3`) indicate the system calls the containers use.

Figure 3 illustrates how our scoring impacts placement decisions. To simplify, we consider two feasible nodes (Node 1 and Node 2) and the raw ExS score as the *sole factor* for ranking the nodes. In practice, our normalized ExS scores may be combined with other scoring mechanisms to rank feasible nodes. The figure shows the placement of three containers (C_1, C_2, and C_3) in Node 1 and Node 2. The numbers inside a pod indicate the system call profile of the pod, i.e., the profile for C_1 is $1, 2, 3, 5$, C_2 is $4, 7, 8$, and C_3 is $1, 3, 5, 9$.

Initially, to schedule container C_1, the ExS scores for both Node 1 and Node 2 are the same (i.e., 0), since there are no containers running on the nodes (Figure 3(a)). Thus, we randomly pick one of the two nodes, Node 1 in this case, and updates Node 1's system call usage list using C_1's system call profile (Figure 3(b)). When container C_2 arrives to be scheduled, our plugin computes the ExS scores 7 and 0 for C_2 w.r.t. Node 1 and Node 2, respectively (Figure 3(b)). In this case, the plugins ranked Node 2 over Node 1 since Node 2 has a lower ExS score (i.e., higher normalized score). Figure 3(c) shows the updated system call list for Node 2. This process repeats for all incoming containers. After all containers are placed, the cluster-wide ExS score in this example is 12, i.e., ExS_1 is 5 and ExS_2 is 7, where 1 and 2 are node numbers.

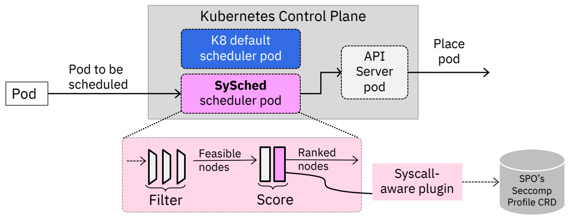


Figure 4: Kubernetes scheduling flow and the syscall-aware scheduling plugin. Both the default and SySched scheduler run in tandem as pods on the master node.

Figure 3 also highlights the opportunities not only for reducing the exposure to extraneous system calls (i.e., ExS score) but also the node's attack surface. If C_3 had been scheduled on Node 2 instead of Node 1, then Node 2 would have a total of 7 system calls "open" $(1, 3, 4, 5, 7, 8, 9)$, instead of only 3 system calls under the SySched approach. We further demonstrate this phenomenon with real kernel CVEs in Section 4.

3.3 Implementation

We implemented a prototype of our scheduler for Kubernetes [2]. In Kubernetes, the scheduler is extensible and supports a plugin architecture model where new features can be added by implementing plugins that extend scheduler specific API extension points. These plugins are then compiled into the scheduler.

The SySched Scheduler Pod. Figure 4 shows the basic framework of Kubernetes' scheduler and our additions to implement the syscall-aware scheduling. In short, we developed a new plugin to implement the scoring extension API point in the Kubernetes scheduler while leaving other filters and scoring plugins intact. We leverage existing filtering and scoring operations of the Kubernetes scheduler to handle other aspects of pod placement such as spreading the pods and ensuring resource availability.

Our scheduler exists in tandem with the default scheduler in Kubernetes. Specifically, it is run as a separate pod alongside the default scheduler's pod in the Kubernetes environment. We deploy our scheduler in the Kubernetes master node but it can run anywhere in the cluster. In our implementation, the name of our syscall-aware scheduler or the default scheduler can be specified in each pod's configuration file for ease of experimentation. Kubernetes will invoke the named scheduler to perform placement.

Our plugin consists of two main components: 1) a lightweight in-memory state store that stores the current mapping of nodes to pods, and 2) mechanisms for obtaining system call sets used by pods and calculating ExS scores. The state store operates as a thread that monitors pod lifecycle events (e.g., creation, stop, destruction, run) to track the location of running pods in the cluster and update its internal mapping. While the entire pod placement state can be retrieved dynamically from the API server, it is expensive to do so for every scheduling event. Therefore, we opt to maintain this state internally within our scheduler.

[2]We are in the process of open sourcing our code. https://github.com/kubernetes-sigs/scheduler-plugins/tree/master/kep/399-sysched-scoring

Table 1: Applications used in the evaluation

Type	Application Name
OS	alpine, centos, ubuntu, busybox
FTP Server	proftpd, vsftpd
Web Server	lighttpd, cherokee, httpd, nginx, hiawatha, thttpd, nodejs
Database	mysql, postgres, mongodb, mariadb, percona, influxdb, elasticsearch
Storage	redis, memcached
Content management	drupal, wordpress, ghost
DevOps	jenkins, maven, gcc, gradle
Messaging system	rabbitmq, nats, lightstreamer
Language/compiler	python, golang, ruby, gcc, openjdk
Container management	docker, docker-registry
Other	vault, zookeeper, nextcloud

To compute the ExS scores for pods, our plugin retrieves pods' system call sets from their seccomp profiles via the Security Profiles Operator (SPO [31]). SPO allows the creation and retrieval of seccomp profiles as Custom Resource Definitions (CRDs). We rely on the SPO to generate and bind seccomp profile CRDs to the pods.

Determining System Call Policies. In this work, we use both dynamic and static container profiling approaches to obtain system call information for generating seccomp profiles. For dynamic profiling, the applications are exercised by their respective workload generators (discussed in Subsection 4.1). We capture the system calls (using sysdig [46]) from the moment a container launches the application (capturing the application's initialization phase) to the end of the benchmark run. We combine the system calls in both phases to generate the dynamic seccomp profiles of the respective container. For static profiling, we rely on tools from Confine [18] to identify all the necessary binaries in a containerized application and extract all reachable system call invocations from those binaries.

4 EVALUATION

We present results that demonstrate the efficacy of our approach. Primarily, we seek to answer the following questions:

- how well can SySched ***reduce ExS***?
- how well does SySched ***achieve our security goals***?
- how likely is ***performance negatively impacted*** by SySched?

4.1 Experimental setup

We deploy Kubernetes version 1.23 in a cluster of VMs. All the VMs run Ubuntu 18 (Linux kernel v4.15) and are of type Libvirt/KVM hosted on a bare-metal server with 96 CPUs and 376GB of memory. Each VM has 8 VCPUs and 8GB of memory and utilizes virtio devices. The VMs' VCPUs are pinned to their respective set of physical CPUs to reduce contention and instability of the performance measurements. Vagrant is used to automate the creation or destruction of the VMs and deployment of the Kubernetes runtime. When compared to available cloud resources, this setup allows finer control over physical resource allocation to the VMs and networking topology, providing a more stable environment for measurements.

To select our experimental application set, we focused on Docker Official images which are around 170 images out of roughly 2,500 publicly available images that we could obtain download statistics. These 170 images comprise greater than 61% of all image downloads. Out of these official images, we selected 42 images across

11 categories (Table 1) comprising ~90% of all the downloads of official images. We also included in this 42 images a few images that are not in Docker Official images to expand the types of application categories. We believe our selection criteria is reasonably reflective of the types and varieties of containers deployed in the real-world.

We generate a `seccomp` profile for each of the containerized applications from their extracted system call profiles (Subsection 3.3). For dynamic system call extraction, we use the following workload generators to emulate typical activities performed by those applications. Specifically, we use ftpbench [45] for FTP servers, wrk [50] for web servers, sysbench [2] for relational databases, perftest [40] for rabbitmq, nats-bench [13, 14] for NATS, lightstreamer builtin demo [36], YCSB [7] for non-relational databases and storage, influxdb-comparisons [25] for InfluxDB and ElasticSearch, zk-smoketest [39] for ZooKeeper and UnixBench [28] for OS containers. Additionally, we created custom workload generators for the remaining applications for browsing and creating content for content/secret/ DevOps management systems and compiling/building projects from source. We discuss the details of workload generators in the Appendix (A.1). Obtaining precise and concise system call set is a known challenge and we discuss more on this issue in Section 5.

For experiments described below, we deploy 3 instances of each of the 42 applications for a total of 126 pods. We automated the pod submission process to scale experiments. In each run of an experiment, we submit all 126 pods in sequential but random order and wait until all pods are in a "run" state before recording the placement of pods and computing the respective metrics being reported. We run these experiments for different numbers of feasible nodes in a cluster, and for a particular number of feasible nodes, we repeat the experiments 10 times. Unless noted otherwise, no special weights are applied to the system calls, i.e., $\boldsymbol{W} = [1, 1, ..., 1]$.

We use numbers of feasible nodes instead of cluster size to emphasize the fact that regardless of the cluster size, the number of actually available nodes only matters to our scheme for scheduling as some fraction of nodes will likely be used for scheduling containers with other higher priority scheduling goals. Providing a range of feasible nodes give a better sense of the range of benefits vs. cost.

4.2 Estimating Optimal ExS Reduction

Before presenting the results, we discuss how to compute, in an ideal situation, the best (lowest) ExS scores that can be obtained when scheduling is the sole means used for affecting the scores. To do so, we employ an offline scheduling scheme where the scheduler has prior knowledge of all containers to be scheduled. While not practical for real-world deployments, this approach is useful to put the experimental results into perspective. Since obtaining optimal placements is not feasible (NP-hard), we approximated optimal placements by selecting the best outcome from multiple rounds of four clustering algorithms: agglomerative, birch, k-means, and mini-batch k-means. These algorithms utilize an ExS-based distance formula (Subsection 3.1) to minimize the cluster-wide ExS score.

4.3 Reducing ExS

To evaluate how well SySched reduces ExS, in addition to the oracle ExS, we compare SySchd to the *default* scheduler in Kubernetes. For the default scheduler, we rely on the scheduler configuration and policies specified at the time of installing and setting up Kubernetes.

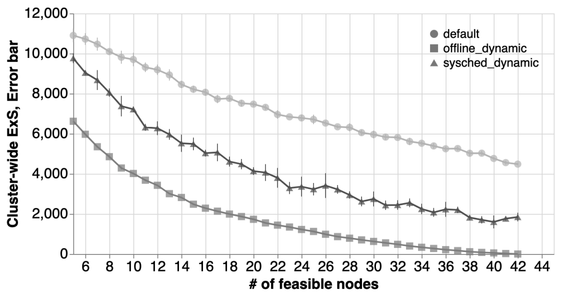
Figure 5 shows the overall cluster-wide ExS score for different size clusters (5 to 42 nodes). Resource constraints prevent us from successfully deploying 126 container instances in a cluster with less than 5 nodes. This cluster-wide score, as explained in Subsection 3.1, gives a high-level summation of the amount of exposures each container experiences on each node across the entire cluster. As can be seen from Figure 5(a), the *offline dynamic* case (computed with dynamic system call profiles) goes from around 6000 ExS score in a five node cluster to zero in a 42 node cluster. This is because in the best case, all three instances of an application get placed on the same node. Since we have 42 nodes for 126 instances, each node gets exactly three instances of an application, which results in a zero ExS score. Of course, for an online scheduling scheme with no system-call aware metric like the default Kubernetes scheduler, the cluster-wide ExS score is much worse. As shown in Figure 5(a), there can be around a factor of four difference in the amount of extraneous system calls exposed to containers between the *offline* and the *default* Kubernetes scheduler. However, by adding our syscall-aware scheduling scheme (i.e., *SySched*) into the Kubernetes scheduler, the ExS score is reduced by more than half for the dynamic case, coming closer to *offline*. The reduction is more pronounced as the number of nodes increases because the number of applications in our experiments is fixed. This means that the scheduler has more choices to optimize the placements of the containers, yielding better results. As such, there is an important relationship between number of nodes and container instances. The improvement in ExS score over the default scheduler is the greatest between 20–25 nodes and maintains that improvement as the cluster size increases. This ratio of containers to nodes and the mix of containers and their system call usage profiles can be important in practice to guide cluster sizing and container admission controls to maximize benefits from using security-aware scheduling schemes such as ours.

To determine whether the results are sensitive to the dynamic benchmark-driven system call profiles we obtained, we also performed the same experiments using `seccomp` profiles obtained through static analysis described in the Confine work [18]. Most of the system call profiles generated by Confine have roughly 2.5 times more system calls and the number of system calls across different applications are closer together compared to the ones generated by our dynamic approach (see Table 3 in Appendix). This is because Confine extracts all system calls reachable from binaries executed in the container, regardless if they are actually used during runtime.

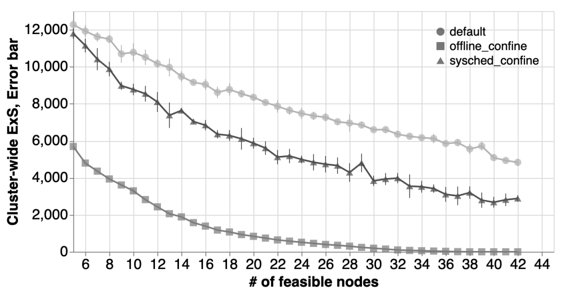
Nevertheless, despite the significantly larger system call profiles, we observed significant ExS reduction trends (up to 51%) by SySched compared to the default with Confine generated system call profiles (Figure 5(b)). Also, our scheme reduced the gap between offline and default by around one-third. However, the overall improvement is less than in the dynamic case due to the smaller variation of system call numbers leading to less room for improvement.

4.4 Security Benefits

Intuitively, our scheduling scheme minimizes the ExS score for all containers throughout a cluster by co-locating containers that have



(a) Cluster-wide ExS score using dynamic system call profiles



(b) Cluster-wide ExS score using static system call profiles

Figure 5: Average cluster-wide ExS score across runs (the lower, the better) using system call profiles extracted through (a) dynamic analysis (Sysdig [46]) and (b) static analysis (Confine [18]).

similar system call usage. This leads to the clustering of containers that utilize a specific set of system calls while pushing away containers with differing system call sets. Such clustering has security benefits as we discuss below.

Minimizing victim nodes and victim containers. We analyze how the reduction in ExS translates to achieving our two main goals stated in Subsection 2.2: (G1) minimizing the number of *hosts* and (G2) the number of *containers* in a cluster affected by one or more malicious container escapes via vulnerable system call(s).

To compute the number of victim nodes and victim containers minimized by our scheme, we first consider one or more containers as malicious. Then we determine the victim nodes by identifying all the nodes where an instance of the malicious containers are present. We also determine the number of neighbors of the malicious containers, i.e., the victim containers. This is akin to discovering a CVE associated with a system call used by a deployed container in real-life and seeing the impact of the CVE on the cluster with respect to the container scheduling schemes.

To examine the impact, we utilized 50 historically exploitable kernel CVEs (Table 4 in the appendix) and the system calls that can be used to reach the vulnerable kernel code reported in [18, 19]. We selected these CVEs as they have been shown to be most likely associated with system calls used by our application set. Specifically, out of the 50, 40 CVEs are associated with system calls that are present in at least one of our 42 applications. There are 23 CVEs that can be reached by at least 30 applications through their system calls. One CVE impacts 14 applications. Each of the remaining CVEs impact a handful of applications ranging from 2–7.

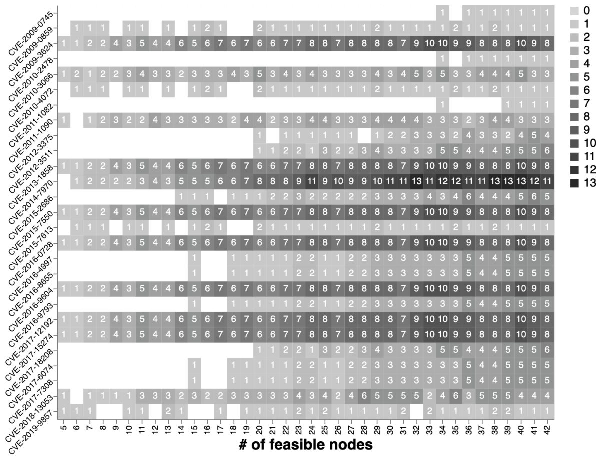
Figure 6 shows the improvement of our scheme (with dynamic system all profiles) over the default scheduler in reducing the (a) number of nodes and (b) number of containers exposed to the aforementioned CVEs. The left-hand side heat map of Figure 6 shows for each of the 40 CVEs, the difference in the number of nodes that have the system calls associated with the CVE in question (i.e., victim nodes) between the default scheduler and SySched. The right-hand side heat map shows the reduction of victim containers. Any positive difference (light gray to black cell) shows the effectiveness of our scheme for reducing the number of victim nodes or victim containers for that CVE. Cell values with zeros (i.e., white color) indicate no improvment by SySched.

Overall, as can be seen in Figure 6(a), our scheme reduces more victim nodes than the default for most CVEs. Compared to the default, our scheme can reduce the victim nodes up to 46% more (w.r.t. the total feasible nodes). However, the majority of the reduction is between one and five nodes more than the default. The improvement is more pronounced for some CVEs, especially where a set of applications with similar system call profiles have one of these CVEs. For example, `lighttpd`, `nginx`, `redis`, `ghost`, and `thttpd` share similar system call profiles and have a common CVE, e.g., CVE-2014-7970. We also observed no difference for 13 CVEs such as CVE-2008-3527, CVE-2009-0745, CVE-2010-4243, CVE-2010-4346, and so on. We omitted these 13 CVEs from the figure due to space. The reason for the low/no improvement for some CVEs is their prevalence in almost all applications under evaluation. As a result, without additional nodes in the cluster, there is no room to isolate the problematic containers having those CVEs.

We observe a similar pattern in reducing the number of victim containers. Figure 6(b) shows that SySched can reduce up to 48% more victim containers than the default. The reduction here is also more pronounced in a cluster when similar containers are malicious. For example, `mariadb`, `mysql`, and `percona` have a common CVE (i.e., CVE-2010-3066) where we have observed the reduction of up to 45 more victim containers. Again, some CVEs have low or no improvement due to their prevalence in many containers as discussed above.

Our scheme also shows improvement compared to the default scheme using statically generated system call profiles. Specifically, our scheme is able to reduce up to 32% more victim nodes and up to 25% more victim containers than default. Due to space limitation, the graphs can be found in the Appendix (Figure 8). The reduction is less than the reduction obtained using dynamic system call profiles because the statically generated profiles contain a larger number of system calls and smaller variation in system call numbers across different applications resulting in less chance for optimal placement.

Impact of weights. Thus far, we have performed the scheduling experiments without considering any emphasis or weights on different system calls. To provide examples of how weights can impact container placement, we selected two historically vulnerable system calls, `madvise` and `pivot_root`, associated with CVE-2017-18208 and CVE-2014-7970, respectively. For each system call, we apply


(a) Reduction of victim nodes

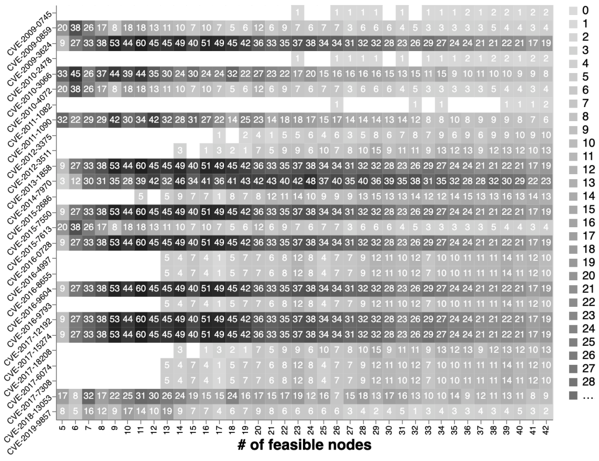
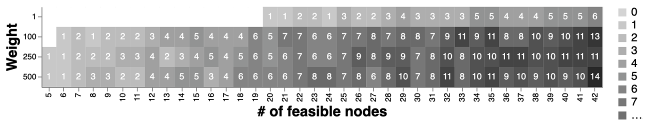

(b) Reduction of victim containers

Figure 6: Kubernetes default scheduler vs. SySched (with dynamic system call profiles) in terms of reducing the number of (a) victim nodes and (b) victim containers for a particular CVE. Each cell represents how many *additional* victim nodes or containers SySched can reduce compared to the default. Positive cell values (gray to black colors) indicate improvement. The white color indicates no improvement.


(a)

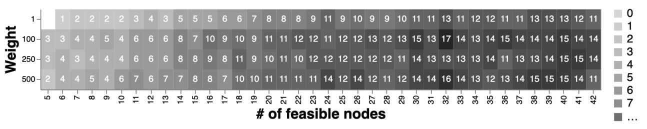

(b)

Figure 7: Impact of weights on vulnerable node reduction for (a) CVE-2017-18208 and (b) CVE-2014-7970.

different weights and observe their impact on reducing the number of vulnerable nodes as the number of feasible node is varied. These two CVEs are chosen due to their presence in 30 and 14 unique applications, respectively, to compare scenarios where relatively large and small numbers of containers are available for exploitation.

We used three different weights for our experiments: $\overline{w_k} = 1$, 2.5, and 5 with s value being 100 (empirically determined based on observing the average change in the ExS scores among the different cluster sizes). Note that the point of this analysis is to determine whether, and by how much, the weight has any impact on reducing vulnerable nodes. Hence, we only show results for cases using dynamic system call profile extraction. A discussion on approaches for weight determination based on risk assessment is in Section 5.

Figure 7 shows our results. We include the case with a weight of 0, i.e., $\overline{w_k} = 0$, for comparison. As can be seen in Figure 7(a) and (b), increasing the weights on the problematic system calls cause the associated pods to concentrate on fewer nodes (as no pods without the system calls will want to be associated with them), thus reducing the overall number of vulnerable nodes for that specific CVE in the cluster. However, the larger weight has less impact on vulnerable node reduction as the smaller weight had potentially reached the optimal placements.

Certainly, putting weights on some system calls will put pressure on the scheduler to co-locate containers that normally would be spread out leading them to possibly be more susceptible to additional vulnerable system calls. Hence, we anticipate our weight mechanism to be used to better isolate one set of "risky" system call at a time.

Host attack surface area. Reducing exposure to extraneous system calls by co-locating containers with similar system call profiles intuitively leads to the reduction of unblocked/open system calls on a node. Our results show that *SySched*, by virtue of scheduling, can reduce host kernel attack surface by up to ~20% when the cluster size is between 38 and 40 for dynamically generated system call profiles and up to ~10% with statically generated profiles. Detailed result graphs can be found in the Appendix (Figure 9).

4.5 Performance Impact

In practice, our scheduler does not work in isolation but in tandem with other scheduling plugins that handle workload and resource balancing. As mentioned earlier, we assume workloads that have strict QoS requirements are placed separately, using higher priority scheduling policies and the remaining workloads that do not have such requirements are left to be scheduled with our scheme using the remaining feasible nodes in the cluster. Nevertheless, it is still of

Table 2: Application throughput with standard deviation across different container placement configurations. Reduction is from baseline. The last column indicates whether single placement configuration performs worst than mixed.

App	baseline	mixed	single	mixed reduction	single reduction	impacted negatively?
mysql1	14569±212	2834±121	7465±37	81%	49%	no
nginx1	22056±547	9306±359	14101±140	58%	36%	no
proftpd1	72±0	66±1	64±1	8%	12%	yes (minor)
rabbitmq1	122649±2015	124463±1327	66304±1336	-1%	46%	yes
mongodb1	6399±74	2111.16±254	2589±75	67%	60%	no
memcached1	7045±412	4574.91±857	7081±168	35%	0%	no
influxdb1	43.74±1	27.24±3.5	12.63±0.23	39%	70%	yes
nodejs1	14892±119	10092.13±1188	15010±175	32%	-1%	no

interest as part of the security vs performance trade-off, to evaluate whether our scheme can lead to degraded container performance by causing excessive resource contention in both the kernel and hardware levels.

In a cluster with many different types of workloads and applications that are possibly distributed, there can be many factors that impact the performances of containers. For this study, we only focus on assessing whether the placement decisions of our scheduling scheme can negatively impact performance.

To do this assessment, we select two sets of applications with each set having four applications that span different application classes. The first set contains: `mysql`, `nginx`, `proftpd`, and `rabbitmq`. The second set contains: `influxdb`, `memcached`, `node.js`, and mongodb. Next, we find the baseline performance, in terms of throughput, of these applications to understand their performance characteristics free from any resource contention by doing the following. Each application is deployed in a VM by itself running inside a pod, and on a separate isolated VM, we run the respective workload generator to stress the application (see Subsection 4.1).

The key placement outcome that our scheme may produce is the placement of many container instances of a *single* application onto the same node to yield the best ExS score. Existing container schedulers may not have this type of outcome. Thus, we compare our placement configuration to a more typical placement configuration where a node contains a mixture of different application instances. We refer to the former configuration as *single* and the latter as *mixed*.

Table 2 shows the performance results of our experiments for the different placement configurations. As can be seen, none of the non-baseline configurations perform significantly better than the baseline, as expected. Ideally, we would like the performance of an application in the *single* configuration to perform no worse than in the *mixed* configuration, which would indicate that our scheduling scheme does not hurt performance. However, as shown, in the first set, there is degradation in throughput for `rabbitmq1` (degradation between *mixed* and *single* for `proftpd1` is not significant). Interestingly, not all applications perform worst in the *single* configuration as one may initially expect. Both `mysql1` and `nginx1` fare much better in the *single* configuration compared to *mixed*. The main reason for this is because in the *mixed* configuration they also share the node with `rabbitmq1`, which is very CPU intensive and therefore hurts the performance of the applications sharing the node with it. `proftpd1` bucks the trend in which both the *mixed* and *single* configurations perform similarly. This is because `proftpd1` does not consume much CPU as the other applications, and is, therefore, less impacted by the CPU-intensive `rabbitmq1`.

In the second set, `influxdb` is the CPU-intensive application that consumes the most CPU (almost 700% of the 8 CPU cores). Unsurprisingly, when multiple instances are executed in the *single* configuration, it exhibits the worst performance reduction. All the other applications are impacted when sharing the node with `influxdb` as shown in the *mixed* column. As `memcached` and `node.js` do not saturate their resources, the *single* configuration does not lead to a reduction of performance compared with the baseline. `mongodb`, on the other hand, has higher CPU utilization than `memcahced` and `node.js`, hence, is negatively impacted compared to baseline.

These results confirm that placing applications that can exhaust a common type of resource can result in negatively impacting performance, which certainly can be a problem for our scheme. The results also show that when resource saturation does not occur, then placing instances from the same application together does not necessarily result in degraded performance compared with a more mixed configuration. As we have shown, in some cases, application characteristics of neighboring containers can play a large role in impacting performance.

One mitigating factor that can be used to reduce the chance of our scheme over-saturating a node is to make use of resource requirements and limitations specification in the pod's configuration. The scheduler will then be able to make use of this information to prevent over-committing a node. Of course, such approaches will trade off the resulting security benefits.

5 DISCUSSION

Limitation: ExS Granularity. In our work, we focused on minimizing extraneous system calls. However, vulnerabilities can also vary based on the arguments of the system calls. Hence, a similar system call may in fact be used very differently. For instance, the file descriptor argument to `open` can determine whether a `write` call affects a file, network, or pipe. We leave the exploration of extending ExS to incorporate other forms of extraneous system properties for future research.

Limitation: Multi-container Pods. We assume single container pods in this work but pods can have multiple containers which can enlarge their system call profiles. Typically, containers that share a pod are of the sidecar or proxy variety and have very specific functionalities. Multi-tier applications such as a database and web-server often do not share a pod for scalability and flexibility reasons [38]. Subsequently, we believe the increase in the system call footprint of multi-container pods will likely be small. Still, this can impact the efficacy of our approach. Fortunately, our results with a much larger system call profile obtained through static analysis show promise for the effectiveness of our approach. We leave the study of scheduling multi-container pods for future work.

Incentivizing tighter policies. Implementing a tight seccomp policy is not trivial [8, 11, 19], and there has been little incentive for developers to do so. In today's systems, even the most conscientious container using few system calls (*e.g.*, with a tighter seccomp policy) can be co-located on the same host with a malicious actor that does not restrict its attack surface to the host, thereby incurring risk. On the other hand, we believe system call-aware scheduling can introduce an incentive for users. If the user is willing to rework their application to achieve a tighter seccomp policy, they will be rewarded by being co-located with other conscientious users. Malicious applications that do not restrict system calls would be scheduled elsewhere due to high degrees of dissimilarity.

Addressing co-location attacks. One question is whether attackers could leverage our scheduling policy to be easily co-located with a specific target. SySched may indeed increase the probability of co-location. However, the co-location attack is very probable in the default scheme [15] too. One mitigation step is to add a non-deterministic threshold that can help SySched to randomly choose a node from candidate nodes where the candidate nodes have the same or similar ExS score. The threshold helps to set the range of ExS scores that should be considered similar. Furthermore, even if such co-location attacks were to succeed and the intention is to compromise the target container via known exploitable vulnerable system calls, weights or anti-affinity mechanisms described earlier in Section 4.4 can be used to isolate such malicious containers.

Obtaining system call profiles. Not all applications have readily available workload generators which can hamper dynamic techniques. Dynamic approaches also suffers from coverage gaps when dealing with unexpected inputs. Static analysis approaches are restricted to certain programming languages and may overestimate system call usage. To address this challenge, a practical approach used in the industry [3, 49] can be deployed. This approach involves a learning phase where workloads are run in sandboxed environments while system call usage are recorded for a specified duration. This captures actual usage data, alleviating some concerns from both dynamic and static approaches but at the cost of increasing application deployment management complexity.

Determining weights of risky system calls. There are two issues to consider: 1) how to determine the risk of certain system calls and 2) how to map those risks into weight values used by SySched to perform placement. The former is an orthogonal problem to our work and we rely on existing approaches used in the industry and academia. For example, one simple method would be to classify all system calls associated with known kernel CVEs as risky, or a more refined approach would be to rank the "riskiness" of system calls based not only on their CVE association but also on their usage in known kernel exploits [26]. Regarding the second issue, sets of policies can be defined to partially automate the mapping of the system call risks into SySched weights. For example, one policy could be to assign a large SySched weight value to each system call determined to be in a risky set. Another policy could be to assign weights based on the normalized values of the ranked risk scores. In practice, administrators must customize these mapping policies to fit their risk tolerance and operating priorities.

6 RELATED WORK

Traditional schedulers. These schedulers [6, 10, 20, 53] focus on increasing density and utilization to reduce cost. They do not focus on securing the host system from malicious workloads nor the issue of bad neighbors.

Techniques for tighter syscall policy. To alleviate the container escape problem (i.e., restricting a container with tight system call profile), researchers have proposed a minimalistic profile of allowed system calls. Many works such as sysfilter [11], Confine [18], temporal specialization [19], Chestnut [8], and Prof-gen [30] utilize static analysis to identify the necessary system calls for the lifecycle of a container so that they can restrict the other system calls.

These works propose techniques to generate a tight syscall policy, but not how to control the placements of neighbors. Hence, application owners enjoy little to no incentive for the non-trivial effort of implementing a tight syscall policy [8, 11, 18, 19, 30]. *SySched*, on the other hand, provides the mechanisms for controlling the placement of "safe" container with one or more safe neighbors, thus offering a way to enable user incentives for tight syscall policies.

Security of virtual machine through scheduling. Research has been conducted into using workload scheduling to improve security focusing on virtual machines [9, 16, 22, 23, 29, 54, 55]. The common goal of these works is to improve security of the entire cloud through VM placement to reduce the probability of risks across several dimensions: co-location risks [4, 17, 22, 23, 55], risks in the hypervisor [22], and network reachability risks [22, 54]. Fine *et al.* [16] design a scheduling algorithm to make placements by minimizing the co-location threats and virtual machine density. To quantify software vulnerability risks in VMs and the hypervisor, many of these works [22, 54] rely on the common vulnerability scoring system (CVSS) score. Allowing users to specify security constraints such as requiring the availability of anti-virus/AppArmor in hosts, network encryption, and VM affinity/anti-affinity policies for use as inputs to the scheduler has been considered [9]. Security-focused scheduling for container-based cloud workloads is discussed by Bahrami *et al.* [5] with a focus on compliance of the underlying system. The scheduling metrics used in these related works do not take into consideration the relationship between scheduling and system call usage profile (i.e., the underlying system with respect to the attack surface area of the host interface).

Host security. Commercial tools such as Twistlock [49] and Aqua [3] enhance container security by securing the CI/CD pipeline such as blocking unsafe builds, scanning images against known CVEs, and providing runtime security. Traditional host security mechanisms such as SELinux, AppArmor, seccomp, and IPtables enforce access control policies on hosts. The security of hosts can also be enhanced by minimizing and specializing the operating environment to reduce the host's attack surface areas [32–34, 41–43, 47]. All of these mechanisms can be potentially leveraged by our scheduling approach to enhance the security of the underlying hosts in the cloud. Performing enforcement such as removing unnecessary code [24] or eliminating control flow (*e.g.*, ROP [44] gadgets) at the kernel level per container requires kernel switching, incurring overhead and complexity [21] but are complementary to SySched.

7 CONCLUSION

We explored the interplay between container scheduling and system call usage profiles to improve container security in the cloud. We implemented a greedy syscall-aware scheduler, *SySched*, utilizing a new ExS metric to make container placement decisions. Our experimental evaluations of *SySched* in a Kubernetes cluster using

42 real-world containers show up to 2x reduction of extraneous system calls compared to the default scheduler, effectively reducing up to 46% more victim nodes and up to 48% more victim containers.

REFERENCES

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation*. Santa Clara, CA, 419–434.

[2] Alexey Kopytov. 2022. sysbench - scriptable database and system performance benchmark. https://github.com/akopytov/sysbench. Accessed 2022.

[3] Aqua. 2018. Aqua Introduces Runtime Protection Against "Zero Day" Vulnerabilities for Containerized Applications. https://www.prnewswire.com/news-releases/aqua-introduces-runtime-protection-against-zero-day-vulnerabilities-for-containerized--applications-300682406.html. Accessed 2022.

[4] Yossi Azar, Seny Kamara, Ishai Menache, Mariana Raykova, and Bruce Shepard. 2014. Co-location-resistant clouds. In *Proceedings of the 6th Edition of the ACM Workshop on Cloud Computing Security*. 9–20.

[5] M. Bahrami, A. Malvankar, K. K. Budhraja, C. Kundu, M. Singhal, and A. Kundu. 2017. Compliance-Aware Provisioning of Containers on Cloud. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*. 696–700.

[6] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *11th USENIX Symposium on OSDI*. 285–300.

[7] Brian Cooper. 2022. YSCB - Yahoo! Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB. Accessed 2022.

[8] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. 2020. Automating Seccomp Filter Generation for Linux Applications. arXiv:2012.02554 [cs.CR]

[9] E. Caron, A. D. Le, A. Lefray, and C. Toinard. 2013. Definition of Security Metrics for the Cloud Computing and Security-Aware Virtual Machine Placement Algorithms. In *Int. Conf. on Cyber-Enabled Distributed Computing and Knowledge Discovery*. 125–131.

[10] Andrew Chung, Jun Woo Park, and Gregory R Ganger. 2018. Stratus: Cost-aware container scheduling in the public cloud. In *ACM Symp. on Cloud Computing*. 121–134.

[11] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. 2020. sysfilter: Automated System Call Filtering for Commodity Software. In *23rd Int. Symp. on Research in Attacks, Intrusions and Defenses*. 459–474.

[12] dirtycow. 2022. Dirty COW (CVE-2016-5195). https://dirtycow.ninja. Accessed 2022.

[13] Xuanyi Dong, Lu Liu, Katarzyna Musial, and Bogdan Gabrys. 2021. NATS-Bench: Benchmarking NAS Algorithms for Architecture Topology and Size. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* (2021).

[14] Xuanyi Dong and Yi Yang. 2020. NAS-Bench-201: Extending the Scope of Reproducible Neural Architecture Search. In *Int. Conf. on Learning Representations*.

[15] Chongzhou Fang, Han Wang, Najmeh Nazari, Behnam Omidi, Avesta Sasan, Khaled N. Khasawneh, Setareh Rafatirad, and Houman Homayoun. 2022. REPTTACK: Exploiting Cloud Schedulers to Guide Co-Location Attacks. In *Network and Distributed Systems Security (NDSS) Symposium*.

[16] Kevin Fine and Ezekiel Kruglick. 2018. Virtual machine placement. US Patent 9,965,309.

[17] Mauro Gaggero and Luca Caviglione. 2018. Model predictive control for energy-efficient, quality-aware, and secure virtual machine placement. *IEEE Transactions on Automation Science and Engineering* 16, 1 (2018), 420–432.

[18] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. 2020. Confine: Automated system call policy generation for container attack surface reduction. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020)*. 443–458.

[19] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2020. Temporal System Call Specialization for Attack Surface Reduction. In *29th USENIX Security Symposium*. 1749–1766.

[20] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. 2016. GRAPHENE: Packing and dependency-aware scheduling for data-parallel clusters. In *12th USENIX Symposium on OSDI*. 81–97.

[21] Zhongshu Gu, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2014. Face-change: Application-driven dynamic kernel view switching in a virtual machine. In *44th IEEE Int. Conf. on Dependable Systems and Networks*. 491–502.

[22] Jin Han, Wanyu Zang, Songqing Chen, and Meng Yu. 2017. Reducing security risks of clouds through virtual machine placement. In *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 275–292.

[23] Yi Han, Jeffrey Chan, Tansu Alpcan, and Christopher Leckie. 2015. Using virtual machine allocation policies to defend against co-resident attacks in cloud computing. *IEEE Trans. on Dependable and Secure Computing* 14, 1 (2015), 95–108.

[24] Kihong Heo, Woosuk Lee, Pardis Pashakhaloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the ACM Conference on Computer and Communications Security*. 380–394.

[25] InfluxData. 2022. influxdb-comparisons. https://github.com/influxdata/influxdb-comparisons. Accessed 2022.

[26] Sunwoo Jang, Somin Song, Byungchul Tak, Sahil Suneja, Michael V. Le, Chuan Yue, and Dan Williams. 2022. SecQuant: Quantifying Container System Call Exposure. In *Computer Security – ESORICS 2022*. 145–166.

[27] Kata Containers. 2022. Kata Containers. https://katacontainers.io/. Accessed 2022.

[28] Kelly Lucas. 2022. byte-unixbench. https://github.com/kdlucas/byte-unixbench. Accessed 2022.

[29] Seontae Kim and Young-ri Choi. 2020. Constraint-aware VM placement in heterogeneous computing clusters. *Cluster Computing* 23, 1 (2020), 71–85.

[30] Sungjin Kim, Byung Joon Kim, and Dong Hoon Lee. 2021. Prof-gen: Practical Study on System Call Whitelist Generation for Container Attack Surface Reduction. In *2021 IEEE 14th International Conference on Cloud Computing*. 278–287.

[31] Kubernetes SIGs. 2022. The Kubernetes Security Profiles Operator. https://github.com/kubernetes-sigs/security-profiles-operator. Accessed 2022.

[32] Hsuan Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. 2020. A Linux in unikernel clothing. In *Proc. of the 15th European Conference on Computer Systems*.

[33] Hsuan-Chi Kuo, Jianyan Chen, Sibin Mohan, and Tianyin Xu. 2020. Set the Configuration for the Heart of the OS: On the Practicality of Operating System Kernel Debloating. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 1 (2020), 1–27.

[34] Anil Kurmus, Sergej Dechand, and R. Kapitza. 2014. Quantifiable Run-Time Kernel Attack Surface Reduction. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. 212–234.

[35] Min Li, Yulong Zhang, Kun Bai, Wanyu Zang, Meng Yu, and Xubin He. 2012. Improving cloud survivability through dependency based virtual machine placement. In *International Conference on Security and Cryptography*. 321–326.

[36] Lightstreamer. 2022. Lightstreamer Doc. https://github.com/docker-library/docs/tree/master/lightstreamer. Accessed 2022.

[37] Max Kellermann. 2022. The Dirty Pipe Vulnerability. https://dirtypipe.cm4all.com. Accessed 2022.

[38] Nick Chase. 2022. Kubernetes multi-container pods and container communication. https://www.mirantis.com/blog/multi-container-pods-and-container-communication-in-kubernetes/. Accessed 2022.

[39] Patrick Hunt. 2022. ZooKeeper Smoketest. https://github.com/phunt/zk-smoketest. Accessed 2022.

[40] RabbitMQ. 2022. RabbitMQ PerfTest. https://rabbitmq.github.io/rabbitmq-perf-test/stable/htmlsingle/. Accessed 2022.

[41] Rancher. 2021. Overview of RancherOS. https://rancher.com/docs/os/v1.x/en/. Accessed 2021.

[42] Red Hat, Inc. 2021. Chapter 5. Red Hat Enterprise Linux CoreOS (RHCOS). https://access.redhat.com/documentation/en-us/openshift_container_platform/4.1/html/architecture/architecture-rhcos. Accessed 2021.

[43] Red Hat, Inc. 2021. kpatch - live kernel patching. https://www.aquasec.com/. Accessed 2021.

[44] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)* 15, 1 (2012), 1–34.

[45] Selectel. 2022. Benchmark for load testing FTP servers. https://github.com/selectel/ftpbench. Accessed 2022.

[46] Sysdig, Inc. 2022. Security Tools for Containers, Kubernetes, and Cloud - Sysdig. https://sysdig.com/. Accessed 2022.

[47] Reinhard Tartler, Anil Kurmus, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Daniela Dorneanu, Rüdiger Kapitza, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. Automatic OS Kernel TCB Reduction by Leveraging Compile-Time Configurability. In *8th Workshop on Hot Topics in System Depend.*

[48] The Authors of gVisor. 2022. gVisor. https://gvisor.dev/. Accessed 2022.

[49] Twistlock. 2017. Twistlock 1.7 With New Runtime Defense Architecture. https://www.prnewswire.com/news-releases/twistlock-announces-twistlock-17-with-new-runtime-defense-architecture-300393120.html. Accessed 2022.

[50] Will Glozer. 2022. wrk - a HTTP benchmarking tool. https://github.com/wg/wrk. Accessed 2022.

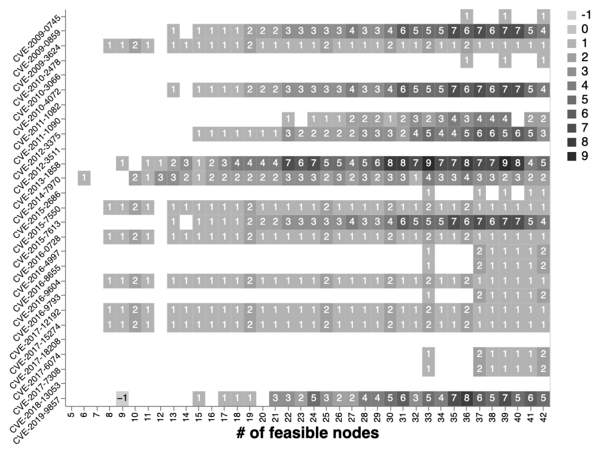
[51] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. 2018. Unikernels as processes. In *Proc. ACM Symposium on Cloud Computing*. 199–211.

[52] Dan Williams, Ricardo Koller, and Brandon Lum. 2018. Say goodbye to virtualization for a safer cloud. In *10th Workshop on Hot Topics in Cloud Computing*.

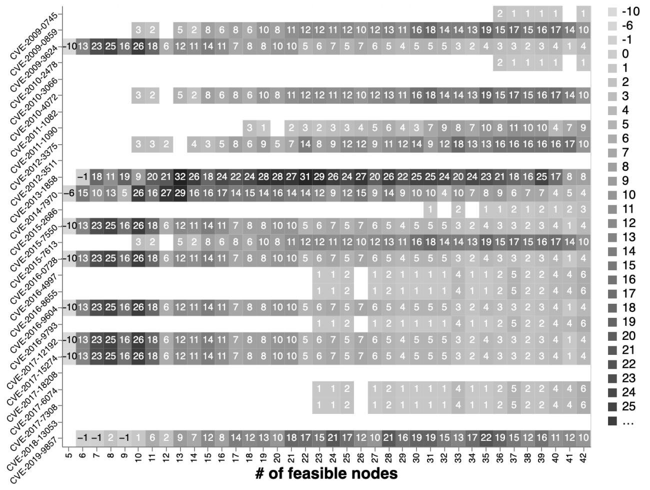
[53] Timothy Wood, Gabriel Tarasuk-Levin, Prashant Shenoy, Peter Desnoyers, Emmanuel Cecchet, and Mark D Corner. 2009. Memory buddies: exploiting page sharing for smart colocation in virtualized data centers. *ACM SIGOPS Operating Systems Review* 43, 3 (2009), 27–36.

[54] Xuebiao Yuchi and S. Shetty. 2015. Enabling security-aware virtual machine placement in IaaS clouds. In *IEEE Military Communications Conf.* 1554–1559.

[55] S. Yu, X. Gui, F. Tian, P. Yang, and J. Zhao. 2013. A Security-Awareness Virtual Machine Placement Scheme in the Cloud. In *IEEE 15th International Conference on High Performance Computing and Communications*. 1078–1083.



(a) Reduction of victim nodes



(b) Reduction of victim containers

Figure 8: Kubernetes default scheduler vs. SySched (using statically generated system call profiles) in terms of reducing the number of (a) victim nodes and (b) victim containers for a particular CVE. Each cell represents how many *additional* victim nodes or containers SySched can reduce compared to the default. Positive cell values (gray to black colors) indicate improvement. The white color indicates no improvement.

A APPENDIX

A.1 Workload Generators

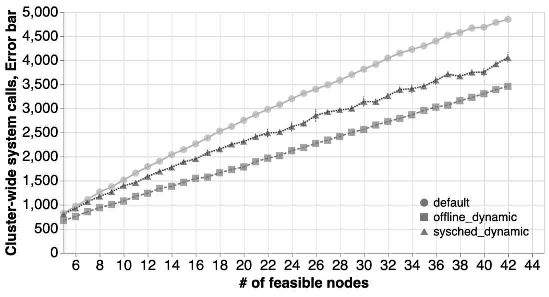
All the workload generators are configured to run for about 60 seconds. The ftpbench [45] uploads a 1MB repeatedly and wrk [50] tool continuously send HTTP requests to servers using 100 connections. Sysbench performs thousands of OLTP type read, write, and update operations. We use a predefined load from YCSB [7] for stressing non-relational databases. `perf-test` [40] publishes messages as quickly as possible. We generate and simulate the time series databases using the influxdb-comparison tool [25]. For UnixBench [28] we run the systems test (e.g., file copy, context switching, process creation, etc.). The zk-smoketest [39] creates zookeeper nodes, attaches and listens to watches on the zookeeper nodes, and delete zookeeper nodes.

Table 3: System calls generated using static analysis and dynamic analysis.

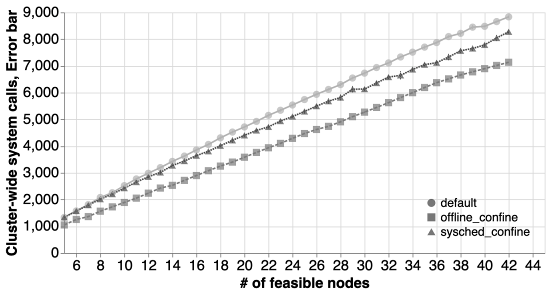
Name	# System calls (static)	# System calls (dynamic)	dynamic / static
python.json	188	46	0.24
nats.json	116	30	0.26
ruby.json	172	48	0.28
hiawatha.json	193	56	0.29
nodejs.json	171	50	0.29
memcached.json	158	53	0.34
cherokee.json	197	71	0.36
alpine.json	180	71	0.39
busybox.json	194	75	0.39
wordpress.json	201	82	0.41
vsftpd.json	201	87	0.43
drupal.json	190	83	0.44
elasticsearch.json	183	80	0.44
golang.json	145	67	0.46
httpd.json	167	76	0.46
gcc.json	145	70	0.48
openjdk.json	184	89	0.48
percona.json	186	90	0.48
proftpd.json	190	91	0.48
dockerreg.json	180	88	0.49
influxdb.json	146	72	0.49
vault.json	204	100	0.49
lightstreamer.json	183	73	0.4
lighttpd.json	204	81	0.4
jenkins.json	191	97	0.51
ubuntu.json	144	74	0.51
centos.json	143	75	0.52
mongodb.json	188	100	0.53
postgres.json	199	105	0.53
redis.json	163	86	0.53
mysql.json	192	105	0.55
rabbitmq.json	186	102	0.55
zookeeper.json	186	107	0.58
ghost.json	186	110	0.59
nginx.json	165	98	0.59
mariadb.json	199	99	0.5
nextcloud.json	198	120	0.61
thttpd.json	116	71	0.61
maven.json	119	79	0.66
docker.json	180	125	0.69
traefik.json	131	90	0.69
gradle.json	119	87	0.73

Table 4: CVE number, the CVE associated system calls, and the number of applications that make use of the associated system calls.

CVE	System calls	# of apps	Applications
CVE-2010-4243	uselib, execve, execveat	41	openjdk, vsftpd, ruby, traefik, mongodb, nextcloud, docker, elasticsearch, golang, maven, percona, influxdb, hiawatha, gradle, jenkins, gcc, lightstreamer, memcached, mysql, httpd, python, drupal, nodejs, ubuntu, cherokee, wordpress, alpine, centos, vault, postgres, thttpd, nginx, ghost, lighttpd, mariadb, proftpd, dockerreg, zookeeper, busybox, redis, rabbitmq
CVE-2018-18281	execve, execveat	41	openjdk, vsftpd, ruby, traefik, mongodb, nextcloud, docker, elasticsearch, golang, maven, percona, influxdb, hiawatha, gradle, jenkins, gcc, lightstreamer, memcached, mysql, httpd, python, drupal, nodejs, ubuntu, cherokee, wordpress, alpine, centos, vault, postgres, thttpd, nginx, ghost, lighttpd, mariadb, proftpd, dockerreg, zookeeper, busybox, redis, rabbitmq
CVE-2010-3066	io_submit	3	percona, mysql, mariadb
CVE-2011-1082	epoll_ctl, epoll_pwait, epoll_wait	39	openjdk, ruby, traefik, mongodb, nextcloud, docker, elasticsearch, golang, percona, influxdb, hiawatha, gradle, jenkins, lightstreamer, memcached, mysql, httpd, python, drupal, nodejs, ubuntu, nats, cherokee, wordpress, alpine, centos, vault, postgres, thttpd, nginx, ghost, lighttpd, mariadb, dockerreg, proftpd, zookeeper, busybox, redis, rabbitmq
CVE-2014-7970	pivot_root	14	alpine, centos, vault, thttpd, nginx, ghost, lighttpd, traefik, docker, nextcloud, ubuntu, dockerreg, zookeeper, redis
CVE-2010-3858	execve, execveat	41	openjdk, vsftpd, ruby, traefik, mongodb, nextcloud, docker, elasticsearch, golang, maven, percona, influxdb, hiawatha, gradle, jenkins, gcc, lightstreamer, memcached, mysql, httpd, python, drupal, nodejs, ubuntu, cherokee, wordpress, alpine, centos, vault, postgres, thttpd, nginx, ghost, lighttpd, mariadb, proftpd, dockerreg, zookeeper, busybox, redis, rabbitmq
CVE-2013-1858	unshare	7	vault, ghost, traefik, docker, nextcloud, dockerreg, zookeeper
CVE-2016-3672	execve, execveat	41	openjdk, vsftpd, ruby, traefik, mongodb, nextcloud, docker, elasticsearch, golang, maven, percona, influxdb, hiawatha, gradle, jenkins, gcc, lightstreamer, memcached, mysql, httpd, python, drupal, nodejs, ubuntu, cherokee, wordpress, alpine, centos, vault, postgres, thttpd, nginx, ghost, lighttpd, mariadb, proftpd, dockerreg, zookeeper, busybox, redis, rabbitmq
CVE-2015-7550	keyctl	7	vault, ghost, traefik, docker, nextcloud, dockerreg, zookeeper
CVE-2012-4530	execve, execveat	41	openjdk, vsftpd, ruby, traefik, mongodb, nextcloud, docker, elasticsearch, golang, maven, percona, influxdb, hiawatha, gradle, jenkins, gcc, lightstreamer, memcached, mysql, httpd, python, drupal, nodejs, ubuntu, cherokee, wordpress, alpine, centos, vault, postgres, thttpd, nginx, ghost, lighttpd, mariadb, proftpd, dockerreg, zookeeper, busybox, redis, rabbitmq
CVE-2014-9585	execve, execveat	41	openjdk, vsftpd, ruby, traefik, mongodb, nextcloud, docker, elasticsearch, golang, maven, percona, influxdb, hiawatha, gradle, jenkins, gcc, lightstreamer, memcached, mysql, httpd, python, drupal, nodejs, ubuntu, cherokee, wordpress, alpine, centos, vault, postgres, thttpd, nginx, ghost, lighttpd, mariadb, proftpd, dockerreg, zookeeper, busybox, redis, rabbitmq
CVE-2015-7613	semget, msgget, shmget	1	postgres
CVE-2015-1593	execve, execveat	41	openjdk, vsftpd, ruby, traefik, mongodb, nextcloud, docker, elasticsearch, golang, maven, percona, influxdb, hiawatha, gradle, jenkins, gcc, lightstreamer, memcached, mysql, httpd, python, drupal, nodejs, ubuntu, cherokee, wordpress, alpine, centos, vault, postgres, thttpd, nginx, ghost, lighttpd, mariadb, proftpd, dockerreg, zookeeper, busybox, redis, rabbitmq
CVE-2010-2478	ioctl	40	openjdk, vsftpd, ruby, traefik, mongodb, nextcloud, docker, elasticsearch, golang, maven, percona, influxdb, hiawatha, gradle, jenkins, gcc, lightstreamer, mysql, httpd, python, drupal, nodejs, ubuntu, cherokee, wordpress, alpine, centos, vault, postgres, thttpd, nginx, ghost, lighttpd, mariadb, proftpd, dockerreg, zookeeper, busybox, redis, rabbitmq
CVE-2019-9857	inotify_add_watch	1	gradle
CVE-2017-15274	add_key, keyctl	7	vault, ghost, traefik, docker, nextcloud, dockerreg, zookeeper
CVE-2016-8655	setsockopt	34	openjdk, vsftpd, traefik, mongodb, nextcloud, docker, elasticsearch, maven, percona, influxdb, hiawatha, gradle, jenkins, lightstreamer, memcached, mysql, httpd, drupal, nodejs, nats, cherokee, wordpress, vault, thttpd, nginx, postgres, ghost, lighttpd, mariadb, proftpd, dockerreg, zookeeper, redis, rabbitmq
CVE-2009-0745	ioctl	40	openjdk, vsftpd, ruby, traefik, mongodb, nextcloud, docker, elasticsearch, golang, maven, percona, influxdb, hiawatha, gradle, jenkins, gcc, lightstreamer, mysql, httpd, python, drupal, nodejs, ubuntu, cherokee, wordpress, alpine, centos, vault, postgres, thttpd, nginx, ghost, lighttpd, mariadb, proftpd, dockerreg, zookeeper, busybox, redis, rabbitmq
CVE-2010-4346	execve, execveat	41	openjdk, vsftpd, ruby, traefik, mongodb, nextcloud, docker, elasticsearch, golang, maven, percona, influxdb, hiawatha, gradle, jenkins, gcc, lightstreamer, memcached, mysql, httpd, python, drupal, nodejs, ubuntu, cherokee, wordpress, alpine, centos, vault, postgres, thttpd, nginx, ghost, lighttpd, mariadb, proftpd, dockerreg, zookeeper, busybox, redis, rabbitmq
CVE-2009-3624	keyctl	7	vault, ghost, traefik, docker, nextcloud, dockerreg, zookeeper
CVE-2017-5123	waitid	2	docker, golang
CVE-2015-2686	sendto, recvfrom	30	openjdk, vsftpd, traefik, docker, nextcloud, maven, percona, influxdb, hiawatha, gradle, jenkins, lightstreamer, memcached, mysql, httpd, drupal, nats, cherokee, wordpress, vault, thttpd, nginx, postgres, ghost, lighttpd, mariadb, proftpd, dockerreg, zookeeper, rabbitmq
...	...	...	...
CVE-2017-18208	madvise	30	openjdk, traefik, mongodb, nextcloud, docker, elasticsearch, golang, maven, percona, influxdb, gradle, jenkins, lightstreamer, memcached, mysql, httpd, drupal, nats, cherokee, wordpress, alpine, vault, ghost, lighttpd, mariadb, dockerreg, zookeeper, busybox, redis, rabbitmq
CVE-2016-9604	keyctl	7	vault, ghost, traefik, docker, nextcloud, dockerreg, zookeeper
CVE-2008-3527	execve, execveat	41	openjdk, vsftpd, ruby, traefik, mongodb, nextcloud, docker, elasticsearch, golang, maven, percona, influxdb, hiawatha, gradle, jenkins, gcc, lightstreamer, memcached, mysql, httpd, python, drupal, nodejs, ubuntu, cherokee, wordpress, alpine, centos, vault, postgres, thttpd, nginx, ghost, lighttpd, mariadb, proftpd, dockerreg, zookeeper, busybox, redis, rabbitmq
CVE-2018-13053	clock_nanosleep	5	openjdk, mariadb, nextcloud, ubuntu, elasticsearch
CVE-2017-7308	setsockopt	34	openjdk, vsftpd, traefik, mongodb, nextcloud, docker, elasticsearch, maven, percona, influxdb, hiawatha, gradle, jenkins, lightstreamer, memcached, mysql, httpd, drupal, nodejs, nats, cherokee, wordpress, vault, thttpd, nginx, postgres, ghost, lighttpd, mariadb, proftpd, dockerreg, zookeeper, redis, rabbitmq
CVE-2012-3511	madvise	30	openjdk, traefik, mongodb, nextcloud, docker, elasticsearch, golang, maven, percona, influxdb, gradle, jenkins, lightstreamer, memcached, mysql, httpd, drupal, nats, cherokee, wordpress, alpine, vault, ghost, lighttpd, mariadb, dockerreg, zookeeper, busybox, redis, rabbitmq
CVE-2010-4072	shmctl	1	postgres
CVE-2016-0728	add_key, request_key, keyctl	7	vault, ghost, traefik, docker, nextcloud, dockerreg, zookeeper
CVE-2011-1090	removexattr, lremovexattr, fremovexattr, setxattr, lsetxattr, fsetxattr	3	vault, docker, nginx
CVE-2012-3375	epoll_ctl	33	openjdk, traefik, mongodb, nextcloud, docker, golang, influxdb, gradle, jenkins, lightstreamer, memcached, mysql, httpd, drupal, nodejs, ubuntu, nats, cherokee, wordpress, alpine, centos, vault, postgres, thttpd, nginx, ghost, lighttpd, mariadb, dockerreg, zookeeper, busybox, redis, rabbitmq
CVE-2017-12192	keyctl	7	vault, ghost, traefik, docker, nextcloud, dockerreg, zookeeper
CVE-2017-14954	waitid	2	docker, golang
CVE-2013-0914	execve, execveat	41	openjdk, vsftpd, ruby, traefik, mongodb, nextcloud, docker, elasticsearch, golang, maven, percona, influxdb, hiawatha, gradle, jenkins, gcc, lightstreamer, memcached, mysql, httpd, python, drupal, nodejs, ubuntu, cherokee, wordpress, alpine, centos, vault, postgres, thttpd, nginx, ghost, lighttpd, mariadb, proftpd, dockerreg, zookeeper, busybox, redis, rabbitmq
CVE-2017-6074	setsockopt	34	openjdk, vsftpd, traefik, mongodb, nextcloud, docker, elasticsearch, maven, percona, influxdb, hiawatha, gradle, jenkins, lightstreamer, memcached, mysql, httpd, drupal, nodejs, nats, cherokee, wordpress, vault, thttpd, nginx, postgres, ghost, lighttpd, mariadb, proftpd, dockerreg, zookeeper, redis, rabbitmq
CVE-2009-0859	shmctl	1	postgres
CVE-2016-9793	setsockopt	34	openjdk, vsftpd, traefik, mongodb, nextcloud, docker, elasticsearch, maven, percona, influxdb, hiawatha, gradle, jenkins, lightstreamer, memcached, mysql, httpd, drupal, nodejs, nats, cherokee, wordpress, vault, thttpd, nginx, postgres, ghost, lighttpd, mariadb, proftpd, dockerreg, zookeeper, redis, rabbitmq



(a) With system call profiles dynamic profiling



(b) With system call profiles static profiling

Figure 9: Average total system calls used by all containers across all nodes for a particular cluster size