Comparing Isolation Mechanisms with *OSmosis*

Sidhartha Agrawal
University of British Columbia
Vancouver, Canada

Shaurya Patel
University of British Columbia,
Google
Vancouver, Canada

Arya Stevinson*
Oracle Labs
Vancouver, Canada

Linh Pham*
Hammerspace
Toronto, Canada

Ilias Karimalis
University of British Columbia
Vancouver, Canada

Hugo Lefeuvre
University of British Columbia
Vancouver, Canada

Aastha Mehta
University of British Columbia
Vancouver, Canada

Reto Achermann
University of British Columbia
Vancouver, Canada

Margo I. Seltzer
University of British Columbia
Vancouver, Canada

Abstract

There exist many mechanisms, ranging from processes to virtual machines, for isolating untrusted computations from each other. Each mechanism explicitly isolates certain resources while, either implicitly or explicitly, sharing the rest. Unfortunately, we lack a comprehensive way to formally and systematically reason about which resources are shared, to what extent they are shared, and how this sharing determines the degree of isolation between any two computations.

We present *OSmosis*, a model that enables reasoning about the precise set of resources shared between two protection domains. The *OSmosis* model represents resources, protection domains, and their relationships as a graph. This graph exposes interactions that affect the confidentiality, integrity, and availability of a protection domain.

We present a tool that extracts the *OSmosis* graph on Linux, using information available through `procfs`. We demonstrate the utility of the model by using it to identify how four popular container implementations differ from one another in terms of the resources shared and trusted processes.

CCS Concepts: • **Software and its engineering** → **Operating systems**; • **Security and privacy** → *Virtualization and security*.

Keywords: Containers, Virtual Machines, Isolation mechanisms, Trusted Computing Base, Formal model

ACM Reference Format:
Sidhartha Agrawal, Shaurya Patel, Arya Stevinson, Linh Pham, Ilias Karimalis, Hugo Lefeuvre, Aastha Mehta, Reto Achermann, and Margo I. Seltzer. 2025. Comparing Isolation Mechanisms with *OSmosis*. In *13th Workshop on Programming Languages and Operating Systems (PLOS '25), October 13–16, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3764860.3768325

*Work done while at the University of British Columbia.

PLOS '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-2225-7/25/10
https://doi.org/10.1145/3764860.3768325

1 Introduction

From the moment that more than one person wanted to use a computer at the same time (some 60 years ago), the systems community has developed myriad techniques to facilitate safe multiplexing. The community continues to develop new mechanisms to facilitate isolation and sharing of different resources among applications and their users [4, 5, 7, 10, 12, 13, 25, 31, 41, 50, 54]. For example, the rise of serverless computing, where low startup latency and strong isolation are both desired, inspired many new mechanisms in search of one that would achieve the isolation of virtual machines with the overhead and startup latency of containers. This plethora of mechanisms makes choosing the correct one challenging; each one prioritizes a different goal, e.g., improving performance [7, 13, 42, 43, 66], improving security [40, 41, 50], or providing reproducibility in developer workflows [2, 5, 19].

Choosing the right isolation mechanism for deploying an application involves many factors (e.g., ease of use, cost, trust assumptions, security requirements, performance), which requires a way for developers to meaningfully compare the mechanisms with respect to these factors [58]. While developers can rely on design specifications and code documentation to determine how easy a mechanism would be to use and deploy, these only vaguely specify the isolation guarantees provided. Consequently, developers resort to ad-hoc interactions with engineering teams to better understand isolation guarantees while making deployment decisions, a process that is laborious and error-prone [57, 58].

The root cause of this problem lies in the absence of a principled approach to identify application state, what parts of it are shared with or isolated from other applications, and which part of the system enforces isolation. While some application state is well understood (e.g., heap, code, data, files), an application might share a significant amount of its state with other applications (e.g., system-level services, files in different namespaces, caches); this sharing is not well understood. Worse yet, the details of what is shared depend on the isolation mechanism's configuration, its implementation, and/or the underlying hardware topology. A lack of understanding of this shared state leads to many problems, ranging from performance anomalies due to unintentional sharing [68], overheads from too much isolation [68], vulnerabilities caused by unintentional sharing of state [72], and unclear trusted computing base [24, 36].

We present the *OSmosis* model (Sec. 3), a formal model that unambiguously describes sharing and isolation of resources between tasks on a system, enabling developers to reason about these aspects. The *OSmosis* model is a graph, with nodes corresponding to *resources*, *protection domains* (PD) and *resource spaces*, which are the context for the allocation of the resources. The edges of the graph precisely describe how nodes interact with each other.

Queries on the *OSmosis* model graph (Sec. 4) reveal how the isolation of PDs varies under different isolation mechanisms. Query results allow us to compute metrics that estimate high-level confidentiality, integrity, and availability properties. For example, we contribute queries to compute the *Trusted Computing Base* (TCB) and the *Impact Boundary* (IB) metrics. In the *OSmosis* model, the TCB of a protection domain PD_x is the set of PDs on which PD_x relies. The IB of PD_x is the set of PDs that can be affected by the behavior of PD_x. These definitions are grounded in Miller's notion of "reliance set" [55], which we formalize with *OSmosis* queries.

We introduce *LinTool* (Sec. 5), a tool that extracts the runtime *OSmosis* model state from Linux by querying the `/proc` pseudo filesystem. We use it to illustrate how the TCB and IB vary across four container mechanisms—Docker, Docker in rootless mode, Apptainer, and Podman (Sec. 5). For example, we show that the daemon-less mechanisms and Docker rootless mode have less impact (e.g., cannot restart the OS) on the host kernel relative to Docker regular mode.

Overall, we contribute a new approach to quantify the isolation provided by different mechanisms. This enables many future works, such as building new metrics to evaluate the confidentiality, integrity, and availability of systems, or designing operating systems that facilitate the extraction of *OSmosis* models (Sec. 7).

2 Anatomy of a Usecase

We use the running example of an application using a Key-Value Store (KVS) to both motivate the need for a model to describe isolation and later to demonstrate how the *OSmosis* model helps us precisely compare isolation mechanisms.

Fig. 1 illustrates six different application configurations. (a) The application links with the KVS library in a single process, so there is no isolation between the application and the KVS, and `Get` and `Set` operations are simple function calls. (b) The application and KVS are part of the same process, but they are compartmentalized [46], sharing only a limited number of pages, so `Get` and `Set` require compartment switching. The shared pages are explicitly set up during application initialization. (c) The application and the KVS run in separate processes on the same OS, `Get` and `Set` operations require IPC via a socket. (d) The application and the KVS run in separate containers on the same host (the file resource is no longer shared as they are in separate mount namespaces). (e) The application and the KVS run in separate processes, each in a separate virtual machine hosted on a shared hypervisor. (f) The application and the KVS run in separate unikernel [43, 52] virtual machines on a shared hypervisor, which is shown by each of the application and the KVS merged with their unique copies of the OS. In the last three configurations, the application and the KVS communicate over the network.

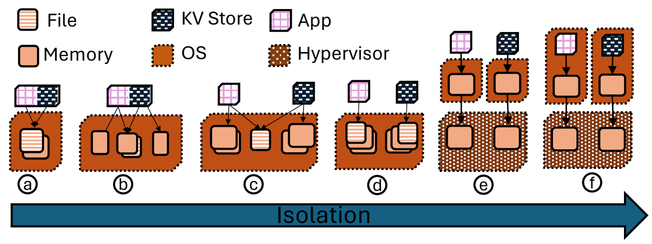


Figure 1. Some options for deploying an application and a Key-Value (KV) Store in: (a) the same process, (b) the same process with intra-address space compartmentalization, (c) separate processes (sharing files, but not memory), (d) separate containers (not sharing files), (e) separate processes inside VMs, and (f) separate unikernel VMs. Scenarios (a) and (b) communicate via shared memory, (c) communicates via IPC and (d) to (f) communicate via the network.

2.1 Two Perspectives on Isolation

It is widely accepted that scenarios (a) to (e) offer increasing isolation between the application and the KVS. But what exactly makes a scenario more isolated than the previous one? The answer to this question is rooted in two different perspectives: (a) the *application developer* and (b) the *infrastructure provider*.

Consider the scenario of a developer deploying the KVS application on a shared cloud infrastructure. How should they choose and justify an isolation mechanism? They would care about the *Trusted Computing Base* (TCB), i.e., the parts of the system that can crash (kill) or stall their application, the parts that have access (read or write) to their application's data, and those that provide system services for the application. For example, when running in a container, the TCB includes the host kernel, orchestration middleware (e.g., Kubernetes), and other containers that can exhaust shared resources and disrupt the application [35, 51, 72, 73].

In contrast, cloud providers focus on the *Impact Boundary* (IB), the potential damage a malicious application can wreak.

For instance, can it crash the kernel [1], bring down the hypervisor [20, 22], or escalate privileges to take over the infrastructure [21, 37]? We formally define both TCB and IB in Sec. 4, and link them to existing notions of confidentiality, integrity, and availability.

2.2 Challenges

Determining the TCB and IB of the different configurations is challenging for several reasons.

C1: Multiple implementations of the mechanisms: Multiple implementations often exist for the same isolation mechanism, making it hard to define exactly what isolation each provides. This challenge persists across mechanisms, such as containers, virtual machines, and intra-address space isolation mechanisms.

'Containers' come in many forms, e.g., Docker, Apptainer, Podman, Kata Containers, but lack a consistent definition. As a result, it is unclear which resources are shared in each case [39]. For example, Docker supports two modes: regular [5] and rootless [14]. Regular mode uses a centralized daemon with root privileges to manage namespaces [10], cgroups [3], and overlay file systems [9]. This shared daemon introduces privilege escalation risks. Rootless mode mitigates these risks by letting users run both the daemon and containers without root privileges. Apptainer [2], unlike Docker, runs containers as the invoking user and avoids a centralized daemon. This reduces the attack surface and simplifies integration in multi-user environments. However, both Docker and Apptainer rely on kernel namespaces, so they share the host kernel with other containers. In contrast, Kata Containers [59] use hardware virtualization, avoiding namespace-related risks as they do not share the kernel.

The presence or absence of a centralized daemon changes the TCB of each mechanism, while differences in the way they share resources affect their IB. Newer implementations for emerging use cases [6, 19, 70] introduce further subtle variations, complicating comparisons even more.

C2: Correct mechanism configuration: Isolation mechanisms often have configuration options that can be set prior to execution. Even a specific implementation such as Docker has configuration parameters that change the isolation guarantees it provides. For example, it is common for Docker containers to be configured such that they share files (e.g., environment configurations) to enable access to the Docker daemon from inside the container [53, 56], allowing unintended interaction between containers. A class of path misresolution vulnerabilities, such as symlink resolution cheating and inducing illegal file execution, arise due to incorrect configuration of file sharing between containers [47]. While mechanisms come with default configurations, even minor changes can significantly impact their TCB and IB.

Takeaways: Isolation mechanisms, the implementation variants of a mechanism, and their deployment configurations differ in subtle ways. This makes it hard for developers to understand the isolation guarantees isolation mechanisms provide, pushing developers to rely on ad-hoc methods to choose a mechanism for their deployment [29, 44, 71]. This lack of a principled approach leads to unpleasant surprises, such as runtime performance anomalies, and vulnerabilities affecting confidentiality, availability, and integrity [39].

```
System Graph :- { nodes:Set<Node>, edges:Set<Edge> }
Node :- ProtectionDomain | Resource(ResourceType)
      | ResourceSpace(ResourceType)
Edge :- (Node, Node, EdgeType, Set<EdgeAttribute>)
EdgeType :- Hold | Map | Request | Subset
EdgeAttribute :- ResourceType | Permissions
ResourceType :- Virtual Addr | DRAM Page
          | Page Quota | File | Directory
Permissions :- Read | Write | Execute | Terminate
```

Listing 1. *OSmosis* Isolation Model

3 *OSmosis* Model

We present the *OSmosis* model to precisely describe the current state of a system in terms of its protection domains, resources and their relationships.

3.1 Design Goals

The first design goal of the *OSmosis* model is to express and reason about sharing of resources between protection domains. The state of most real-world systems can change at runtime, e.g., by creating a process or sharing a page. We design *OSmosis* to represent a static snapshot of the system *at a given point in time*. Each variation of a system's state can be represented with a different instance of the model.

The second design goal is to express and reason about the implications of a system's sharing profile on availability. The model achieves that by modeling hard quotas and allocation limits, which are common mechanisms to prevent resource exhaustion. However, we must maintain a balance between the precision of the model and its usability. Thus, we do not express soft policies for resource allocation, allowed information flow, or cleanup strategies.

3.2 Model Definition

OSmosis models the system as a graph. Listing 1 shows the model definition, and Fig. 2 illustrates how the model expresses a process in a UNIX-like OS.

3.2.1 Nodes: The graph has three types of nodes. *Protection domains (PDs)* correspond to an execution context, such as a process, container or virtual machine. Fig. 2 shows a 'Process' and a 'Kernel' PD as elongated hexagons. *Resource spaces* provide pools for resource allocations, e.g., a virtual address space, DRAM pages, or cgroups [3]. Fig. 2 shows two resource spaces: '`virtaddr`:1' and 'DRAM:0' as rounded rectangles. *Resources* are passive entities, such as a virtual page (`virtaddr`), DRAM page, or file. Fig. 2 shows a "`virtaddr`:code" resource in an oval node. Historically, these are called objects [45, 63]. Resource and resource space nodes have a type defining the kind of resource they represent. Fig. 2 shows resources and resource-spaces of type `virtaddr` and DRAM.

3.2.2 Edges: There are four kinds of edges. *Hold-edges* indicate that a PD currently holds rights over a resource (e.g., 'Process' holds two virtual pages in Fig. 2), can allocate from or change a resource space (e.g., 'Kernel' can allocate from 'DRAM:0'), or controls another PD (e.g., 'Kernel' can terminate 'Process'). *Request-edges* between PDs signify that the source PD can request resources of specific types from the destination PD (e.g., a process can call `mmap()` to request virtual memory i.e., resource type `virtaddr` from the kernel in Fig. 2). *Subset-edges* indicate from which resource space a resource is allocated (e.g., in Fig. 2 the virtual pages for code and heap indicated by `virtaddr`:code and `virtaddr`:heap are part of the virtual address space `virtaddr`:1). All resource nodes must have the same type as the resource space from which they are allocated. *Map-edges* connect two resource nodes or two resource space nodes. A map-edge from one resource to another expresses either a fixed mapping determined by the system topology (e.g., how a DRAM address maps to cache sets) or a dynamically created one (e.g., the code page maps to a DRAM page in Fig. 2), A map-edge from one resource space to another indicates that resources from the source map to resources from the destination (e.g., '`virtaddr`:1' maps to 'DRAM:0' in Fig. 2).

3.2.3 Edge Attributes: Some edges carry attributes.

Resource Type: Request-edges have a type specifying what resources a PD can request. Fig. 2 shows the '`virtaddr`' attribute on the request-edge between 'Process' and 'Kernel'. Listing 1 shows the resource types discussed in this paper, but the model generalizes to other types of resources.

Permission: Hold-edges have permissions indicating what a PD can do with a resource or another PD (e.g., in Fig. 2, 'Process' has read and execute permissions on the code page, and 'Kernel' has the permission to terminate 'Process').

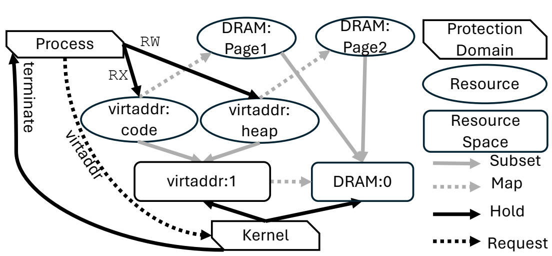


Figure 2. *OSmosis* model graph for process and kernel PDs in a UNIX-like OS. The process's code and heap consist of a single page. The request-edge allows the process to ask for additional virtual memory resources (of type `virtaddr`) from the kernel. For simplicity, we exclude the process data segment and most OS resources.

3.3 Model Invariants

Graphs must satisfy invariants to be valid *OSmosis* models. This is to ensure that *OSmosis* models represent only systems that are meaningful and instantiable.

- Resource nodes must connect to a resource space of the same type via a subset edge and must be reachable from a PD via hold edge.
- Resource space nodes must be reachable from a PD node via a hold edge.
- Request edges exist only between two PD nodes, and their resource types must exist in the graph.
- Hold edges originate at a PD node.
- Map edges exist only between two resource nodes or between two resource space nodes.
- A map edge between two resource nodes exists only if their corresponding resource spaces are also connected by a map edge.

4 Applying the Model

The results of querying the model enable reasoning about isolation and sharing with different mechanisms. We first describe a parameterized breadth-first search (BFS, Sec. 4.1) that constitutes our fundamental query building block. We then show how BFS queries compose into larger *OSmosis* queries that identify interactions between PDs in terms of confidentiality, integrity, and availability (Sec. 4.2). Finally, we show how *OSmosis* queries compute the TCB and IB of a protection domain (Sec. 4.3).

4.1 The *OSmosis* Query Building Block

The parameterized BFS on an *OSmosis* graph starts from a PD_x and produces a complete picture of the resources, resource spaces, and other PDs with which PD_x interacts.

Depending on the question we want to answer, we may limit the traversal based on `EdgeTypes` (e.g., request-edge to a PD that provides resources), `EdgeDirection` (e.g., traverse request-edges in reverse to find which PDs request resources from PD_x), `AccessMode` (e.g., permission on hold-edge), and `Depth` (e.g., ∞ for a full BFS or 1 for a neighborhood query). Sometimes, we also want to filter the results based on `NodeType` (e.g., to find either PDs, resources, or resource-spaces) or `ResourceType` (e.g., to identify all memory resources accessible to a PD). Our BFS primitive takes parameters for each of these traversal and filtering criteria:

```
BFS(PDx, EdgeTypes, EdgeDirection, AccessMode,
         Depth, NodeTypes, ResourceTypes)
```

4.2 Identifying PD interactions

We now illustrate how the BFS query identifies the PDs that affect a given PD's confidentiality, integrity, and availability.

4.2.1 Shared Resources:
PDs that can access the same resources as PD_x, either directly or indirectly through mappings, can compromise PD_x's confidentiality (if they have read access) or integrity and availability (with write access). We compute this set of PDs by iteratively intersecting the results of two BFS traversals following only hold-edges and map-edges, one starting at PD_x and one at PD_i for i over all PDs of the system. The result of each intersection is a set of resources. If this set is non-empty, then PD_i has access to some of the resources of PD_x, and we add PD_i to the final result set.

$$\begin{aligned}&\textbf{SharedResources}(PD_x,\ \texttt{ResourceTypes},\ \texttt{AccessMode}) =\\ &\quad\{\ PD_i\ \mid\ PD_i \neq PD_x \wedge (\\ &\qquad BFS(PD_x,\ \{\texttt{hold},\ \texttt{map}\},\ \texttt{fwd},\ \texttt{ANY},\ \infty,\ \{\texttt{Resource}\},\\ &\qquad\qquad\qquad \texttt{ResourceTypes})\\ &\qquad\quad \cap\ BFS(PD_i,\ \{\texttt{hold},\ \texttt{map}\},\ \texttt{fwd},\ \texttt{AccessMode},\ \infty,\\ &\qquad\qquad\qquad\quad \{\texttt{Resource}\},\ \texttt{ResourceTypes}) \neq \emptyset\ )\ \}\end{aligned}$$

4.2.2 PD Control:
A PD can control the execution of another PD. For example, the kernel PD can suspend or terminate a process PD, hence affecting the availability of the process PD. The *OSmosis* model expresses this as hold-edges between PDs, so we can find the set of relevant PDs using BFS traversal on the hold-edges to PD nodes.

$$\begin{aligned}&\textbf{CanControl}(PD_x) = BFS(PD_x, \{\texttt{hold}\},\ \texttt{rev},\ \texttt{ANY},\ 1, \{\texttt{PD}\},\ \texttt{ANY})\\ &\textbf{ControlBy}(PD_x)\ \ = BFS(PD_x, \{\texttt{hold}\},\ \texttt{fwd},\ \texttt{ANY},\ 1, \{\texttt{PD}\},\ \texttt{ANY})\end{aligned}$$

4.3 Trusted Computing Base and Impact Boundary

We provide a definition of TCB from the literature and then show how we can define the TCB using the *OSmosis* model and the queries we just defined. Finally, we define the impact boundary (IB) of a PD that quantifies how much a PD can affect other PDs.

4.3.1 Trusted Computing Base (TCB).
There exist several definitions of TCB [46, 62, 64, 69]. We follow Miller's definition [55] of the TCB of PD_x as its "*reliance set*", i.e., all the PDs on which PD_x relies for its own correct behavior. A PD "*relying*" on another is vague. We make it precise by defining the reliance set of a PD_x as the set of PDs that can violate the confidentiality, integrity or availability of PD_x. We combine the *OSmosis* queries to compute the TCB.

$$\begin{aligned}\textbf{TCB}(PD_x, types, mode) &= \texttt{SharedResources}(PD_x, types, mode)\\ &\quad \cup\ \texttt{CanControl}(PD_x)\end{aligned}$$

This definition can be specialized to specific resource types and access modes.

4.3.2 Impact Boundary (IB).
A faulty or malicious PD_x can violate the confidentiality, integrity, and availability of other PDs in the system, and those PDs comprise the IB of PD_x. We again use Miller's reliance set to define a PD's IB. The only difference is that we want the PDs *controlled by* PD_x rather than those that can control PD_x, so computing IB is nearly identical to computing TCB. We combine *OSmosis* queries to compute the IB. As with the TCB, we can also specialize the IB by limiting the resource types.

$$\begin{aligned}\textbf{IB}(PD_x, types, mode) &= \texttt{SharedResources}(PD_x, types, mode)\\ &\quad \cup\ \texttt{ControlBy}(PD_x)\end{aligned}$$

There are other ways a PD can interact with other PDs that affects its confidentiality, integrity, and availability. We discuss these interactions and how to extend TCB and IB to account for them in Sec. 7.

5 *OSmosis* Evaluation

We show that *OSmosis* model is 1) practical and 2) expressive.

For 1) we build *LinTool*, a Python script that extracts relevant information (e.g., user permissions, signals, Linux capabilities [60]) from the `/proc` system. *LinTool* demonstrates that production-grade operating systems such as Linux already maintain the information needed to construct the *OSmosis* model and that this information can be accessed from user-space without requiring any kernel modifications. *LinTool* models processes as PDs and the kernel as a special PD, PD_k. We add a hold-edge from a PD_x to PD_y if the corresponding process x can send a signal (e.g., SIGKILL) to process y. We add a hold-edge with its corresponding permissions from a PD_x to a resource R_y (e.g., file) if a process x can access the resource R_y. *LinTool* uses the `networkx` Python package [11] to maintain and query the graphs. The parameterized Bread First Search (BFS) query from Sec. 4.1 is written as a `networkx` BFS function, which is in turn used to construct the TCB and IB functions in Python. We run *LinTool* on an Intel Xeon W-2275 with 14 cores with hyperthreading enabled and 128 GB RAM. The host OS is Ubuntu 24.04 with kernel `v6.5.3`.

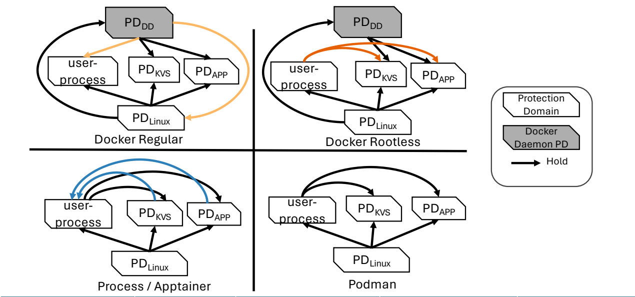
For 2), we compare processes and four container variants. We show that we can use the model queries to answer two non-trivial questions about scenarios involving different container implementations (Sec. 5.1). Specifically:

- In what scenarios can PDs control each other? (Sec. 5.2)
- How are resources shared? (Sec. 5.3)

5.1 Evaluation Scenario Setup

We compare five scenarios from Sec. 2 using standard processes, Apptainer [2], Podman [19], and Docker (regular and rootless) [14]. PD_{App} and PD_{KVS} communicate via GRPC [18]. We built minimal images for each scenario and launched them with default commands. In each setup, we deployed both PDs and used *LinTool* to extract model graphs. Since each setup runs a full Linux environment, *LinTool* captures hundreds of processes. We focus only on subgraphs connected to the PDs of interest.

We compare each mechanism's TCB and IB as defined in Sec. 4.3. Recall that TCB and IB use two sub-queries. As subqueries yield identical results, we highlight only those that show differences. Thus, figures in these sections show partial model graphs—limited to nodes and edges relevant to the varying confidentiality, integrity, or availability properties.



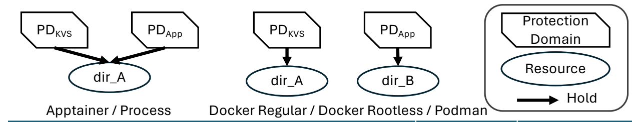
Query	Docker	Docker RL	Apptainer / Process	Podman
CanControl(PD_{KVS})	PD_{DD}, PD_{Linux}	PD_{DD}, $PD_{user\text{-}proc}$, PD_{Linux}	$PD_{user\text{-}proc}$, PD_{Linux},	$PD_{user\text{-}proc}$, PD_{Linux},
ControlBy(PD_{KVS})	∅	∅	$PD_{user\text{-}proc}$	∅

Figure 3. *OSmosis* graphs showing the results of the "CanControl' and "ControlBy" queries, highlighting the relevant hold-edges between PDs. Processes and Apptainer are identical and shown together on the bottom left. PD_{DD} represents the Docker Daemon. Colored edges highlight the differences between mechanisms across the columns in the same row.

5.2 How PDs control each other

Fig. 3 shows which PDs can control each other (by virtue of having hold-edges) to affect availability in the five scenarios. The extra PD_{DD} in the top row corresponds to Docker's daemon process; Apptainer and Podman do not have a daemon process (bottom row). In Docker's regular mode (top left), the hold-edge from PD_{DD} to PD_{Linux} (the kernel) indicates that a compromise in the Docker daemon (which runs with root privileges and has CAP_SYS_BOOT capability) can affect the kernel. In contrast, the absence of such an edge from PD_{DD} to PD_{Linux} (top right) indicates that the (non-root) Docker daemon has a lower impact on the kernel in Docker's rootless mode. In Docker's regular mode (top left), the edges from PD_{DD} to PD_{App}, PD_{KVS}, and "user process" indicate that the Docker daemon controls the App and KVS containers in addition to all the users' processes. In Docker's rootless mode (top right), the daemon controls only the containers it started, not all users' processes.

Broadening our discussion to Apptainer and Podman (bottom row), the hold-edges from "user process" to PD_{App} and PD_{KVS} in Docker rootless, Process/Apptainer, and Podman indicate that the App and KVS containers can be killed by any process running as the user running the containers. In contrast, the absence of these edges in Docker's regular mode indicates that the user's other processes cannot kill the containers in this configuration. Unlike with Docker and Podman, when the App and KVS are run as either Apptainer containers or standard processes, they can also kill other processes of the same user (indicated by blue hold-edges in the bottom left). The same is impossible in the other configurations, because the containers in these mechanisms run in separate PID namespaces.



Query	Apptainer / Process	Docker Regular Docker Rootless /Podman
SharedResources(PD_{KVS}, {dir-resource}, {R, W})	PD_{APP}	∅

Figure 4. *OSmosis* graphs showing differences in the directory resource sharing for five isolation mechanisms on Linux. The table shows the differences in TCB and IB for PD_{KVS} in these scenarios that arise when comparing results of the "SharedResources" query.

Takeaway: *OSmosis* graphs clearly illustrate how adding a daemon that runs as root increases the IB. The daemonless mechanisms and Docker rootless mode have less impact on the host kernel compared to Docker regular mode. In contrast, processes and containers can directly impact each other in all modes except Docker regular mode.

5.3 How resources are shared

There are many resources in each isolation mechanism. In the interest of space, we focus on discussing the filesystem directory resource, which illustrates key differences in container configurations. Fig. 4 (left) shows that both Apptainer instances and standard processes share directories of the same user, because the user's home directory is shared with the container by default. Docker and Podman (right) container instances do not share the home directory by default, although changing configuration options allows for the creation of the shared directory setup of Apptainer.
Takeaway: With Apptainer and a standard process isolation mechanism, all processes of the same user appear in the TCB and IB of PD_{KVS} for file resources; thus, they can violate the confidentiality and integrity of the KVS.

Both observations match the recommendations of container security blog posts [29, 71] and prior academic studies [39]. This shows that *OSmosis* graphs are expressive enough to extract useful, non-trivial differences between container technologies.

6 Related Work

Modeling operating systems has been an active area of research since the 1970s. The proposed models vary based on the question they are intended to answer. Our model is heavily inspired by the models built to answer questions about protection in an operating system [27, 28, 32, 38, 45, 46, 48, 49, 55, 61, 63, 65] as well as the models built specifically to verify operating systems [23, 62, 67]. Our model adopts the ideas of protection-domains (aka principals) and resources (aka objects) that appear in all prior models. However, we add the notion of mappings, which let us obtain a closure of all the resources accessible to a protection domain. Models for verification often focus exclusively on tangible resources (e.g., physical memory), whereas *OSmosis* also models abstract resources (e.g., file, quota). Sockeye [33, 34] creates a model expressing which resources can be accessed from which cores. It explicitly models who can change the configuration of the memory hardware, e.g., by writing to the page tables. This corresponds to changing map-edges in the *OSmosis* Graph. Similar to models built for verification, Sockeye focuses primarily on physical memory resources, whereas *OSmosis* also considers other resources (e.g., virtual memory, page quota, files, directories). Tracking information flow [32] is out of scope for us, so if a resource is shared, we assume that information flow can happen through it.

Prior work has presented approaches that compare protection for a subset of the information we track [30, 69]. For instance, Liang et al. [30] compare access control mechanisms across different operating systems. While similar in motivation to our work, we focus on comparing different isolation mechanisms in their entirety, which includes resource quota isolation and hierarchical protection domains.

Our definitions for TCB and IB are grounded in Miller's [55] definition of "reliance set". We add precision to the notion of "a PD being vulnerable to another PD," by identifying CIA violations by composing *OSmosis* queries. This composition also allows the user to define a narrower definition of TCB, containing only PDs enforcing specific policies [64, 69]. Our approach of using the flexible definitions of TCB and IB to compare the mechanisms is rooted in the different concerns of the developer and the infrastructure provider.

7 Future Work

Extending TCB and IB with other interactions: PDs can affect each other in three major ways apart from the interactions we already discussed: (1) by being the PD that is managing a resource used by another PD (i.e., being a resource server), (2) by sharing a resource server (3) by sharing a resource space with another PD. The TCB and IB definitions can be easily extended to capture these interactions.

Enhancing *LinTool*: Our current implementation of *LinTool* extracts information related only to a process' ability to send signals and its access to file system resources. It can be extended by adding information from other subsystems (e.g., networking) and resource control mechanisms (e.g., cgroups [3], seccomp [15], SELinux [16]), allowing us to compare more mechanisms such as VMMs.

***OSmosis* focused OS:** Linux-based tools can only approximate the *OSmosis* model, as no single source provides a complete system view. A capability-based kernel (e.g., seL4 [17], Barrelfish [26], Genode [8]) could enable direct extraction of the full *OSmosis* model, since capabilities explicitly record rights and interactions, simplifying graph construction.

Design Space Exploration: *OSmosis* provides a way to express isolation mechanisms as graphs. By exploring different graph structures and edge attributes, we can explore the design space of isolation mechanisms. As the number of possible graphs is practically infinite, we need a way to set meaningful starting points (e.g., an existing mechanism) and termination strategies (desired TCB and IB).

8 Conclusion

The lack of a principled way to express the level of isolation and sharing between different OS abstractions and isolation mechanisms makes comparing them challenging. We present the *OSmosis* model, which lets us precisely state what resources are shared between applications. This lets us compare and reason about the explicit and implicit sharing between applications in a principled way using TCB and IB. Our analysis on Linux showed that we can extract useful instances of the *OSmosis* model and uncover non-trivial differences between container technologies.

References

[1] [n. d.]. All Linux CVEs. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=linux [Accessed 10-09-2025].
[2] [n. d.]. Apptainer User Guide. https://apptainer.org/docs/user/main/. [Accessed 10-09-2025].
[3] [n. d.]. cgroups(7) – Linux manual page. https://man7.org/linux/man-pages/man7/cgroups.7.html [Accessed 10-09-2025].
[4] [n. d.]. clone(2) – Linux manual page. https://man7.org/linux/man-pages/man2/clone.2.html [Accessed 10-09-2025].
[5] [n. d.]. Docker. https://docs.docker.com/ [Accessed 10-09-2025].
[6] [n. d.]. Firejail Security Sandbox. https://firejail.wordpress.com/. [Accessed 10-09-2025].
[7] [n. d.]. FreeBSD Manual Pages: jail. https://www.freebsd.org/cgi/man.cgi?jail [Accessed 10-09-2025].
[8] [n. d.]. Genode Operating System Framework. https://genode.org/ [Accessed 10-09-2025].
[9] [n. d.]. The Linux Kernel documentation: Overlay Filesystem. https://docs.kernel.org/filesystems/overlayfs.html. [Accessed 10-09-2025].
[10] [n. d.]. namespaces(7) – Linux manual page. https://man7.org/linux/man-pages/man7/namespaces.7.html [Accessed 10-09-2025].
[11] [n. d.]. NetworkX: Network Analysis in Python. https://networkx.org/. [Accessed 18-04-2025].
[12] [n. d.]. OpenVZ Container. https://wiki.openvz.org/Container [Accessed 10-09-2025].
[13] [n. d.]. Oracle Solaris Information Library: zones(5). https://docs.oracle.com/cd/E36784_01/html/E36883/zones-5.html#REFMAN5zones-5 [Accessed 10-09-2025].
[14] [n. d.]. Rootless mode. https://docs.docker.com/engine/security/rootless [Accessed 10-09-2025].
[15] [n. d.]. SecComp BPF Secure Computing with filters. https://www.kernel.org/doc/html/v5.0/userspace-api/seccomp_filter.html. [Accessed 10-09-2025].
[16] [n. d.]. SELinux(8) - Linux manual page. https://man7.org/linux/man-pages/man8/selinux.8.html. [Accessed 10-09-2025].
[17] [n. d.]. The seL4 Microkernel. https://sel4.systems/ [Accessed 10-09-2025].
[18] [n. d.]. What is gRPC? Core concepts, architecture and lifecycle. https://grpc.io/docs/what-is-grpc/core-concepts/. [Accessed 10-09-2025].
[19] [n. d.]. What is Podman? https://docs.podman.io/en/latest/index.html. [Accessed 10-09-2025].

[20] 2019. CVE-2019-19332. https://www.cve.org/CVERecord?id=CVE-2019-19332 [Accessed 10-09-2025].

[21] 2021. CVE-2021-22543. https://cve.mitre.org/cgi-bin/cvename.cgi?name=2021-22543 [Accessed 10-09-2025].

[22] 2021. CVE-2021-43056. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-43056 [Accessed 10-09-2025].

[23] Reto Achermann, Lukas Humbel, David Cock, and Timothy Roscoe. 2018. Physical Addressing on Real Hardware in Isabelle/HOL. In *Interactive Theorem Proving*, Jeremy Avigad and Assia Mahboubi (Eds.). Springer International Publishing, Cham, 1–19.

[24] Markuze Alex, Shay Vargaftik, Gil Kupfer, Boris Pismeny, Nadav Amit, Adam Morrison, and Dan Tsafrir. 2021. Characterizing, exploiting, and detecting DMA code injection vulnerabilities in the presence of an IOMMU. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 395–409. doi:10.1145/3447786.3456249

[25] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 164–177. doi:10.1145/945445.945462

[26] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 29–44. doi:10.1145/1629575.1629579

[27] D Elliott Bell and Leonard J LaPadula. 1973. *Secure computer systems: Mathematical foundations.* Technical Report.

[28] Matt Bishop. 1981. Hierarchical Take-Grant Protection systems. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles* (Pacific Grove, California, USA) *(SOSP '81)*. Association for Computing Machinery, New York, NY, USA, 109–122. doi:10.1145/800216.806598

[29] Davis Catherman. 2022. Why you should use Apptainer. https://medium.com/@dcat52/why-you-should-use-apptainer-21ef1fe7e0bb. [Accessed 10-09-2025].

[30] Liang Cheng, Yang Zhang, and Zhihui Han. 2013. Quantitatively measure access control mechanisms across different operating systems. In *2013 IEEE 7th International Conference on Software Security and Reliability*. IEEE, 50–59.

[31] Fernando J. Corbató, Marjorie Merwin-Daggett, and Robert C. Daley. 1962. An experimental time-sharing system. In *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference* (San Francisco, California) *(AIEE-IRE '62 (Spring))*. Association for Computing Machinery, New York, NY, USA, 335–344. doi:10.1145/1460833.1460871

[32] Dorothy E. Denning. 1976. A lattice model of secure information flow. *Commun. ACM* 19, 5 (May 1976), 236–243. doi:10.1145/360051.360056

[33] Ben Fiedler, Roman Meier, Jasmin Schult, Daniel Schwyn, and Timothy Roscoe. 2023. Specifying the de-facto OS of a production SoC. In *Proceedings of the 1st Workshop on Kernel Isolation, Safety and Verification* (Koblenz, Germany) *(KISV '23)*. Association for Computing Machinery, New York, NY, USA, 18–25. doi:10.1145/3625275.3625400

[34] Ben Fiedler, Daniel Schwyn, Constantin Gierczak-Galle, David Cock, and Timothy Roscoe. 2023. Putting out the hardware dumpster fire. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems* (Providence, RI, USA) *(HotOS '23)*. Association for Computing Machinery, New York, NY, USA, 46–52. doi:10.1145/3593856.3595903

[35] Xing Gao, Zhongshu Gu, Zhengfa Li, Hani Jamjoom, and Cong Wang. 2019. Houdini's Escape: Breaking the Resource Rein of Linux Control Groups. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1073–1086. doi:10.1145/3319535.3354227

[36] Xiling Gong, Peter Pi, and Tencent Blade Team. 2019. *Exploiting Qualcomm WLAN and Modem Over the Air.* Technical Report. https://i.blackhat.com/USA-19/Thursday/us-19-Pi-Exploiting-Qualcomm-WLAN-And-Modem-Over-The-Air-wp.pdf [Accessed 10-09-2025].

[37] Google Project Zero. 2021. Project Zero: An EPYC Escape: Case-study of a KVM breakout. https://googleprojectzero.blogspot.com/2021/06/an-epyc-escape-case-study-of-kvm.html [Accessed 10-09-2025].

[38] G. Scott Graham and Peter J. Denning. 1971. Protection: principles and practice. In *Proceedings of the May 16-18, 1972, Spring Joint Computer Conference* (Atlantic City, New Jersey) *(AFIPS '72 (Spring))*. Association for Computing Machinery, New York, NY, USA, 417–429. doi:10.1145/1478873.1478928

[39] Md Sadun Haq, Thien Duc Nguyen, Ali Şaman Tosun, Franziska Vollmer, Turgay Korkmaz, and Ahmad-Reza Sadeghi. 2024. SoK: A Comprehensive Analysis and Evaluation of Docker Container Attack and Defense Mechanisms. In *Proceedings of the 2024 IEEE Symposium on Security and Privacy (S&P'24)*. IEEE, 4573–4590. doi:10.1109/SP54263.2024.00268

[40] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. 2016. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. Association for Computing Machinery. doi:10.1145/2976749.2978327

[41] Yuzhuo Jing and Peng Huang. 2022. Operating System Support for Safe and Efficient Auxiliary Execution. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*. USENIX Association. https://www.usenix.org/conference/osdi22/presentation/jing

[42] Tom Kuchler, Michael Giardino, Timothy Roscoe, and Ana Klimovic. 2023. Function as a Function. In *Proceedings of the 2023 ACM Symposium on Cloud Computing (SoCC '23)*. Association for Computing Machinery, New York, NY, USA, 81–92. doi:10.1145/3620678.3624648

[43] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. 2021. Unikraft: Fast, Specialized Unikernels the Easy Way. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery. doi:10.1145/3447786.3456248

[44] Rakesh Kumar and B Thangaraju. 2020. Performance Analysis Between RunC and Kata Container Runtime. In *Proceedings of the 2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT'20)*. doi:10.1109/CONECCT50063.2020.9198653

[45] Butler W. Lampson. 1974. Protection. *ACM SIGOPS Operating Systems Review* 8, 1 (jan 1974), 18–24. doi:10.1145/775265.775268

[46] Hugo Lefeuvre, Nathan Dautenhahn, David Chisnall, and Pierre Olivier. 2025. SoK: Software Compartmentalization. In *Proceedings of the 2025 IEEE Symposium on Security and Privacy (S&P'25)*. IEEE Computer Society, Los Alamitos, CA, USA. doi:10.1109/SP61157.2025.00075

[47] Zhi Li, Weijie Liu, XiaoFeng Wang, Bin Yuan, Hongliang Tian, Hai Jin, and Shoumeng Yan. 2023. Lost along the Way: Understanding and Mitigating Path-Misresolution Threats to Container Isolation. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 3063–3077. doi:10.1145/3576915.3623154

[48] J. Liedtke. 1995. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, USA) *(SOSP '95)*. Association for Computing

Machinery, New York, NY, USA, 237–250. doi:10.1145/224056.224075

[49] R. J. Lipton and L. Snyder. 1977. A Linear Time Algorithm for Deciding Subject Security. *J. ACM* 24, 3 (July 1977), 455–464. doi:10.1145/322017.322025

[50] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. https://dl.acm.org/doi/10.5555/3026877.3026882

[51] Congyu Liu, Sishuai Gong, and Pedro Fonseca. 2023. KIT: Testing OS-Level Virtualization for Functional Interference Bugs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 427–441. doi:10.1145/3575693.3575731

[52] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. Association for Computing Machinery, 461–472. doi:10.1145/2490301.2451167

[53] Rory McCune. 2016. The Dangers of Docker.sock. https://raesene.github.io/blog/2016/03/06/The-Dangers-Of-Docker.sock/ [Accessed 10-09-2025].

[54] R. A. Meyer and L. H. Seawright. 1970. A virtual machine time-sharing system. *IBM Systems Journal* 9, 3 (Sept. 1970), 199–218. doi:10.1147/sj.93.0199

[55] Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph. D. Dissertation. Johns Hopkins University, Baltimore, Maryland, USA.

[56] OscarAkaElvis. 2018. How can I call docker daemon of the host-machine from a container? https://stackoverflow.com/questions/48152736/how-can-i-call-docker-daemon-of-the-host-machine-from-a-container [Accessed 10-09-2025].

[57] Palo Alto Networks. 2024. The State of Cloud-Native Security. https://www.paloaltonetworks.com/apps/pan/public/downloadResource?pagePath=/content/pan/en_US/resources/research/state-of-cloud-native-security-2024 [Accessed 10-09-2025].

[58] Qualys. 2024. The State of Cloud and SaaS Security Report. https://cdn2.qualys.com/docs/mktg/qualys-state-of-cloud-and-saas-security-report.pdf [Accessed 10-09-2025].

[59] Alessandro Randazzo and Ilenia Tinnirello. 2019. Kata Containers: An Emerging Architecture for Enabling MEC Services in Fast and Secure Way. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. doi:10.1109/IOTSMS48152.2019.8939164

[60] RedHat. 2024. Linux Capabilities and Seccomp for Docker. https://docs.redhat.com/en/documentation/red_hat_enterprise_linux_atomic_host/7/html/container_security_guide/linux_capabilities_and_seccomp#linux_capabilities. [Accessed 10-09-2025].

[61] Nick Roessler, Lucas Atayde, Imani Palmer, Derrick McKee, Jai Pandey, Vasileios P. Kemerlis, Mathias Payer, Adam Bates, Jonathan M. Smith, Andre DeHon, and Nathan Dautenhahn. 2021. μSCOPE: A Methodology for Analyzing Least-Privilege Compartmentalization in Large Software Artifacts. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses* (San Sebastian, Spain) *(RAID'21)*. Association for Computing Machinery, New York, NY, USA, 296–311. doi:10.1145/3471621.3471839

[62] J. M. Rushby. 1981. Design and Verification of Secure Systems. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles* (Pacific Grove, California, USA) *(SOSP'81)*. Association for Computing Machinery, New York, NY, USA, 12–21. doi:10.1145/800216.806586

[63] J.H. Saltzer and M.D. Schroeder. 1975. The protection of information in computer systems. *Proc. IEEE* 63, 9 (1975), 1278–1308. doi:10.1109/PROC.1975.9939

[64] Jerome H. Saltzer and M. Frans Kaashoek. 2009. *Principles of Computer System Design: an Introduction* (1st ed.). Morgan Kaufmann.

[65] Pierangela Samarati and Sabrina Capitani de Vimercati. 2001. Access Control: Policies, Models, and Mechanisms. In *Foundations of Security Analysis and Design*, Riccardo Focardi and Roberto Gorrieri (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 137–196.

[66] Vasily A Sartakov, Lluís Vilanova, David Eyers, Takahiro Shinagawa, and Peter Pietzuch. 2022. CAP-VMs:Capability-Based Isolation and Sharing in the Cloud. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*. https://www.usenix.org/conference/osdi22/presentation/sartakov

[67] Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. 2011. seL4 enforces integrity. In *Proceedings of the 2nd International Conference on Interactive Theorem Proving* (Berg en Dal, The Netherlands) *(ITP'11)*. Springer-Verlag, Berlin, Heidelberg, 325–340.

[68] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and Y. C. Tay. 2016. Containers and Virtual Machines at Scale: A Comparative Study. In *Proceedings of the 17th International Middleware Conference* (Trento, Italy) *(Middleware '16)*. Association for Computing Machinery, New York, NY, USA, Article 1, 13 pages. doi:10.1145/2988336.2988337

[69] Rui Shu, Peipei Wang, Sigmund A Gorski III, Benjamin Andow, Adwait Nadkarni, Luke Deshotels, Jason Gionta, William Enck, and Xiaohui Gu. 2016. A Study of Security Isolation Techniques. *ACM Comput. Surv.* 49, 3, Article 50 (Oct. 2016), 37 pages. doi:10.1145/2988545

[70] Ariel Szekely, Adam Belay, Robert Morris, and M. Frans Kaashoek. 2024. Unifying serverless and microservice workloads with SigmaOS. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 385–402. doi:10.1145/3694715.3695947

[71] David W. 2025. Rootless and Standard Docker: A Useful Comparison. https://overcast.blog/rootless-and-standard-docker-a-useful-comparison-6e07e19ab505. [Accessed 10-09-2025].

[72] Nanzi Yang, Wenbo Shen, Jinku Li, Yutian Yang, Kangjie Lu, Jietao Xiao, Tianyu Zhou, Chenggang Qin, Wang Yu, Jianfeng Ma, and Kui Ren. 2021. Demons in the Shared Kernel: Abstract Resource Attacks Against OS-level Virtualization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 764–778. doi:10.1145/3460120.3484744

[73] Yutian Yang, Wenbo Shen, Xun Xie, Kangjie Lu, Mingsen Wang, Tianyu Zhou, Chenggang Qin, Wang Yu, and Kui Ren. 2022. Making Memory Account Accountable: Analyzing and Detecting Memory Missing-account bugs for Container Platforms. In *Proceedings of the 38th Annual Computer Security Applications Conference* (Austin, TX, USA) *(ACSAC '22)*. Association for Computing Machinery, New York, NY, USA, 869–880. doi:10.1145/3564625.3564634