Comparing Isolation Abstractions with *OSmosis*

Sidhartha Agrawal
University of British Columbia
Vancouver, Canada

Shaurya Patel
University of British Columbia
Vancouver, Canada

Arya Stevinson*
University of British Columbia
Vancouver, Canada

Linh Pham*
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

Operating systems provide an abstraction layer between the hardware and higher-level software. Many abstractions, such as threads, processes, containers, and virtual machines, are mechanisms to provide isolation. New application scenarios frequently introduce new isolation mechanisms. Expressing each isolation mechanism as an independent abstraction makes it difficult to reason about the state and resources shared among two tasks running in a given isolation mechanism. This inability to reason about such relationships leads to security vulnerabilities and performance interference.

We present *OSmosis*, an isolation model that expresses a precise level of resource sharing between two executing entities, a framework for tracking model state and comparing isolation mechanisms at runtime, and *CellulOS*, an implementation of the framework on seL4.

The *OSmosis* model lets the user query and compare the isolation guarantees provided by different isolation abstractions. This comparison empowers developers to make informed decisions about isolation and performance trade-offs. *CellulOS* enables developers to easily reason about new and existing isolation abstractions, extract the model state at runtime, and verify that the desired level of isolation was achieved.

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 struggle to provide the right degree of sharing and isolation for a given application and its users [12, 15, 27, 33, 35, 37, 38, 40, 41, 43, 45]. This has become even more apparent with the rise of serverless architectures, where the simple choice between VM and container has become significantly more complicated, and myriad new container/VM hybrids emerge regularly [30, 50, 52, 54, 57, 58]. There is no one-size-fits-all solution, and for a given application, one might want to pick different degrees of isolation/sharing between applications running on the same machine based on security or performance trade-offs.

Even more problematic is that there is no clear understanding of the isolation levels provided by different mechanisms. Perhaps more fundamentally, given an isolation mechanism, it is not immediately clear what comprises an application's state and what parts of that state are shared with, or isolated from, other applications. While some application state is known (*e.g.,* heap, code, data), there exists a significant amount of unknown state that the application is inadvertently sharing with other applications (*e.g.,* system-level services). Worse, we lack a common vocabulary to describe an application's resources, including its known and unknown software state. This lack of vocabulary has led to many problems ranging from performance anomalies due to unintentional sharing, overheads from too much isolation [53], security vulnerabilities caused by unintentional sharing of known and unknown software state [39, 59], and side-channel due to hardware resource sharing [16, 22, 48].

After 60 years, it is somewhat surprising that we have no way to precisely compare isolation mechanisms. We claim that there is a need for a principled way to talk about isolation and sharing and a framework for building implementations.

We present *OSmosis*, which is both a *model* and a *framework*. The *OSmosis* model is a DAG that permits us to reason about isolation and sharing. The DAG has three types of nodes, *protection domains*, which are entities running on a system (*e.g.,* threads, processes, containers, kernel, hypervisor), *resources*, which can be virtual or physical (*e.g.,* an application's virtual memory, its files, the OS state it can query, the processors on which it is allowed to run, and the

Unpublished '24, 2024, Canada
2024. ACM ISBN xxx-x-xxxx-xxxx-x/xx/x
https://doi.org/xx.xxxx/xxxxxxx.xxxxxxx

DRAM pages it can use) , and *resource spaces* that provide context for resources (*e.g.,* virtual address space and `mount` namespaces [42]).

The edges of the DAG precisely describe how nodes interact, *e.g.,* how a protection domain's resources interact with each other and with resources external to the protection domain.

Edges can be of *four* different types: a) a *subset* from a resource to a resource space (*e.g.,* a code region is a subset of the virtual address space), b) a *map* from one resource to another (*e.g.,* a virtual page mapping to a physical page) or one resource space to another c) a *request* that indicates that a PD can request resources from another PD (*e.g.,* a process asks the kernel for more virtual memory), and d) a *hold* that indicates a PD has direct access to a resource.

We introduce two metrics. The ***Resource Similarity Index (RSI)*** quantifies the degree to which two PDs share resources, and thus their level of isolation. The ***Fault Radius (FR)*** describes the distance of two PDs from the first PD on which they both depend (i.e., their least common ancestor in the models' DAGs) for either resource allocations (create a file) or resource access (read a file).

We can compare different isolation mechanisms through the lens of these metrics. Fig. 1 shows two processes running natively on a shared kernel (c) versus two processes, each running inside two separate virtual machines running on a shared hypervisor (d). We consider only the memory resource for ease of exposition, although each process also has other resources, such as files, physical memory, CPU core, caches, sockets, etc. Two processes running natively, each having its own address space, do not share any virtual memory, so their **RSI** for the virtual memory resource is zero. However, those address spaces were allocated using the same operating system—which is their common ancestor; hence, their **FR** is one. Thus, if one process can cause its kernel to crash, that kills the other process as well. In contrast, when the two processes run in their own VMs, their private OS manages each of their virtual address spaces. Thus, these processes can affect the other PD only through their hypervisor, which is two edges away, so **FR** equals two.

The *OSmosis* framework identifies the core operations the operating system needs to provide to support the *OSmosis* model, e.g., tracking the relationship among the different resources and their ownership. We built a prototype system, *CellulOS*[1], that faithfully implements the *OSmosis* framework on the seL4 microkernel [36]. *CellulOS* makes it possible to extract the instantaneous model state on which we run queries to calculate the **RSI** and **FR**.

We evaluate *CellulOS* to show that it is feasible to implement the *OSmosis* framework with acceptable overhead.

[1]Pronounced as cellulose

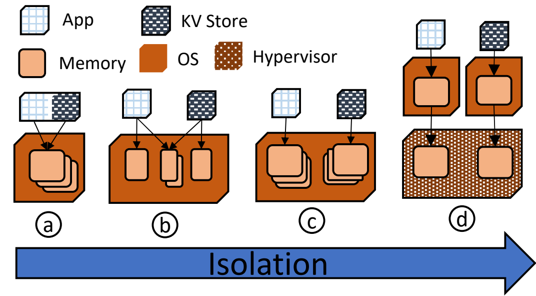


Figure 1: Some options for deploying an application and a KV Store: (a) in the same process, (b) with address-based compartmentalization, (c) separate process, and (d) separate processes inside VMs

For common primitives, *CellulOS* is $1.1\times - 5\times$ slower than our `sel4test` baseline implementation. This is an aggressive baseline, showing *CellulOS* in its worst possible light, as `sel4test` is a test suite and not a practical μ−kernel deployment. We demonstrate that different isolation levels can be achieved using our framework by running an application with a KV store in five different configurations. We then calculate *OSmosis* metrics by computing the metrics on the model state of the running system to demonstrate that comparing isolation abstractions is possible.

Summary. To the best of our knowledge, this is the first attempt to formalize a model for isolation mechanisms and to enable a quantitative comparison of different isolation mechanisms. We make the following contributions:

- We introduce the *OSmosis* model consisting of protection domains, resources, their relationships, and the associated metrics **RSI** and **FR** enabling us to describe isolation precisely (Sec. 3).
- We develop the *OSmosis* framework that identifies the OS mechanisms required to realize this model (Sec. 4).
- We present the *CellulOS* operating system, which implements the *OSmosis* framework and explores what the application of the *OSmosis* model looks like (Sec. 5).
- We show that using the *OSmosis* model we can describe and compare various isolation abstractions (Sec. 3.4.1 & Sec. 6.2).
- We show that *CellulOS* is a faithful implementation of the model and that common operation overheads in *CellulOS* are not prohibitively expensive (Sec. 6.3).

2 ANATOMY OF A USECASE

We use the running example of an application using a KV 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 abstractions.

Fig. 1 demonstrates four different application configurations. (a) The application links with the KVS library in a single process, so `Get` and `Set` operations are simple function calls. (b) The application and KVS are part of the same process, but they use separate address spaces within that process, sharing only a limited number of pages. In this scenario, `Get` & `Set` trigger an address space change. (c) The application and the KVS run in separate processes on the same OS. (d) The application and the KVS run in separate processes, each in a separate virtual machine hosted on a shared hypervisor. It is widely accepted that scenarios (a) to (d) offer increasing isolation between the application and the KVS. However, there is no way to describe this additional isolation precisely.

Comparing the isolation guarantees of the different configurations is challenging for several reasons.

Challenge 1: Multiple implementations of the mechanisms. Often, there exist many different implementations for the same configuration. It is not always possible to determine if each implementation provides exactly the same degree of isolation. For instance, Light-weight context (lwC) [33] and SpaceJMP [17] are two different mechanisms to achieve configuration (b). lwC uses address spaces to create multiple contexts within a process; SpaceJMP uses address spaces to create contexts that are not associated with any process. Both systems can place the KVS in a context separate from the application, however, the resources shared in the two systems are different. For example, lwC isolates file descriptors while SpaceJMP shares file descriptors between contexts. The defaults for the two mechanisms also differ; creating a new lwC is like a fork, and the new lwC has access to all resources of the parent, while SpaceJMP creates *empty* contexts.

Challenge 2: Correct mechanism configuration. A configuration's guarantees can depend on subtle implementation details. For example, while containers might seem to be more isolated than processes, any inadvertent sharing of files with the host or with other containers may actually weaken their isolation guarantees. Indeed, 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 [47, 49], which allows unintended interaction between the containers.

Challenge 3: Lack of hardware isolation. Software abstractions may not actually protect against side channels [13, 16, 21, 22, 26, 34, 46, 48, 60–62] and performance issues [29] that can arise due to hardware resource sharing.

Takeaway. We claim that there is no known method to systematically compare different mechanisms (e.g., processes, containers, and VMs) or even different implementations of similar mechanisms (e.g., lwC and SpaceJMP). Furthermore, we argue that the first step to enabling this comparison is

```
System DAG :- { 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 | Permission
ResourceType :- Virtual(String)|Physical(String)
```

Listing 1: *OSmosis* Isolation Model

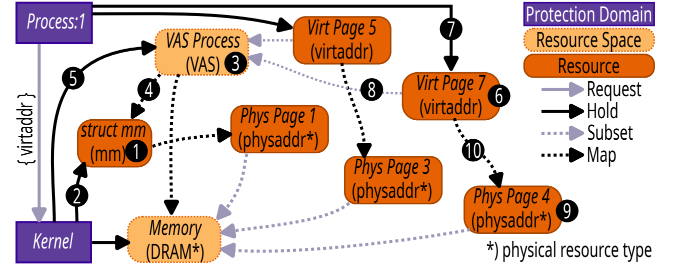


Figure 2: A partial view of system DAG for the process and kernel PDs in *OSmosis* model. Nodes contain their label (*italic*) and type (in parentheses). VAS is for virtual address space

to develop a model to: (1) describe which parts of the application state are shared between two protection domains, (2) describe which hardware resources (*e.g.,* caches, cores, buses, DRAM) are shared between two protection domains, and (3) determine the fault boundary of a protection domain.

3 *OSMOSIS* MODEL

We present the *OSmosis* model in five parts. First, we formalize the notion of resources and relationships. Next, we express common OS operations in the model. Then, we formalize the *Resource Similarity Index* (**RSI**) and *Fault Radius* (**FR**) metrics introduced in Sec. 1. We conclude this section by using the model to analyze several different isolation mechanisms and then discuss some limitations of the model.

3.1 Model

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

Nodes. The DAG has three types of nodes: *(i) Protection domains (PDs)* correspond to active entities of the system (*e.g.,* processes, threads, virtual machines; shown as boxed nodes in Fig. 2). *(ii) Resource spaces* provide context for resources (*e.g.,* virtual address space, file system, `mount` namespace [42]; shown as patterned rounded boxes in Fig. 2). *(iii) Resources* are passive entities and can be either physical (*e.g.,* DRAM, disks) or virtual (*e.g.,* virtual page, file,

socket); these are solid rounded nodes in Fig. 2. Both resources and resource spaces are typed. Resource nodes are non-overlapping.

Edges. There are four kinds of directed edges, and each edge can have a type-specific set of attributes (e.g., the permission over a held or mapped resource or the type of resources that can be requested). *(i) Hold-edges* connect a PD to a resource or resource space, indicating that a PD currently holds certain rights over the resource (e.g., a process can access a virtual page or open a file); these are shown with solid black arrows in Fig. 2). *(ii) Request-edges* connect two PDs, signifying that the source PD can request resources of a specified type from the destination PD (e.g., a process invokes the kernel's `mmap()` system call to obtain more virtual memory); these are shown with sold gray arrows in Fig. 2). *(iii) Subset-edges* associate a resource with its originating resource space. The attributes of an edge identify the portion of the resource space comprising the resource (e.g., the specific virtual page that is a subset of a virtual address space; these are shown with dotted gray arrows in Fig. 2). *(iv) Map-edges* connect resource and resource space nodes (shown with dotted black arrows in Fig. 2). Map edges can connect different node types. A map-edge between: 1) two resource nodes 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., a virtual page maps to a physical page), 2) a resource space and a resource expresses a metadata dependency (e.g., the `struct mm` contains metadata that tracks a process's virtual address space), and 3) two resource spaces indicates that subsets of the source space are permitted to map to subsets of the destination space.

Resource Types. Each resource and resource space has a type. The resource type indicates whether the resource is virtual or physical. All resources with a subset edge to the same resource space have the same type, i.e., in Fig. 2 all Virt Pages that subset from "VAS process" have type `virtaddr`.

3.2 Expressing OS operations in *OSmosis*

We illustrate how common OS operations modify the System DAG through the addition and deletion of nodes and edges.

Creating Resource Space. Protection domains can create new virtual resource spaces at will. First, the PD allocates a bookkeeping data structure. In the example that follows, all circled numbers correspond to Fig. 2. The Linux kernel creates a new virtual address space by allocating a new `struct mm`. In *OSmosis*, this corresponds to making `struct mm` (1) a new virtual resource to which the kernel has a hold-edge (2). This creates a new resource space node with type virtual address space (VAS) (3) and two new edges: a *map* edge from the VAS node to the `struct mm` resource node (4) and a *hold* edge from the kernel PD to the VAS node (5).

Requesting Resources (`mmap`). Protection domains request resources of type X from the protection domain for which they have a request edge with a type attribute X. Continuing the example above, the user process has a request edge to the kernel protection domain with a type attribute corresponding to the process' virtual pages (i.e., the gray edge with attribute virtaddr). This enables the process to call `mmap()` to request virtual pages from the kernel. When the kernel allocates virtual page 7 to the process, this is reflected in the graph by creating a resource node (6) and two new edges – a *hold* edge from the process to the resource (7) and a *subset* edge from the resource to the resource space (8) (i.e., the VAS node).

Creating Mappings (pagefault). The `mmap` operation described above does not eagerly allocate physical memory. The process dereferencing the newly allocated virtual page raises an exception. In response, the kernel allocates a physical page and creates a mapping, which is reflected in the graph by a new resource node (9) and a new map edge from the virtaddr node to the physical page (10). The graph now reflects that virtual page 7 is mapped to a physical page 4.

3.3 Isolation Metrics

We now introduce two metrics that quantify the isolation between a pair of PDs. Two PDs can affect each other in two ways: (i) through a shared resource, and (ii) through a common ancestor PD. We propose two metrics—the *Resource Similarity Index* (**RSI**) and the *Fault Radius* (**FR**)—that quantify the two effects, respectively.

3.3.1 *Resource Similarity Index (RSI).*

The Resource Similarity Index (**RSI**) quantifies the relative similarity of two PDs in terms of resources available to them, either by holding them directly or through mappings. Intuitively, the **RSI** quantifies the degree of resource sharing between two PDs. The **RSI** is a vector with one element per resource type in the union of the resource types available to the two PDs. Each element of the vector is a value between zero and one that indicates the fraction of the resources of a given type shared between the two PDs. A value of zero means that the two PDs currently share no resources, and a value of one means that they share all resources of the given type.

RSI Calculation. We calculate the **RSI** in two steps: 1) For each of the two PDs, we execute a breadth-first traversal along the hold and map edges of the System DAG, collect all reachable resource nodes, and group them by their type. 2) For each resource type, we calculate the Jaccard Similarity Coefficient, i.e., the ratio of the intersection of the resource nodes to the union. If a resource type is present in only one of the PDs, we set the value to zero.

Comparing RSIs for two mechanisms. Consider a pair of application components that could be implemented as two PDs using different isolation mechanisms. Given the **RSI** metric, we can compare the **RSI** vectors for different mechanisms to determine how they differ.

Continuing with our running example, suppose the two PDs for the KVS and the application are created using two isolation mechanisms $\mathcal{M}_1$ and $\mathcal{M}_2$, such that **RSI** vectors for the PD pairs under the two mechanisms are $\mathcal{R}_1$ and $\mathcal{R}_2$, respectively. We say that the PD pairs are equally or more isolated under mechanism $\mathcal{M}_1$ than under $\mathcal{M}_2$ iff $\mathcal{R}_1 \preceq \mathcal{R}_2$. Furthermore, $\mathcal{R}_1 \preceq \mathcal{R}_2$ iff $\forall r \in \mathcal{R}_1, \exists r' \in \mathcal{R}_2 \mid \texttt{TypeOf}(r) = \texttt{TypeOf}(r') \wedge r \leq r'$ and $|\mathcal{R}_1| = |\mathcal{R}_2|$, which implies that each resource type is more shared under $\mathcal{M}_2$ than under $\mathcal{M}_1$.

Even when the **RSI** vectors of two mechanisms are incomparable (i.e., $\mathcal{R}_1 \npreceq \mathcal{R}_2$), one could choose an isolation mechanism based on any combination of the level of isolation required and the level of isolation tolerated on specific sets of resources.

3.3.2 Fault Radius (FR). The Fault Radius (**FR**) quantifies the distance to the nearest common ancestor between two PDs, where $ancestor(PD1, PD2)$ means that PD1 depends on PD2. PDs depend on each other in two ways: explicitly via request-edges and implicitly via its virtual resource mappings. For example, in Fig. 2, Process depends on Kernel explicitly via its request-edge of type virtaddr, but also depends on Kernel implicitly for virtual memory (virtaddr) mappings: the process holds "virtual page 7" that has a subset-edge to VAS, for which there exists a hold edge from the Kernel – i.e., the kernel can change the map edges of the virtaddr resources held by the process. The algorithms for detecting implicit dependencies and computing **FR** appear in Appendix A.

The **FR** is an integer value between one and ∞, where ∞ means that the two PDs do not have a common ancestor, and a value, n, means that the least common ancestor is at least n edges away from both PDs. Intuitively, **FR** provides a measure of how hard it is for one PD to affect another PD by causing a fault in a common ancestor PD (*e.g.,* if an application PD crashes the kernel PD, then all other PDs running on the kernel crash).

Comparing FR for two mechanisms. The **FR** is a direct measure of isolation. Assuming that the **FR** metric for two PDs created using mechanisms $\mathcal{M}_1$ and $\mathcal{M}_2$ is $\mathcal{F}_1$ and $\mathcal{F}_2$, respectively, we say that the PD pairs are equally or more isolated under $\mathcal{M}_1$ than under $\mathcal{M}_2$ iff $\mathcal{F}_1 \geq \mathcal{F}_2$.

3.4 Using the *OSmosis* Model

We now show how the *OSmosis* model enables us to compare various isolation abstractions from the literature using the metrics and to determine which resources and PDs should be reclaimed after a fault occurs in a given PD.

3.4.1 Comparing Isolation Abstractions. Table 1 shows the results of our comparison of nine deployment scenarios of the KVS application from Sec. 2. This table was generated manually, before we conducted the evaluation that appears in Sec. 6, based on our understanding of these abstractions.

For each deployment scenario (columns in Table 1), we construct the model and evaluate queries to compute **RSI** and **FR**. We group the scenarios into mechanisms that are in-process (1-3), between processes (4-6), and between virtual machines (7-9). We discuss our analysis for each group, highlighting the differences in the deployments.

Threads, SpaceJMP & lwC. In *OSmosis*, we model threads as separate protection domains. Two threads share both the entire virtual address space and the same set of physical pages, so the **RSI** values for both resources are one. In contrast, SpaceJMP and lwC create separate address spaces, so their virtaddr **RSI** is zero. However, SpaceJMP and lwC can share access to physical pages, producing an **RSI** value greater than zero but less than one (i.e., V). We expect this value to be different for SpaceJMP and lwC, because lwC does not share the heap and stack, while SpaceJMP does. Focusing only on virtual and physical page resources, the **RSI** clearly illustrates how the mechanisms differ in their sharing behavior. Similarly, the **RSI** captures how lwC isolates file descriptors while threads and SpaceJMP do not.

Processes and their variants. Two processes each have their own virtual address space, hence the **RSI** for virtaddr is zero. However, they can share some physical pages or open files, indicated by V in the table. The **FR** value is again one for all these scenarios as they all request resources from the kernel. The three scenarios differ, however, in their file system configuration. Each process has its own file descriptor table (**RSI** is zero). Interestingly, we see that adding separate file system namespaces (NS_Child) *does not* affect the **FR**, as in both cases the same kernel manages the file system. When processes are in separate namespaces, some parts of the file system may be either hidden or shared between them, hence the **RSI** for file objects is V. This value alerts us to the sharing of files that can happen in all implementations of containers. In contrast to namespaces, two independent file systems implemented by two separate file servers produce a file object **RSI** of zero, as file sharing is impossible. Thus we distinguish the effect of namespaces by examining the **RSI** values for file paths and file object resource types.

Virtual Machines and their variants. Unsurprisingly, all virtual-machine scenarios show a virtaddr **RSI** of zero. However, **FR** increases to two, as the guest kernels are independent, and the hypervisor is the least common ancestor. Without strict host-physical memory isolation, VMs can share physical memory (e.g., same-page merging [32, 56]), producing an **RSI** of V. With strict physical memory partitioning, as used in Cellular Disco [20] or core slicing [63],

Resource Similarity Index (RSI)	Threads (1)	SpaceJMP [17] (2)	IwC [33] (3)	Processes (4)	Processes in Mount Namespace (5)	Processes in different Filesystem (6)	Processes in separate VM (7)	Processes in separate VM with static Phys. Mem (8)	Processes in separate VM with LLC aware static Phys. Mem (9)
virtaddr	1	0	0	0	0	0	0	0	0
Host PhysPages	1	V	V	V	V	V	V	0	0
LLC Cache Sets	V	V	V	V	V	V	V	V	0
Open File Table	1	1	0	0	0	0	0	0	0
File Path (NS_Child)	NA	NA	NA	NA	0	NA	NA	NA	NA
File Path (NS_Global)	1	1	1	1	V	0	0	0	0
File Object	1	1	1	1	V	0	0	0	0
Fault Radius (FR)	1	1	1	1	1	1	2	2	2

Table 1: RSI and FR metrics for nine deployment scenarios. V stands for a value between zero and one. NA means that resource type is not applicable in this scenario.

the **RSI** is zero, because the VMs no longer share physical pages. Cache-conscious memory partitioning can be used to limit VMs to distinct cache sets, producing an **RSI** of zero for LLC cache set resource. Interestingly, adding hypercalls [10], introduces a request edge from the process to the hypervisor, reducing **FR** to one.

Summary. Our analysis shows that many deployment scenarios exhibit subtly different behavior that is nicely captured by our model and the **RSI** and **FR** metrics. Moreover, we show that namespaces do not provide the level of isolation one might expect and that certain optimizations (e.g., hypercalls) poke holes into the isolation boundaries, potentially making such systems more prone to a single point of failure.

We believe that **FR** metric can also be calculated for specific resource types, and can be used to find the common ancestor PD providing a particular resource type. This might be helpful when different resources are considered differently vulnerable to attack, and one might wish for isolation mechanisms that isolate different resources to differing degrees.

3.4.2 Resource Cleanup. A second benefit of the *OSmosis* model is that we can now precisely identify which protection domains and resources need to be reclaimed after a crash. A simple, but perhaps overly conservative, rule is to kill all PDs for which there exists a request edge to the failing PD. Then, reclaim all resources held by the terminated PD. Finally, recursively clean up all protection domains that also held those reclaimed resources. We can adjust the degree of conservatism by using the **RSI** and **FR** metrics to set thresholds that determine when a PD should be terminated. Intuitively, if the level of sharing is high (**RSI** is close to one), we should assume that most of the resources of the PD may have been corrupted. Finally, we can use the **FR** metric between the failing PD and another PD to determine whether we should clean up the other PD as well. For example, if we clean up everything up to an **FR** value of one, this means we would clean up only those processes running on the same kernel, but not those running on a kernel in a separate VM.

3.5 Discussion

The *OSmosis* model and its metrics provide the foundation to talk about isolation in a computer system in a principled way. Two important aspects we currently do not capture are the notion of resource allocation policies, such as quotas, and specific protection domain policies that might lead to better isolation, e.g., if an allocator correctly allocates only free pages, then certain kinds of sharing will never happen, and we can more precisely constrain **RSI**. However, we can express certain policies through static partitioning of resources into protection domains and then carefully use request edges to express the policy. For example, we could deploy two independent memory servers and then give PDs a request-edge to only one of them.

4 *OSMOSIS* FRAMEWORK

The *OSmosis* framework describes a set of core operations required to maintain and extract the model state. Existing kernel data structures already contain most of the required information from which to construct the *OSmosis* model. The *OSmosis* framework describes functionality to track the creation or deletion of nodes and edges in the *OSmosis* model (Listing 1) from existing operating system data structures where possible.

4.1 Protection Domains

Protection domains correspond to entities in the system (*e.g.,* threads, processes, virtual machines etc.). Operating systems already track them: for example, Linux uses `struct task_struct`, seL4 has thread control blocks (TCB), and hypervisors use virtual machine control structures.

Creating a PD. Operating systems typically create new entities by allocating some form of control structure. In *OSmosis*, this corresponds to the creation of a new, empty PD node with no edges. In some cases, creating a new domain might also involve allocating additional data structures such as the open file table or the capability table.

Adding and removing resources and request edges. Before a new PD can execute, it needs resources that can be given explicitly or inherited from its parent (*e.g.,* PDs spawned through `fork()` or `clone()` system calls inherit resources). Operating systems already maintain data structures that track which entities have access to which resources – an important aspect when cleaning up entities after they exist or crash. PDs have hold-edges to resources that they can access. For example, Linux keeps track of virtual memory areas of a process using `struct vma_region`. Thus, when adding a new virtual memory area to a process, this corresponds to adding a hold-edge from the PD to the resource. Inversely, deleting a `vma_region` removes the hold-edge. On Linux, a process has, by default, the ability to perform all systems calls (e.g., `mmap` or `open`). This is reflected as a request-edge from the PD to the kernel. In a capability-based system such as seL4, these operations closely correspond to capability operations (e.g., endpoint invocations).

Destroying a PD. Before a PD can be destroyed, all its resources must be freed or returned to the managing PD. We can identify all such resources by following the hold-edges. If the exiting PD is the only one holding the resource, we can follow the subset-edge of the resource to return to the resource space. In Linux, this is implicit; when a process terminates, the kernel that tracks all the resources associated with a process, reclaims them.

4.2 Resources

The *OSmosis* framework requires that we track the creation, and deletion, of resources and their relationship to PDs and other resources. Operating systems already keep track of resources; the *OSmosis* framework merely makes this state explicit and requires that a system provide ways to access it.

Resource state. Each resource has a type (e.g., physaddr, virtaddr), a unique ID, and metadata required for executing *OSmosis* queries. For example, a physical page should be tagged with a Host PhysPage address. A virtaddr (vma_region in Linux) will be tagged with its starting and ending addresses.

Physical resources. All systems start with an initial set of physical resources (e.g., CPUs, physical memory, caches). The framework piggybacks on the regular system startup code that finds and initializes these physical resources. It assigns appropriate resource state to the physical resources and creates new nodes for these resources (such as the DRAM resource space shown in Fig. 2). It also creates *hold-edges* from the PD managing physical resources to the resources themselves.

Virtual resources. PDs create virtual resources and virtual resource spaces. They can create virtual resource spaces by allocating data structures representing the resource (*e.g.,* an operating system creates a PD/virtual address space for a new process) or they can issue a request to the PD for which they have a request-edge for the resource type (*e.g.,* a user process requests a new virtaddr from the operating system PD). After creation, the framework creates a *subset-edge* between the resource space and the resource and a *hold-edge* between the resource and the creator PD. Then, it copies the resource to the new PD, creating a new *hold-edge*, *e.g.,* a new vma_region is added to the `mm struct` in Linux when `mmap` is called.

Accounting for Indirect Resources Indirect resources are resources that cannot be assigned/requested directly but are controlled by allocating other resources. For instance, cache sets cannot be directly allocated (ignoring Intel CAT [2]). However, they can be allocated by controlling which physical addresses are allocated to a process. When a physical page is allocated, the framework can calculate all potential cache sets accessible to the PD receiving the physical page (by using the topology service described below) and add a map-edge from a physical page to a cache set resource.

Topology service Some edges are inherent to system topology and not stored in PDs (*e.g.,* physical address to LLC cache set mapping). For such edges, we assume the existence of a *system topology service* that can be queried akin to the `sysfs`, `dev`, and `proc` systems in Linux.

4.3 Hyrbid Nested PDs

As demonstrated by the various scenarios, PDs can nest (*e.g.,* PDs running inside a VM are nested inside the VM PD from the perspective of the hypervisor). However, until now, our discussion assumes that all the PDs adhere to the *OSmosis* model. Consider a scenario in which a hypervisor adheres to the *OSmosis* model, but its VMs run conventional operating systems that do not. In such cases, the *OSmosis* framework might not know about the nested PDs. Nonetheless, the hypervisor or VM might want to incorporate the state of nested PDs into the *OSmosis* model. In this scenario, the simplest approach is for the hypervisor and virtual machines to track resources only to the level of the guest operating

systems and accept that what happens inside those operating systems is opaque. A slightly more intrusive approach is to implement a set of framework functions that enable the guest operating systems to export model state to the VMs in which they run. In general, any piece of software *can* produce *OSmosis* state, but no software is required to do so. Failure to do so simply limits the granularity at which *OSmosis* can respond to state queries.

5 IMPLEMENTATION

We present *CellulOS*, a seL4-based, user-space server that implements the *OSmosis* framework. We used seL4 as the basis for the *CellulOS* prototype for two main reasons. SeL4's capabilities are a natural fit to delegate and track resources across different PDs. As a microkernel, seL4 provides only low-level abstractions (page table manipulations), allowing us to design higher-level abstractions (*e.g.,* process or VM creation). *CellulOS* is one realization of *OSmosis*, but it is not the only possible one. We first introduce the key seL4 features that act as building blocks for *CellulOS*, and then we show how we use them to implement the *OSmosis* framework.

5.1 seL4 Primer

SeL4 [28] is a formally verified microkernel from the L4 family [24, 31]. We focus on seL4 capabilities, protection domains, and IPC as these are the building blocks upon which we implement *CellulOS*.

Capabilities. seL4 uses a capability system for access control to system resources such as memory frames and thread control blocks (TCB). Applications invoke capabilities to request services, e.g., to map a region of memory in their address space. The capabilities serve as unforgeable tokens that give applications the right to invoke specific methods on system resources. The capabilities of an application live in its *capability space (CSpace)* that is directly accessible only by the kernel.

Protection domains. Applications run in *protection domains (PD)* each of which has an associated CSpace that contains all capabilities (and resources) held by a PD. Thus, access rights to resources must be explicitly given to applications by transferring capabilities to the application's CSpace or by sending capabilities via IPC.

Inter-process communication (IPC). Threads in seL4 communicate using blocking, synchronous IPC using kernel objects called endpoints. Messages and capabilities are sent and received by invoking endpoints.

Capability transfer: To transfer a capability, the receiver specifies the location in its CSpace where the capability should be placed when calling the (blocking) receive system call. The sender passes the capability when invoking the send system call. The kernel then transfers the capability to the specified location in the receiver's CSpace.

Badged endpoints An endpoint can be *badged*, i.e., have a 64-bit value attached to it. Applications choose badge values by *minting* an unbadged endpoint, which creates a new endpoint with an immutable badge attached to it. IPC transmits badge values with messages. A typical use of badged endpoints is a server that listens on an endpoint, minting a new, badged endpoint for each client. The server then uses the badge value to identify the client that sent a message.

Bootstrapping the Root Server A minimal seL4 system consists of the seL4 μ-kernel and a *root-server*. The root-server is the first userspace PD started by the seL4 kernel. During boot, the kernel creates capabilities for all system resources and places them in the CSpace of the root-server. The location of these capabilities and other information (e.g., the number of cores) is passed to the root-server via the `BootInfo struct`.

5.2 *CellulOS* Architecture

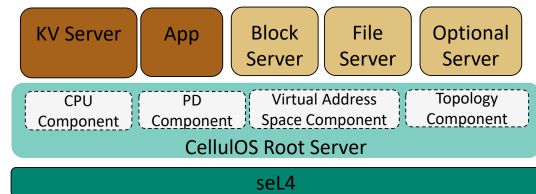


Figure 3: *CellulOS* Architecture. The *CellulOS* root-server is composed of several components that manage core resources. Optional Resource servers (such as File Servers) also manage resources and must register themselves with the root-server

We implement *CellulOS* in the root-server as shown in Fig. 3. The root-server manages *OSmosis* protection domains and resources as required by *OSmosis* framework. It also manages the system hardware topology (e.g., the memory hierarchy and processor cores with their caches). During system startup, the root-server initializes the allocators for CSpace and physical memory based on the information obtained from the `BootInfo struct` and the subsystem components and system services. Optionally, the root-server can create new PDs running *resource servers* (*e.g.,* file server, block server).

5.2.1 *CellulOS capabilities.*

CellulOS keeps track of resources and request-edges using badged-endpoint capabilities. We assign badge values as follows: *<unused:16, resID:16, resServerPD:4, holdPD:4, perm:16, resType:8>* The badge representation limits the total number of resources and PDs that

CellulOS can track. We leave the scalability of that as future work.

5.2.2 ***Managing PDs.*** The root-server's PD component is responsible for creating and managing protection domains. For each PD, the root-server maintains a descriptor containing its ID, capabilities for the set of resources it holds, the request-edges it may use and one to the PD's CSpace (to provide future capabilities). PD creation starts with allocating a new, empty seL4 CSpace. The root-server then populates the CSpace with an initial set of resources and requests endpoints, both of which are badged seL4 endpoint capabilities.

When a PD is destroyed the root-server first returns all the resources from the PD back to the PDs referenced by request-edges and then destroys the CSpace and the PD.

5.2.3 ***Starting Optional Resource Servers PDs.*** Starting an optional resource server PD, such as a file server, is the same as creating any other PD with a few additional steps. After starting up, this new PD first registers with the root-server to indicate that it will be providing a new type of virtual resource. The root-server then creates its own unbadged endpoint on which the new resource server will listen. The root-server needs the unbadged endpoint so that it can properly badge endpoints when it transfers a virtual resource from the resource server to some other PD. Future requests to access the virtual resource (*e.g.,* read from the file), will not go through the root-server.

5.2.4 ***Creating Virtual Resources.*** The root-server creates virtual resources for vCPUs and virtual address spaces. Additional virtual resources can be created by other resource server PDs. For example, a file server creates file virtual resources. Fig. 3 shows a scenario with a file resource server backed by a block resource server running as separate PDs.

When a PD requests a new resource, it looks up the corresponding request edge (badged endpoint capability) in its CSpace and invokes the endpoint. The endpoint is badged with information identifying the requesting PD. When the server receives the request, it can identify the PD requesting the resource and transfer the newly allocated virtual resource to the requester via the root-server.

5.2.5 ***Transfer of Virtual Resources.*** Server PDs need to inform the root-server of all resources they transfer so that the root-server can keep track of all allocated resources in the system to extract the *OSmosis* model. Server PDs enable this by sending newly created badge information to the root-server. The root-server then badges the server PD endpoint with information about the new resource and then transfers the resource (the badged endpoint) to the receiver by copying it into the receiver PD's CSpace.

```
req = getCurrentRequestEdges();                    1
vas = newVAS();                                    2
// Get code, stack, heap, and vCPU resources       3
code, heap, stack = load(vas, "binary");           4
vcpu = newvCPU();                                  5
resources = {code, heap, stack, cpu};              6
// Create the PD                                   7
pdID = newPD(resources, req);                      8
```

Listing 2: Process creation in *CellulOS*

5.3 Extracting the System DAG

The root-server extracts the System DAG by combining information from all PDs. As described in Sec. 3.1, the DAG has three types of nodes: PDs, resource spaces, and resources. and four types of edges: map, hold, subset, and request. The root-server iterates over all the PDs and lists out their resources as nodes. It creates hold-edges from the PD nodes to their respective resource nodes. It also connects PDs to their respective destination PDs using request-edges. For resources of which the root-server is aware, it creates subset and map-edges to connect resource and resource space nodes. The root-server receives nodes and edges for resources managed by optional server PDs by querying those PDs. The root-server connects those nodes to the DAG it already has built. *CellulOS* dumps this information into a CSV file that we load into a Neo4j graph database [11]. We then calculate the metrics described in Sec. 3.4.1 and discuss the results for use cases we built in Sec. 6.2.

5.4 Unified API to Build Abstractions

CellulOS provide a single API to build different isolation abstractions. *newPD*, that can create any type of PD (inspired by the `posix_spawn` in Linux [44]), given a set of *resources* and *request-edges*. Listing 2 shows the pseudo code to create a standard process in *CellulOS*. Since a process runs in its own address space, we first create a new virtual address space resource. The `load` call reads a binary file and assigns an inital set of resources and request-edges. `load` allocates from the `VAS` resource space to create `code`, `heap`, and `stack` resources. It assigns the same request-edges to the new PD as the creator PD.

There are instances where the new PD is a close replica of an existing PD, and the pseudo code in Listing 2 can be cumbersome. Taking inspiration from `clone` [41], *clonePD* creates a new PD by calling an *isolation function* that defines how each resource and request edge is shared with its creator PD (or another PD) before calling *newPD*. It is not fundamental to our model or framework, but it is syntactic sugar that makes *CellulOS* easier to use. We have written a few such *isolation functions*, for example, one that creates a process, one creates the PD with a slightly different address

space (*e.g.*, something that shares code and data segments but not the heap), one that starts the new process in a different mount namespace, and one that starts the new process along with a new file server. We evaluate this API in Sec. 6.1

6 *CELLULOS* EVALUATION

The goal of our evaluation is to examine both the utility and feasibility of implementing a system that natively supports the *OSmosis* model. We address the following research questions:

- Does the *OSmosis* model facilitate easy implementation of isolation scenarios?
- Does *CellulOS* capture information at runtime that usefully lets us compare isolation primitives?
- How much overhead does *OSmosis* model tracking introduce?

Test Setup: We run *CellulOS* on the Odroid-C4 [23] board with a 64-bit quad-core ARMv8 Cortex-A55 cluster and 4 GB of RAM.

6.1 Building Isolation Abstractions

We present a small case study showing how to use *CellulOS* to easily construct five isolation abstractions, using the API described in Sec. 5.4: threads, HighJMP[2] , process, process with mount namespace, and process with different file systems servers. In addition to the KVS server and client PDs, all of the scenarios include a ramdisk PD and a file system PD. The file system PD has a request edge to request blocks from the ramdisk PD.

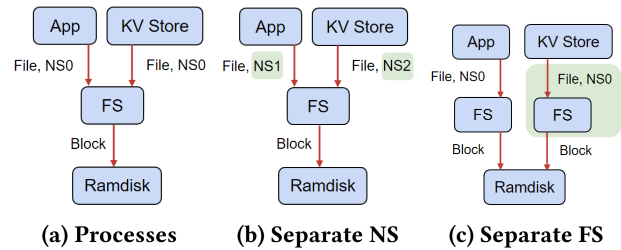
We create a thread using a `load` call to set up its virtual address space. This thread also shares the original thread PD's request edges for virtual/physical pages and files. We then create a second thread, using `clonePD` with an isolation function such that it shares all resources with the previous thread except CPU, so that it can be scheduled independently, and shares all the request edges. The isolation function also creates a new stack for the second thread by allocating new virtual and physical pages.

We set up HighJMP with a different isolation function from the thread scenario. This function creates a new virtual address space for the second PD and mirrors the virtual pages for code, data, stack, heap, and IPC buffer. The virtual pages that comprise the code, data, stack, and IPC buffer have map edges to the same physical pages as the first PD. In contrast, the virtual pages that comprise the heap's virtual memory region map to newly allocated physical pages.

[2]In HighJMP (inspired from SpaceJMP), the PDs have two separate virtual address spaces that share the code, stack, and data pages at the same virtual address in both virtual address spaces. However, they have private pages for their heaps. This differs from SpaceJMP in which an address space is a standalone entity to which any process can attach, at which point, the process' code, data and stack are added to the address space.

Resource Type	Threads	HighJMP	Processes	Processes in Mount NS	Processes w different FS
virtaddr	1	0	0	0	0
PhysPage	1	0.78	0	0	0
File Object	1	1	1	0	0
Fault Radius	1	1	1	1	1

Table 2: RSI and FR metrics for implemented PDs.



(a) Processes **(b) Separate NS** **(c) Separate FS**

Figure 4: Processes & variants

Unlike threads and HighJMP, processes share no resources, and variants are implemented by modifying the file request edges of the KVS server and client. We can create processes as clones of the PD calling *clonePD*, where the isolation function chooses which file request edge they receive. In Fig. 4, we show that the PDs for two standard processes have request edges to request files from the same file system. For mount namespaces, we first request two new namespaces from the file system and initialize the request edges of the PDs to different namespaces. When the file server receives requests, it uses the namespace specified in the caller's request edge. Finally, to set up PDs in different file systems, we clone the first file system PD and set the KVS server and client's request edges to request files from separate file system PDs.

With the building blocks of the *CellulOS* framework, we were able to implement each of these scenarios with a few lines of code.

6.2 Comparing Isolation Abstractions

We evaluate the efficacy of using the *OSmosis* model to differentiate the five isolation mechanisms described in the previous section. At runtime, we extract the model state, import it into Neo4j, and then query the state to calculate the **RSI**. For expediency, we computed **FR** using a Python script.

The results from our implementation, shown in Table 2, align with the predictions made in Table 1. We did, in fact,

create Table 1 manually, before we had a working implementation. As expected, the **RSI** for virtaddr is one for threads and 0 for others. and the **RSI** for file objects is zero when the PDs use disjoint namespaces or separate file servers. It is possible for the processes running in different namespaces to have non-zero **RSI** value for file objects if they share files at the host level, however, in our scenario, the namespaces are not set up to share any underlying files. Note that our implementation does not yet include the file name resource, so we evaluate the sharing of file objects only. The **FR** is one in all our scenarios, as the PDs have a request edge to the root-server.

The only difference from Table 1 is that SpaceJMP's **RSI** for physical pages was **V**, whereas the **RSI** for physical pages in Table 2 for HighJMP is 0.78. This is expected as at runtime, we get an exact value based on the number of pages shared at that instant. By comparing the **RSI** vectors for threads and HighJMP (as explained Sec. 3.3.1) we can say that HighJMP is more isolated as compared to threads.

OSmosis model visualizations also provided helpful insights at debug time. When we first imported the HighJMP scenario's model state into Neo4j, it uncovered a bug where the PDs had independent physical pages for their code and data sections, instead of shared pages as we had intended. In this case, the **RSI** metric was 0.46, expressing the lesser degree of sharing. Similarly, when debugging the failure of HighJMP's address space switch, the model state exposed the culprit: the PDs were not sharing the same page for their IPC buffer. The system call to switch address spaces used the IPC buffer and failed when its contents were not as expected.

6.3 Overhead of Maintaining Model State

The primary way in which *CellulOS* differs from other operating systems is that it implements the *OSmosis* framework in a central root-server to track the *OSmosis* model. Hence, we measure the overhead added by the *OSmosis* framework.

We run the OS microbenchmarks, shown in Table 3, and compare *CellulOS* to the baseline of the sel4test suite [6], which contains various utility libraries for user-space virtual memory management, process creation, and memory allocators. `sel4test` is a particularly aggressive baseline as it is not meant to be a realistic μKernel deployment. It does not run system services such as file servers or block servers in separate PDs. All baseline operations were run in a regular, user-level process, where resources are managed by the process itself, using allocation libraries included in `sel4test`.

We run *CellulOS* operations similarly, in a user-level process, however, with resources managed by the *CellulOS* root-server and optional resource servers (for files and blocks).

In both the baseline and *CellulOS* tests, we first create the root-server (i.e., an initial privileged process) and one user-level process.

We then run each operation, one after another, in the order listed in Table 3. We reboot the system between trials. We use the ARMv8 Performance Monitor register (`PMCCNTR_EL0`), which tracks processor CPU cycles, to produce latency measurements.

The overheads reported in Table 3 reflect the additional work that must be done in *CellulOS* for each listed operation, relative to the native seL4 process. This additional work includes an IPC round-trip to contact the root-server and the root-server's model state tracking and endpoint badging for resource creation (see Sec. 5.2.4).

As shown in Table 3, we took intermediate measurements at each stage of *CellulOS* operations to determine how much each overhead type contributes to the reported overhead. `IPC Overhead` corresponds to the CPU cycles (in thousands) that are attributable to IPC round-trips. Similarly, `Endpoint Badging Overhead` reports CPU cycles (in thousands) that are attributable to endpoint badging during resource creation. Overhead attributed to model state tracking varies based on the operations. We then use these measurements to estimate the baseline latency for creating a file as a virtual resource. As we did not have a baseline filesystem implementation, the baseline latency reported in Table 3 for file creation is the *CellulOS* latency, excluding CPU cycles contributed by IPC round-trips and endpoint badging. It, however, does not exclude overhead contributed to by *OSmosis* model state bookkeeping, as this can vary depending on the state of the resource server.

CellulOS primitives are $1.1 - 5\times$ slower than the baseline. There are two categories of overhead observed in *CellulOS* (a) constant, and irreducible, and (b) variable and reducible.

CellulOS's highest overhead operations are the simplest ones because there is so little work done in the base case. The overhead stems from **additional** IPC calls and endpoint badging operations (that fall under category (a)) required in *CellulOS*. *CellulOS* processes can obtain resources only by communicating with servers via IPC while native seL4 processes do not require IPC. For instance, Table 3 shows that IPC overhead accounts for nearly all of *CellulOS*'s `Switch Address Space` overhead; this operation requires no endpoint badging. Similarly, during `Transfer Resource`, IPC and endpoint badging combined contribute to $18k$ cycles out of the total $22k$ cycles overhead.

Heavyweight operations, such as process creation, setup, and spawning (`Process Spawn`), exhibit lower relative overhead. *CellulOS* adds 5 IPCs and 3 badging operations (about 100k cycles) in addition to baseline `process spawn` (2897k cycles) leading to a lower overhead. The majority of overhead for these operations falls under category (b). We prioritized

Operation	Baseline (seL4test)	*CellulOS*	IPC Overhead	Endpoint Badging Overhead	Total Overhead
Create PD	60.90 ± 1.96	112.53 ± 1.48	14.77	5.36	51.63
Transfer Resource	4.23 ± 0.14	26.31 ± 0.59	12.13	5.36	22.08
Create Address Space	37.86 ± 1.35	94.17 ± 1.57	14.77	5.36	56.31
Map Memory	15.17 ± 0.58	31.78 ± 0.68	12.13	*N/A*	16.61
Create CPU (TCB)	12.22 ± 0.37	42.48 ± 1.30	14.77	5.36	30.26
Spawn Process	2897.50 ± 19.47	4218.40 ± 4.04	73.86	16.09	1320.94
Switch Address Space	4.74 ± 0.32	23.33 ± 0.64	14.77	*N/A*	18.59
Create Virtual Resource (File)	374.21 ± 2.68	424.48 ± 2.68	14.77	5.36	50.27

Table 3: Average latency and overheads (in thousands of CPU cycles) and for common OS operations, over 60 trials

simpler data structures for *OSmosis* model state tracking rather than efficient ones, so many of *CellulOS*'s bookkeeping requires slow traversals over linked lists. There is plenty of room to optimize these bookkeeping mechanisms. It is worth noting that `Spawn Process` specifically, in native seL4, also requires a significant amount of bookkeeping (in the seL4 utility libraries). Additional bookkeeping in complex operations amortizes the cost incurred due to IPC and endpoint badging, which is why *CellulOS* is only about 1.5× slower than the baseline for these operations.

7 RELATED WORK

Related works fall into two broad categories: frameworks for building different types of isolation abstractions, and existing models that represent resources within a PD or an SoC.

Frameworks. The `clone` system call [41] can be thought of as a simple framework in which a caller controls what parts of a virtual address are shared between the caller and the newly created process [3–5, 7–9, 42]. The *unified API* we presented in Sec. 5.4 is a logical extension of Linux `clone`, as *OSmosis*'s resource space is a generic form of a namespace. However, *CellulOS* provides finer granularity in terms of what can be shared.

Genode [1] is an userspace operating system framework that runs on a capability-based microkernel such as OKL4 [25], seL4 [28], and Nova-Hypervisor [55]. *CellulOS* is heavily inspired by Genode, with two key distinctions: 1) Genode has a 1:1 mapping of protection domains to address spaces, while *CellulOS* allows a PD to have multiple address spaces, and 2) there is no model underlying the Genode framework.

Castes et. al [14] suggest decoupling the trusted parts of the system from the parts that implement the isolation mechanism and is the closest contemporary we could find whose goals include exploration of the isolation mechanism design space. Though that is our secondary goal, we believe that the first step in that journey should be to create a model of the OS.

Modeling of Operating Systems. Prior attempts to model resource sharing and isolation focus on different aspects. For instance, μScope [51] focuses on which functions access which data objects. Based on this information, they compute the ratio of extra privilege for monolithic programs. Their metric serves as an indicator of where to draw isolation boundaries. μScope does not compare isolation mechanisms or OS abstractions.

Sockeye [18, 19] 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. In *OSmosis*, this corresponds to changing map-edges in the System DAG. Sockeye mainly focuses on memory resources at the physical level, whereas *OSmosis* also includes other resources into consideration.

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 software and hardware resources are shared between applications. This lets us compare and reason about the explicit and implicit sharing between applications in a principled way using the *resource similarity index* and *fault radius.* We introduced *CellulOS*, an implementation of the *OSmosis* framework demonstrating the feasibility of realizing the model in an operating system.

A FAULT RADIUS (FR) ALGORITHM

```
transform(g)
  for space in g.spaces
    owners = [p for p in g.pds if space in p.holds]
    resources = [r for r in g.resources
                   if space in r.subset]
    dependents = [p for p in g.pds
      if p not in owners
      and not isempty
```

```
      [r for r in resources if r in p.holds]]

    forall owner in owners, dependent in dependents
      dependent.request += owner

  return dependent

calculate_fr(g, PD1, PD2)
  # Add implicit request edges to the graph
  g = transform(g)

  # Compute the shortest path from PD1 & PD2
  # to all ancestors in g
  # pds_1: set of tuples (<pd>, <n>)
  # n is the shortest path from PD1 to <pd>
  pds_1 = SSSP(PD1, g)
  pds_2 = SSSP(PD2, g)

  pds_union = [<PD, min(n1, n2)>
    forall (PD, n1) in ancestors_1
    and (PD, n2) in ancestors_2]

  return min(ancestor_union.n)
```

REFERENCES

[1] 2021. *Genode Operating System Framework.* https://genode.org/
[2] 2022. *Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family.* https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html
[3] 2022. *Ipc_namespaces(7) - Linux Manual Page.* https://www.man7.org/linux/man-pages/man7/ipc_namespaces.7.html
[4] 2022. *Network_namespaces(7) - Linux Manual Page.* https://www.man7.org/linux/man-pages/man7/network_namespaces.7.html
[5] 2022. *Pid_namespaces(7) - Linux Manual Page.* https://www.man7.org/linux/man-pages/man7/pid_namespaces.7.html
[6] 2022. *sel4test is a test suite for seL4.* https://docs.sel4.systems/projects/sel4test/
[7] 2022. *Time_namespaces(7) - Linux Manual Page.* https://www.man7.org/linux/man-pages/man7/time_namespaces.7.html
[8] 2022. *User_namespaces(7) - Linux Manual Page.* https://www.man7.org/linux/man-pages/man7/user_namespaces.7.html
[9] 2022. *Uts_namespaces(7) - Linux Manual Page.* https://www.man7.org/linux/man-pages/man7/uts_namespaces.7.html
[10] 2022. *VMCALL — Call to VM Monitor.* https://www.felixcloutier.com/x86/vmcall
[11] 2024. *Neo4j Graph DB.* https://neo4j.com/
[12] 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. *ACM SIGOPS operating systems review* (2003). https://doi.org/10.1145/1165389.945462
[13] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. 2020. {RELOAD+REFRESH}: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks. In *29th USENIX Security Symposium (USENIX Security 20).*
[14] Charly Castes, Adrien Ghosn, Neelu S Kalani, Yuchen Qian, Marios Kogias, Mathias Payer, and Edouard Bugnion. 2023. Creating Trust by Abolishing Hierarchies. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems.* 231–238.
[15] 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.* https://dl.acm.org/doi/10.1145/1460833.1460871
[16] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. 2017. Prime+abort: a timer-free high-precision L3 cache attack using intel TSX. In *Proceedings of the 26th USENIX Conference on Security Symposium (SEC'17).*
[17] Izzat El Hajj, Alexander Merritt, Gerd Zellweger, Dejan Milojicic, Reto Achermann, Paolo Faraboschi, Wen-mei Hwu, Timothy Roscoe, and Karsten Schwan. 2016. SpaceJMP: Programming with Multiple Virtual Address Spaces. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16).* Association for Computing Machinery, New York, NY, USA, 353–368. https://doi.org/10.1145/2872362.2872366
[18] 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. https://doi.org/10.1145/3625275.3625400
[19] 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.* 46–52.
[20] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. 1999. Cellular Disco: Resource management using virtual clusters on shared-memory multiprocessors. In *Proceedings of the seventeenth ACM symposium on Operating systems principles.* 154–169.
[21] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721.*
[22] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache template attacks: automating attacks on inclusive last-level caches. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC'15).*
[23] Hard Kernel. 2023. *Odroid Wiki.* https://wiki.odroid.com/odroid-c4/odroid-c4
[24] Gernot Heiser and Kevin Elphinstone. 2016. L4 Microkernels: The Lessons from 20 Years of Research and Deployment. *ACM Trans. Comput. Syst.* 34, 1, Article 1 (apr 2016), 29 pages. https://doi.org/10.1145/2893177
[25] Gernot Heiser and Ben Leslie. 2010. The OKL4 Microvisor: Convergence point of microkernels and hypervisors. In *Proceedings of the first ACM asia-pacific workshop on Workshop on systems.* 19–24.
[26] Guangyuan Hu, Zecheng He, and Ruby B. Lee. 2021. SoK: Hardware Defenses Against Speculative Execution Attacks. In *2021 International Symposium on Secure and Private Execution Environment Design (SEED).* 108–120. https://doi.org/10.1109/SEED51797.2021.00023
[27] Yuzhuo Jing and Peng Huang. 2022. Operating System Support for Safe and Efficient Auxiliary Execution. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22).* USENIX Association. https://www.usenix.org/conference/osdi22/presentation/jing
[28] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel. 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, 207–220.

https://doi.org/10.1145/1629575.1629596

[29] Younggyun Koh, Rob Knauerhase, Paul Brett, Mic Bowman, Zhihua Wen, and Calton Pu. 2007. An analysis of performance interference effects in virtual environments. In *2007 IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE, 200–209.

[30] 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. https://doi.org/10.1145/3447786.3456248

[31] J. Liedtke. 1995. On Micro-Kernel Construction. *SIGOPS Oper. Syst. Rev.* (dec 1995). https://doi.org/10.1145/224057.224075

[32] Linux KVM. 2024. Kernel Samepage Merging. Online. https://www.linux-kvm.org/page/KSM.

[33] 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 *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. https://dl.acm.org/doi/10.5555/3026877.3026882

[34] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. *2015 IEEE Symposium on Security and Privacy* (2015).

[35] Richard A. Meyer and Love H. Seawright. 1970. A virtual machine time-sharing system. *IBM Systems Journal* (1970). https://dl.acm.org/doi/10.1147/sj.93.0199

[36] Online. 2022. *The seL4 Microkernel*. https://sel4.systems/

[37] Online. 2023. *Container - OpenVZ Virtuozzo Containers Wiki*. https://wiki.openvz.org/Container

[38] Online. 2023. *Docker*. https://docs.docker.com/

[39] Online. 2023. *DOS CVE*. https://www.cvedetails.com/vulnerability-list/opdos-1/denial-of-service.html

[40] Online. 2023. *FreeBSD: Jails*. https://www.freebsd.org/cgi/man.cgi?jail

[41] Online. 2023. *Linux: Clone*. https://man7.org/linux/man-pages/man2/clone.2.html

[42] Online. 2023. *Linux: Mount Namespace*. https://www.man7.org/linux/man-pages/man7/mount_namespaces.7.html

[43] Online. 2023. *Linux: Namespaces*. https://man7.org/linux/man-pages/man7/namespaces.7.html

[44] Online. 2023. *Linux: Posix Spawn*. https://man7.org/linux/man-pages/man3/posix_spawn.3.html

[45] Online. 2023. *Solaris: Zones*. https://docs.oracle.com/cd/E36784_01/html/E36883/zones-5.html#REFMAN5zones-5

[46] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. 2015. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications *(CCS '15)*.

[47] OscarAkaElvis. 2018. *How can I call docker daemon of the host-machine from a container? - Stavk Overflow*. https://stackoverflow.com/questions/48152736/how-can-i-call-docker-daemon-of-the-host-machine-from-a-container

[48] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. *Topics in Cryptology – CT-RSA* (2006).

[49] Raesene. 2016. *The Dangers of Docker.sock*. https://raesene.github.io/blog/2016/03/06/The-Dangers-Of-Docker.sock/

[50] 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)*. https://doi.org/10.1109/IOTSMS48152.2019.8939164

[51] Nick Roessler, Lucas Atayde, Imani Palmer, Derrick McKee, Jai Pandey, Vasileios P Kemerlis, Mathias Payer, Adam Bates, Jonathan M Smith, Andre DeHon, et al. 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*. 296–311.

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

[53] 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. https://doi.org/10.1145/2988336.2988337

[54] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. https://www.usenix.org/conference/atc20/presentation/shillaker

[55] Udo Steinberg and Bernhard Kauer. 2010. NOVA: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems*. 209–222.

[56] Carl A. Waldspurger. 2002. Memory resource management in VMware ESX server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (Copyright Restrictions Prevent ACM from Being Able to Make the PDFs for This Conference Available for Downloading)* (Boston, Massachusetts) *(OSDI '02)*. USENIX Association, USA, 181–194.

[57] Nicholas C Wanninger, Joshua J Bowden, Kirtankumar Shetty, Ayush Garg, and Kyle C Hale. 2022. Isolating functions at the hardware limit with virtines. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 644–662.

[58] Andrew Whitaker, Marianne Shaw, Steven D Gribble, et al. 2002. Denali: Lightweight virtual machines for distributed and networked applications. (2002). http://web.cs.ucla.edu/~miodrag/cs259-security/whitaker02denali.pdf

[59] 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*. Association for Computing Machinery. https://doi.org/10.1145/3460120.3484744

[60] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. *USENIX Security* (2014).

[61] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2012. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (Raleigh, North Carolina, USA) *(CCS '12)*. Association for Computing Machinery, New York, NY, USA, 305–316. https://doi.org/10.1145/2382196.2382230

[62] Zirui Neil Zhao, Adam Morrison, Christopher W Fletcher, and Josep Torrellas. 2024. Last-Level Cache Side-Channel Attacks Are Feasible in the Modern Public Cloud. (2024).

[63] Ziqiao Zhou, Yizhou Shan, Weidong Cui, Xinyang Ge, Marcus Peinado, and Andrew Baumann. 2023. Core slicing: closing the gap between leaky confidential VMs and bare-metal cloud. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association.