OSmosis: Modeling & Building Flexible OS Isolation Mechanisms

Sidhartha Agrawal* Shaurya Patel* Reto Achermann Margo I. Seltzer
University of British Columbia *Student

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. Implementing each isolation mechanism as an independent abstraction makes it difficult to reason about the state and resources shared among different tasks, leading to security vulnerabilities and performance interference.

We present OSmosis, an isolation model that expresses the precise level of resource sharing, a framework in which to implement isolation mechanisms based on the model, and an implementation of the framework on seL4. The OSmosis model lets the user determine the degree of isolation guarantee that they need from the system. This determination empowers developers to make informed decisions about isolation and performance trade-offs, and the framework enables them to create mechanisms with the desired degree of isolation.

1 Motivation

New isolation mechanisms are constantly emerging and are often motivated by one of: the emergence of a new use case, improving the performance of an existing use case, or defending against a security vulnerability. However, these new mechanisms always use isolation as a tool. For example, they reserve resources (memory, storage, CPU time) [7], restrict access to unneeded state (kernel) [11], or share underlying state (drivers) [2, 3] to improve performance. Similarly, they increase the isolation of resources or underlying state to build defenses. Given the importance of varying isolation, it is imperative to have a clear understanding of which resources and state are shared when using any particular isolation mechanism. However, we lack a precise vocabulary to describe this sharing and isolation, which leads to security vulnerabilities.

Even when applications appear isolated, such as in the case of containers, they still share kernel state. For instance, Linux namespaces [9], which are a building block for containers, do not isolate all the kernel's visible state; some state can leak across container boundaries (e.g., the open file table) leading to denial-of-service attacks on other applications on the same host [12]. Additionally, since the container infrastructure and the kernel run in the same address space, a simple buffer overflow in one part of the kernel can bring down the shared kernel and both containers. Lightweight VMs such as FireCracker [2] and KataContainers [10] are more secure alternatives to Linux containers, providing the security of VMs with the overhead of containers. However, they achieve this performance by having the host OS (rather than the guest OS) provide functionality (e.g., drivers) for all VMs. Unfortunately, this leads to more shared state in the host kernel. Just as a shared kernel exposed issues with state leakage (for containers), shared drivers in the host kernel can do the same (for VMs).

2 Overview of OSmosis

We present OSmosis, which consists of three parts. First, a mathematical *model* that enables us to precisely describe the resources in the system and how they are shared. Second, a framework that identifies the features needed by an operating system to build isolation mechanisms based on the model. Third, an implementation of the framework on seL4 [6].

Model: In *OSmosis*, the running system consists of a set of *protection domains* and *resources*. Protection domains correspond to active entities such as processes, threads, and virtual machines. A protection domain has a set of resources and a *resource directory*. Resources can be either physical or virtual entities (e.g., virtual memory region, file, socket) that can be partitioned into smaller resources. The resource directory is a dictionary that identifies the protection domain responsible for satisfying a request for a resource that the current protection domain does not possess. For example, a user-level process wanting to allocate some memory will call `mmap()` to request more virtual memory resources. This corresponds to a lookup in the PD's resource directory for virtual memory resources and then requesting more virtual memory from the PD to which that resource maps. The resource relation describes dependencies between two resources (e.g., the page table keeps track of which virtual memory page depends on which physical page).

We show the flexibility of our model by describing five scenarios in Fig. 1. We focus only on memory resources, but the concepts apply to all types of resources. The arrows indicate resource relations, the rounded rectangles are resources, and the blue boxes are protection domains. No resource directories are shown in the figure. Resources A and B represent the stack resource. Two threads both have each other's stacks in their protection domain (Fig. 1 (a)). In 'threads with isolated stacks' (Fig. 1 (b)), each thread has access only to its own

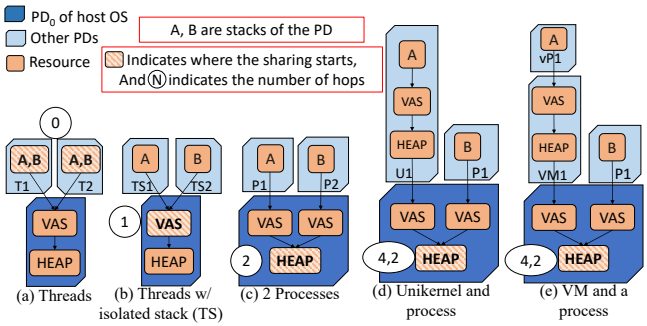

Figure 1. Five mechanisms modeled with *OSmosis*

stack. However, they are still allocated from the same address space. Two processes (Fig. 1 (c)) have separate address spaces, but their virtual address space (VAS) data structures in PD_0 depend on the kernel heap. In the case of a unikernel (Fig. 1 (d)), although there are additional levels of abstraction, address space management and the application are in the same PD. In the case of a guestOS (i.e., virtual machine), a process running on the VM is in a separate PD (Fig. 1 (e)).

Viewing the system as nodes (resources and PDs) and edges (resource relations) in a graph makes it easy to identify shared resources, whether direct or indirect. In Fig. 1, the first shared resource between the two user PDs (light blue) is shown in a patterned box, and the number of the hops at which the sharing happens is shown in the circle. In both Fig. 1 (d) and (e), the stack of the user PD running in the unikernel and VM, respectively, is four hops from the heap of the host OS. Whereas the stack of the process running directly on the host is only two hops away.

This difference in the number of hops shows that one is more isolated than the other; and more isolated says that is more difficult to leak state between them.

Framework: The *OSmosis* framework consists of three building blocks that an OS needs to realize the model. First, we need a protection domain abstraction, which is the explicit owner of a set of resources. Second, we need a way to extract the resource relation from the system to create the system's dependency graph. This information is often already present; for example, page-tables describe how virtual addresses depend on physical addresses. And finally, we need an API that instantiates a PD given the isolation properties of each resource described in the model. This is inspired by the `clone` system call in Linux [8] and is primarily syntactic sugar for the developer to create PDs with desired isolation properties easily.

Implementation: We have implemented a small portion of *OSmosis*– dealing with memory resources – using the capabilities-based microkernel seL4 [4, 6]. We chose this microkernel as it has no existing abstractions for processes, containers, or virtual machines. This lack of existing abstractions allows us to define the building blocks as we see fit. *Capabilities* and *capability spaces* [1] map well to *OSmosis*'s resources and protection domains. In seL4, the capability space is modeled as a tree, and, by sharing parts of the subtree with other protection domain, we implement the sharing of resources amongst PDs.

3 Discussion and Use cases

OSmosis allows us to explore the space of isolation mechanisms in a principled way. We discuss how the *model* enables us to reason about isolation and the *framework* lets us build new abstractions quickly.

Comparing Isolation level between PDs: Viewing the systems as a collection of resources and relations enables us to define queries on the model state that can be used to precisely compare the level of isolation between two PDs. For instance, if we take the transitive closure of the resource relations starting at a PD, we get a set of all the resources on which a PD depends. Alternatively, we can restrict the number of relations to traverse (i.e., hops) to a small number and see how many resources two PDs share for a given value of hops, e.g., what is the set of resources that are shared in the 3-hop radii of two PDs. If a pair of PDs share fewer resources at a given number of hops than another pair of PDs, we can say that the former pair is more isolated than the latter.

Existing and New Mechanisms: In Sec. 2, we showed with some examples that *OSmosis* is rich enough to capture existing mechanisms. For example, unikernels are similar to virtual machines in many respects, but the distinction between them is clearer in *OSmosis*. The application and kernel belong to the same PD, whose resources and resource directory is a subset of the union of a conventional virtual machine and process implementation. Similarly, building slight variations of existing mechanisms is trivial. For instance, to build processes that operate on a separate set of physical pages, *OSmosis* assigns different resource directory entries (with a disjoint set of pages) to the PDs of those two processes. Lightweight contexts (LwC) [5] are equally straight forward; each LwC is a separate PD, but the various PDs share only the necessary resources, e.g., virtual memory, files.

Viewing Isolation as spectrum: With *OSmosis*, by increasing the number of hops at which sharing happens for a specific type of resource, we can increase the isolation just for that resource type. Thus, we show that there exists a vast high-dimensional space of isolation primitives created by assigning different isolation levels to different resources. When deploying a new PD in a shared cloud environment, the operator can vary isolation levels for different resources against other trusted and untrusted PDs. For example, if a new threat is discovered in the networking stack, *OSmosis* enables the deployment engineer to run just the networking stack with an additional isolation level until the vulnerability is patched.

References

[1] 2021. seL4 Manual. (2021). https://sel4.systems/Info/Docs/seL4-manual-latest.pdf

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*. https://www.usenix.org/conference/nsdi20/presentation/agache

[3] Simon Kuenzer, Vlad-Andrei B*u*adoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi R*u*aducanu, Cristian Banu, Laurent Mathy, R*u*azvan Deaconescu, Costin Raiciu, and Felipe Huici. 2021. Unikraft: Fast, Specialized Unikernels the Easy Way. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21)*. Association for Computing Machinery. https://doi.org/10.1145/3447786.3456248

[4] Ben Leslie and Gernot Heiser. 2020. The sel4 core platform. *TS/sel4cp/2011-draft-spec. pdf* (2020).

[5] 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

[6] Online. 2022. The seL4 Microkernel. (2022). https://sel4.systems/

[7] Online. 2023. Linux: Cgroups. (2023). https://man7.org/linux/man-pages/man7/cgroups.7.html

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

[9] Online. 2023. Linux: Namespaces. (2023). https://man7.org/linux/man-pages/man7/namespaces.7.html

[10] 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

[11] 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

[12] 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