OSmosis: No more Déjà vu in OS isolation

Sidhartha Agrawal
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 to satisfy application requirements. Implementing each isolation mechanism as an independent abstraction makes it difficult to identify the state and resources shared among different tasks, leading to information leakage via shared kernel or hardware resources.

We present *OSmosis*, an isolation model and framework to express the precise level of resource sharing and software building blocks required to implement isolation mechanisms in operating systems. The *OSmosis* model lets us identify the extent of sharing between different applications and 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.

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 of techniques to facilitate safe multiplexing. The community continues to struggle with how to provide the right degree of sharing and isolation for a given application and its users [2, 5, 8, 11, 12, 14, 16, 18, 21, 22, 25]. 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 the application state consists of and thus, what parts of the application's state are shared with or isolated from other applications. While some application state is known (e.g., heap, code, data, and less obvious the kernel), there exists a significant amount of unknown state that the application is inadvertently sharing with other applications (e.g., undiscovered micro-architectural state in the CPU, or system-level services (Section 2.2)). Worse, we lack a common vocabulary to describe an application's resources, including its known and unknown software and hardware state. This has led to many problems ranging from performance anomalies due to unintentional sharing and overheads from too much isolation to security vulnerabilities caused by unintentional sharing of known and unknown software and hardware state [3, 6, 17, 31].

We claim that there is a need for a principled way to talk about isolation and sharing, and a framework upon which to build implementations. Our hypothesis is that all OS mechanisms can be described as a set of **resources** and the **relationships** describing dependencies among them. Resources can be the virtual memory an application uses, the files to which it has access, the OS state it can query, etc. Resources lie on a sharing spectrum ranging from wholly shared to completely isolated. The metric that defines this spectrum is the distance to the first common resource found in the resource relationships of two entities (i.e., processes, containers, virtual machines). For example, two threads are on the shared end of the spectrum, because they share a virtual address space resource. Two processes running on two different virtual machines are more isolated, because the first resource they share is state maintained by a hypervisor.

We present *OSmosis*, which is composed of two parts. First, the *OSmosis* model (Section 3.1) describes the types of entities in a system and how their sharing lies on a spectrum (Section 3.3). The model gives us a principled way to express isolation and sharing. Second, the *OSmosis* framework (Section 4) describes the required OS mechanisms and tools for implementing the isolation model. The *OSmosis* framework allows us to select a specific point in a high-dimensional space, where the different axes correspond to the different resources (e.g., physical memory, CPUs).

We have built a prototype system on the capabilities-based seL4 microkernel [13] that implements parts of our framework. We have built two existing and two new mechanisms with our framework. In contrast to existing implementations, we use the same set of building blocks for every mechanism.

OSmosis lets us model the levels of isolation for each subsystem independently. For instance, if we are more concerned about attacks in the networking stack, we can give the networking stack stronger isolation than the file system stack. When subsystems are tightly coupled, (e.g., the virtual memory and file systems), increasing the isolation level for one raises the isolation level for the other, but this tight coupling does not exist between all subsystems.

Draft, July 2023, DO NOT CIRCULATE
2023. ACM ISBN xxx-x-xxxx-xxxx-x/xx/x…$15.00
https://doi.org/xx.xxxx/xxxxxxx.xxxxxxx

Physical resource partitioning can provide performance guarantees and defenses against specific side-channel attacks. *OSmosis* makes it easy to model shared hardware and, if needed, partition resources and restrict the usage of a given part of a resource to a given application.

With *OSmosis*, we can model the isolation requirements of a given application and then easily build it using the framework. This is especially exciting 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 [10, 26, 28–30]. There is no one-size-fits-all, and for a given application, one might want to pick different degrees of isolation/sharing between applications running on the same machine.

2 MOTIVATION

New mechanisms 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, the solutions always use isolation as a tool. For example, they reserve resources (memory, storage, CPU time) [20], restrict access to unneeded state (kernel) [29], or share underlying state (drivers) [1, 10] to improve performance. Similarly, they increase the isolation of resources or underlying state to build defenses. Given the importance of varying isolation, it is useful to have a clear understanding of which resources and state are shared among applications.

2.1 New use cases

The systems community has developed many different isolation mechanisms in the last decade [4, 7, 9, 11], and each provides a slightly different degree of isolation for a different resource, such as virtual memory, open files, performance, etc. Shreds [4], Secure Memory Views (SMV) [7], and Light Weight Contexts (LwC) [11] focus on providing compartmentalization within an address space, while LwCs also provide compartmentalization of some kernel state (e.g., file descriptor table) within the same process.

Whenever a new scenario arises, a solution is built to fit it, but there is no principled way to describe and implement these solutions, making it difficult to formally distinguish different solutions from one another. With new paradigms such as Function-as-a-Service, we see many more mechanisms arise [27, 29]. Some restrict access to underlying state [29], not needed by the function. Others run multiple functions in the same process and use additional mechanisms to create intra-address-space isolation [28]. Although these mechanisms provide incremental isolation levels, their implementations are not incremental. A new implementation is prone to bugs since it cannot take advantage of years of testing on existing mechanisms [15, 19, 24].

Furthermore, some organizations do not have the engineering resources to develop a new mechanism from scratch, so applications are retrofitted into existing mechanisms. Application developers might use a mechanism that provides weaker isolation than desired, leaving them vulnerable to exploits. Alternatively, they might use a mechanism with overly strong isolation and pay more for their deployment in a shared cloud environment.

2.2 Unintentional resource sharing

The lack of clarity about the extent of sharing between two applications is also a source of security vulnerabilities. This unintentional sharing of resources can occur at the software level (e.g., kernel subsystems) or the hardware level (e.g., caches, shared buffers).

Software Even when applications appear isolated, such as in the case of a container, they still share kernel state. Since namespaces do not isolate all the visible state in the kernel, some state still leaks across container boundaries (e.g., the open file table) leading to denial-of-service attacks on other applications on the same host [31]. 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 [1] and KataContainers [26] are more secure alternatives to containers, providing the security of VMs with the overhead of containers. However, they achieve this performance by having the host OS provide functionality (e.g., drivers) for all VMs instead of the guest OS. 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).

Hardware Contemporary microprocessors leverage various optimizations in hardware, not all of which are advertised in their manuals. When two applications share the same hardware resource by time multiplexing it, they leak state even when a context-switch is done as the manual prescribes. Both Meltdown and Spectre classes [3] of attacks exploit the fact that the hardware state in the CPU is shared even when the two tasks are independent. For instance, Spectre exploits the fact that the branch prediction buffer and the cache are shared among the hardware threads in a core. Once these attacks are identified, defenses can be built. But, other unexposed hardware might still be unknowingly shared. Thus, we need a way to model the existence of microarchitectural details that could be shared but are not yet known so that if an attack is discovered in the future, the same mechanism used to isolate other resources can be used for it too. There are also known aspects of the hardware, e.g., memory buses, that face contention over shared access and can also lead to information leakage. Modeling such sharing is also useful.

3 *OSMOSIS* ISOLATION MODEL

We now present the *OSmosis* isolation model, the types of queries possible on the model, and how they lead to a precise definition of the isolation spectrum.

3.1 Model

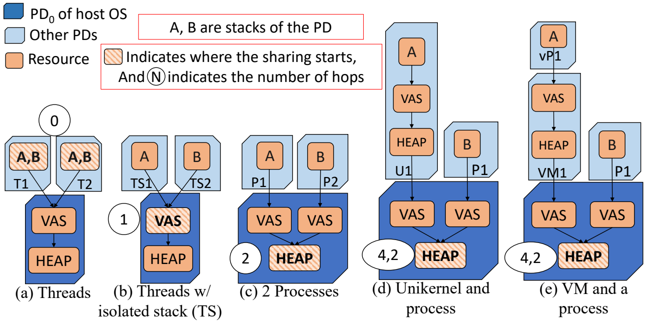
Listing 1 shows the *OSmosis* model. It consists of a *system* and a *resource relation*. A system consists of a set of *protection domains* and a set of *resources*. Protection domains (PD) correspond to active entities (e.g., processes, threads, virtual machines). A PD has a set of resources and a *resource directory*. Resources are passive entities and can be either physical (e.g., RAM, CPUs or devices) or virtual (e.g., virtual memory region, file, socket). Physical resources all derive from tangible elements; virtual resources can be created by a PD. Both types of resources can be partitioned into smaller resources. The resource directory is a dictionary, keyed by a resource, that identifies the PD responsible for satisfying a request for a resource that the current PD 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.

```
System = { pds:Set<PD>, res:Set<Resource> }
Resource Relation :: Resource x Resource
PD = { res:Set<Resource>, rdir:ResourceDirectory }
Resource = Virtual Resource | Physical Resource
Virtual Resource = Virtual Memory, File, ...
Physical Resource = RAM, Blocks, NIC, ...
ResourceDirectory<Resource> = PD for a resource
```

Listing 1: *OSmosis* Isolation Model

The *resource relation* describes dependencies between two resources. Each traversal of the relation is called a hop. There are three types of resource relations. The first is due to the system topology and does not change. For example, the contents of DRAM may be loaded in the processor caches or sent over the memory bus. The second type is added by system software. For example, the page table keeps track of which virtual memory pages are mapped to a physical page. And finally, a resource depends on the underlying resource from which it was allocated. For example, a virtual page depends on the virtual address space (resource) from which it was allocated.

We show the flexibility of our model by describing five scenarios in Fig. 1: 1) two threads in a process, 2) two threads with isolated stacks, 3) two processes, 4) a unikernel and a process, 5) a virtual machine and a process – all running on the same monolithic OS (e.g., Linux). In this example, we focus only on memory resources, but the concepts apply to all types of resources. The patterned boxes indicate where the sharing begins, and the ovals indicate the number of hops at which the sharing happens. 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 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)).



(a) Threads (b) Threads w/ isolated stack (TS) (c) 2 Processes (d) Unikernel and process (e) VM and a process

Figure 1: Five mechanisms modeled with *OSmosis*

None of the PDs for threads, threads with isolated stacks, or processes have direct access to physical resources. Instead, when they need physical memory (i.e., on a page fault), the Resource Directory indicates that PD_0 (i.e., the operating system) will handle requests for physical memory. In contrast, the guest OS handles such requests from the process running inside the virtual machine, while the host OS handles requests from the hypervisor and its native process (P1). When PD_0 maps a physical page to a virtual page, conceptually, it adds an entry to the resource relation, even though in implementation, this information is recorded in a page table.

3.2 Queries

We now define queries on the model to extract information about PDs, their resources, and most importantly the relationship among resources in different PDs. In the next section, we show how to use this information to define the *isolation spectrum.*

NHopResources: The *resource relation* lists the possible "one-hop" dependencies of a resource. However, as we saw in the discussion of virtual addresses and processor caches, there may be multiple levels of dependencies. Thus, to identify all the resources on which a specific resource depends, we compute the *n-hop transitive-reflexive* closure for n=INFINITY on the *resource relation.*

$$NHopResources :: \mathbb{N} \Rightarrow Set\text{<}Resource\text{>} \Rightarrow Set\text{<}Resource\text{>}$$

$$NHopResources :: n\ R = \bigcup_{r \in R} ResourceRelation^{n} + \varepsilon$$

We express the possibility of having some unknown micro-architectural state with the resource set ε.

Referring back to Fig. 1(c), consider the stack virtual memory region (A). The stack's one-hop closure includes the VAS(in PD_0); The VAS depends on the heap virtual memory resource of PD_0 from which its metadata was allocated; its two-hop closure includes cache sets (assuming virtually indexed caches) and the physical pages that have been allocated to the virtual memory region.

3.3 Isolation Spectrum

Using the model and its queries, we can now define different forms of isolation as points on a *spectrum* and thus, we can quantify how isolated two PDs are from each other.

NHopResourcesOfPD: We derive the n-hop resources of a PD by computing the NHopResource function on the PD's resources unioned with the $(n-1)$-hop computation of NHopResourcesOfPD for each PD in the resource directory.

$$NHopResourcesOfPD :: \mathbb{N} \Rightarrow PD \Rightarrow Set\text{<}Resource\text{>}$$

$$NHopResourcesOfPD :: n\ pd = (NHopResources\ n\ pd.res) \cup \bigcup_{p \in pd.rdir.values} (NHopResourcesOfPD\ (n-1)\ p)$$

Note, this includes both the resources currently accessible as well as those to which it may acquire access in the future.

NHopShared: We can now express the degree of sharing between two PDs by examining the intersection of the sets produced by NHopResourcesOfPD for any values of n.

$$NHopShared :: \mathbb{N} \Rightarrow \mathbb{N} \Rightarrow PD \Rightarrow PD \Rightarrow Set\text{<}Resource\text{>}$$

$$NHopShared :: n_1\ n_2\ pd_1\ pd_2 = ((NHopResourcesOfPD\ n_1\ pd_1) \cap (NHopResourcesOfPD\ n_2\ pd_2))$$

NHopIsolated: We say that two PDs are *n-hop isolated* if they do not share any resources within n hops of either PD, subject to an exclusion set, δ. The exclusion set $\delta = \gamma \cup \varepsilon$ is the union of the set of resources the application does not care about (γ) and the set of unknown micro-architectural resources (ε). When two processes do not care about sharing a cache or the file system, we add those resources to γ. We say that a PD is *NHopIsolated in the system* if it is *NHopIsolated* with every other PD.

$$NHopIsolated :: \mathbb{N} \Rightarrow \mathbb{N} \Rightarrow Set\text{<}Resource\text{>} \Rightarrow PD \Rightarrow PD \Rightarrow bool$$

$$NHopIsolated :: n_1\ n_2\ \delta\ pd_1\ pd_2 = NHopShared\ n_1\ n_2\ pd_1\ pd_2 \subseteq \delta$$

IsolationLevel: Given two PDs, we define *Isolation Level* between them as the number of hops at which sharing begins. We find the minimum value of n_1 or n_2 for which *NHopIsolated* is false. Taking the minimum of the tuple ensures proper accounting for asymmetric configurations. In Fig. 1(e) the isolation level is 2 derived from $min(4, 2)$, which means that sharing starts at 2 hops from at least one of the PDs.

$$IsolationLevel :: PD \Rightarrow PD \Rightarrow Set\text{<}Resource\text{>} \Rightarrow \mathbb{N}$$

$$IsolationLevel :: pd_1\ pd_2\ \delta = n \mid \forall n_1 n_2\ \neg NHopIsolated\ n_1\ n_2\ \delta\ pd_1\ pd_2 \implies n \le min(n_1, n_2)$$

One could argue that it's more useful to instead view isolation level asymmetrically, i.e. from the perspective of each PD as opposed to between PDs. Both perspectives have merit and we believe that more experience in implementing various configurations will shed insight into which is more useful. In either case, the model provides all the information necessary to engage in this debate, and in fact, without the model, such debates cannot happen.

4 *OSMOSIS* FRAMEWORK

We now map the model described in the previous section to the functionality required to realize it.

4.1 Unified API

Our model enables a unified API that can create any type of PD (inspired by the `posix_spawn` in Linux [23]). *newPD* takes a set of *resources* and a *resource directory*. By default (i.e., if the resource directory is empty), a PD directs requests for resources not given during its creation to the PD that created it (e.g., processes redirect those requests to the OS).

```
directory = currentResourceDirectory();
vas = newVAS();

// Get code, stack, heap, and vCPU resources
code, heap, stack = load(vas, "binary");
vCPU = newVCPU();
resources = {code, heap, stack, vCPU};

// Create the PD
pdID = newPD(resources, directory);
```

Listing 2: Process creation in *OSmosis*

Listing 2 shows the pseudo code to create a normal process in *OSmosis*. Since a process runs in a separate address space, we first create a new virtual address space resource. The `load` call reads a binary file and allocates from the `vas` resource to create `code`, `heap`, and `stack` resources.

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` [21], *clonePD* creates a new PD by calling an *isolation function* that defines how each resource and resource directory entry is shared with its parent (or another PD) before calling *newPD*. It is not fundamental to our model or framework, but it is the syntactic sugar that makes the model easier to use. We have written a few *isolation functions*, for example, one that creates a process or a thread with a slightly different address space and one where the resource directory entries for different types of resources are different PDs. We can also define *isolation functions* that set up resources for the new PD based on the degrees of isolation (n_1, n_2) for each resource and a δ. We envision having a suite of such template functions in a userspace library.

4.2 Determining resource relations

The resource relation (Section 3.1) makes it possible to determine what underlying resources are shared; this knowledge is crucial for identifying possible side channels. The resource relation captures all dependencies between resources and thus can become quite large. Yet, much of the information that is conceptually part of the resource relation is already present in the system. For example, Linux provides the `sysfs`, `procfs`, `dev` file systems, which describe system topology; page tables store virtual address to physical address dependencies. Dependencies between resources and the PDs that allocated them are implied. Designing an API to query the resource relation is straightforward; building an efficient implementation of that query API is a more interesting problem. Fortunately, queries of the resource relation are not on the performance critical path of normal operation.

5 DISCUSSION AND USE CASES

OSmosis enables 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.

Existing and New Mechanisms: In Section 3.1, 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) [11] are equally straight forward; each LwC is a separate PD, but the various PDs share only the necessary resources, e.g., virtual memory, files.

Side channel awareness: The *OSmosis* model explicitly identifies where protection domains share state; this shared state exposes a potential side channel. Using *NHopShared* described in Section 3.3, we can determine the resources shared between PDs. The greater the number of hops, the more difficult it is to exploit the side channel. Yang et al. and Gao et al. exploited the data structures shared in the kernel [6, 31]. These data structures would be part of the shared state in our model.

OSmosis also lets us model the sharing of some unknown microarchitectural resource, via ε (Section 3.2). As these resources become known, they become part of the resource relation. For example, a branch target buffer might initially be a piece of the unknown shared microarchitectural resource. When an exploit is discovered, we can make the branch target buffer explicit in the resource relation.

Viewing Isolation as spectrum: With *OSmosis*, it is possible to provide different isolation levels for different resources. By varying n_1, n_2 and δ in *NHopIsolated* shown in Section 3.3, 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 these three parameters against other trusted and untrusted PDs. And for a given deployment, the operator can use *IsolationLevel* to determine the degree of isolation between two untrusting 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 a vulnerability is patched.

6 CONCLUSION

We identify the problem that there is a lack of understanding about the level of isolation and sharing provided by different isolation mechanisms. Additionally, the plethora of isolation mechanisms do not share an underlying framework, which makes it challenging to build new mechanisms.

We present the *OSmosis* model, which lets us reason about isolation between applications in a principled way. It lets us model precisely which parts of the known and unknown hardware and software state are shared between applications.

We then present the *OSmosis* framework, designed to realize the model by identifying the essential building blocks.

Finally, we show how *OSmosis* lets us model and build new and existing mechanisms that are precisely tailored to the user's isolation requirements, identify potential side channels, and view the isolation of resources as a spectrum.

REFERENCES

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

[2] 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

[3] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A systematic evaluation of transient execution attacks and defenses. In *28th USENIX Security Symposium (USENIX Security 19)*. https://dl.acm.org/doi/10.5555/3361338.3361356

[4] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. 2016. Shreds: Fine-grained execution units with private memory. In *2016 IEEE Symposium on Security and Privacy (SP)*. https://ieeexplore.ieee.org/document/7546495

[5] 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

[6] 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*. https://doi.org/10.1145/3319535.3354227

[7] 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. https://doi.org/10.1145/2976749.2978327

[8] 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

[9] 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

[10] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ștefan 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

[11] 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

[12] 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

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

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

[15] Online. 2023. *CWE-243: Creation of chroot Jail Without Changing Working Directory*. https://cwe.mitre.org/data/definitions/243.html

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

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

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

[19] Online. 2023. *KVM: CVE Details*. https://www.cvedetails.com/product/19922/Redhat-KVM.html?vendor_id=25

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

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

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

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

[24] Online. 2023. *LXC: CVE Details*. https://www.cvedetails.com/vulnerability-list/vendor_id-13134/product_id-27105/Linuxcontainers-LXC.html

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

[26] 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

[27] Ali Raza, Ibrahim Matta, Nabeel Akhtar, Vasiliki Kalavri, and Vatche Isahagian. 2021. SoK: Function-As-A-Service: From An Application Developer's Perspective. (2021). https://doi.org/10.5070/SR31154815

[28] 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

[29] 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

[30] 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

[31] 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