Say Goodbye to Virtualization for a Safer Cloud

Dan Williams Ricardo Koller Brandon Lum
IBM T.J. Watson Research Center, Yorktown Heights, NY

When it comes to isolation on the cloud, conventional wisdom holds that virtual machines (VMs) provide greater isolation than containers because of their low-level interface to the host. A lower-level interface reduces the amount of code and complexity needed in the kernel that must be relied upon for isolation. However, it is incorrectly assumed that virtualization mechanisms are required to achieve a low-level interface suitable for isolation. In this paper, we argue that the interface to the host can be lowered for any application by moving kernel components to userspace. We show that using a userspace network stack results in a 33% reduction in kernel code usage, which is 20% better than when resorting to virtualization mechanisms and using a VM.

1 Introduction

Cloud security is an ever-present and growing concern as more enterprises consider running workloads in the cloud. Multi-tenancy brings challenges to security, as cloud users must trust the cloud provider to maintain isolation between their workloads and any potentially malicious tenants that are co-located with them on the same physical infrastructure. Currently, running each tenant in its own virtual machine (VM) is the most common practice used by cloud providers for isolating tenants.

However, as containers gain popularity as a lightweight, developer-friendly alternative to VMs, it is reasonable to ask whether VMs are still needed for isolation. Arguments for isolation typically boil down to discussions about differences in the interface between the guest and the host, which are often conflated with the mechanism in use. VMs, using hardware virtualization mechanisms, utilize a *low-level interface* which is thought to provide strong isolation. Containers, using process mechanisms, utilize a *high-level interface*, which is thought to contribute to poor isolation. For example, the authors of LightVM describe the cause of container security concerns as follows:

> The main culprit is the hugely powerful kernel syscall API that containers use to interact with the host OS. [23]

Yet LightVM then goes on to try to introduce container-like properties to VMs, without giving up the complex hardware-based virtualization *mechanism* implementing the low-level interface to guests.

In this paper, we make the observation that the level of the interface between the guest and the host is *not* fundamentally tied to the actual mechanism used to separate the guest from the host (like the virtualization or process mechanism). In Section 3, we describe how the high-level container interface can be arbitrarily lowered by introducing library OS (libOS) components to applications or using userspace services in a manner we dub *microkernelification*. While we do not focus on enforcing isolation in this paper, we assume guards can protect a given interface despite the mechanism.[1]

Furthermore, we claim that the mechanism used by virtualization actually *hurts* isolation due to its complexity. To back up this claim, we introduce a rough metric based on kernel function tracing, *kernel code usage*, with the assumption that an interface that requires less kernel code results in greater isolation for tenants. We find that applying libOS techniques to the network stack alone is enough to produce a 20% smaller kernel code usage than traditional virtualization, which is 33% smaller than regular Unix processes.

2 Isolation and Kernel Code Usage

The ability to correctly isolate tenants from each other is a fundamental part of designing and operating a cloud. However, there is no trivial quantitative metric for isolation. Instead, practitioners often resort to heuristics such

[1] For example, using virtualization mechanisms, a monitor may implement guards, whereas using process mechanisms, system call whitelisting (e.g., `seccomp`) may implement them.

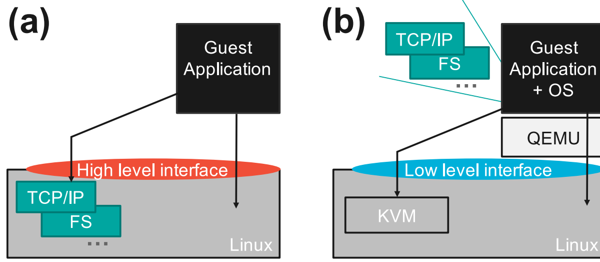

Figure 1: (a) A process/container typically accesses the host kernel through a high-level interface, such as using its network stack via `socket` system calls. (b) A VM uses a low-level interface, such as using a network tap device to process layer 2 packets (the TCP stack is implemented in the guest kernel).

as how high-level (abstract) or low-level (hardware-like) the interface between cloud applications (guests) and the host is. For example, a process expects a high-level, abstract POSIX interface, whereas a VM expects a low-level virtual hardware interface.

A system that implements a high-level interface hides complexity to the user at the cost of becoming more complex itself, usually in the form of an increased code base. More code can lead to more errors and vulnerabilities; the industry average is between 15 and 50 errors per 1000 lines of code [24, 10]. Conservatively, any vulnerability in the underlying system that provides isolation can lead to an exploit that violates the isolation it provides. Thus we say a system that can be implemented with less code (e.g., one that provides a low-level interface) can provide greater isolation.

Suppose we consider the Linux kernel as the underlying system that provides isolation between tenants in a cloud environment. Most familiarly, as seen in Figure 1(a), Linux provides a high-level POSIX interface for processes. Alternatively, using the KVM kernel module, Linux can provide a low-level virtual hardware interface. Figure 1(b) shows such a case, where the guest application has its own OS and uses low level (hardware-like) abstractions from the kernel. We define *kernel code usage* as the number of unique kernel functions that are used throughout the execution of cloud applications. This metric provides an estimate of how much of the kernel is needed to implement high or low-level interfaces. As described above, we would expect Linux configured to provide a low-level interface to result in lower kernel code usage (and thus greater isolation) than Linux configured to provide a high-level interface including system calls that interact with large, complex subsystems like the network stack and filesystems.

In a practical setting, the kernel code usage metric could be used by cloud practitioners to lock down cloud applications to some fixed interface, via guards. In a more extreme case, if the kernel code usage for a given interface was small enough, the cloud provider could implement a hypervisor, microkernel [18] or separation kernel [25, 27] with formal methods [18, 14] that exposes the desired interface, providing high assurance that the isolation property remains upheld.

3 No Need for Virtualization Mechanisms

In this section, we claim that the interface mechanism (i.e., the mechanism used to implement processes vs. VMs) has little to do with the interface level. Furthermore, we describe how to arbitrarily lower the interface level for any process, thereby reducing the kernel code usage and improving isolation.

3.1 Interface Mechanism $\neq$ Interface Level

Processors have evolved their *interface mechanism* (protection features) by introducing protection rings and execution modes in order to adapt to new paradigms. Initially, three protection rings were developed to help implement processes [3]. We refer to these as the *process interface mechanism*. Since then, processes have been used to implement containerized applications. In response to the popularity of VMs, a hypervisor mode was introduced (VT in Intel [19]), which is used in virtual machine monitors like QEMU/KVM (see Figure 1(b)) to implement VMs. We refer to this as the *virtualization interface mechanism*. Cloud applications run in VMs on top of a guest kernel which interacts with a monitor process, QEMU. QEMU interacts with the KVM module in the Linux kernel running in hypervisor mode. Both the monitor (via the process mechanism) and the guest (via the virtualization mechanism) interact with the kernel through low-level interfaces.[2]

Most likely for historic reasons, people tend to equate the interface mechanism with what it was designed for, specifically, use of the virtualization mechanism tends to be associated with low-level-interface VMs. However, equating the interface mechanism to the interface level or drawing the conclusion that low-level interfaces require the virtualization mechanism is an invalid generalization.

3.2 Lowering Process Interfaces

Figure 2 shows the expected relationship of interface level to kernel code usage. Unsurprisingly, we depict tra-

[2] Although out of the scope of this paper, we note that new interface mechanisms are emerging with enclave technologies such as Intel Software Guard Extensions (SGX) [8]. Our point still applies: the interface level used on them is completely arbitrary.

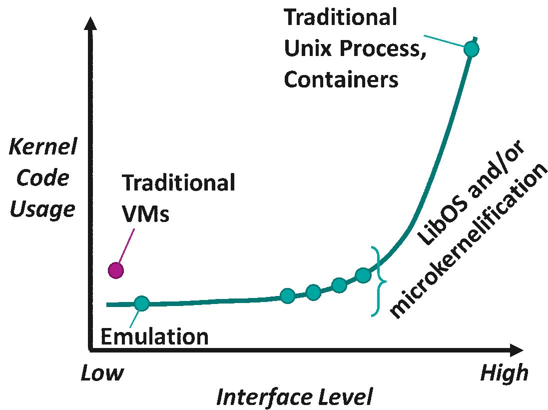

Figure 2: The level of interface between the guest and host can be lowered by libOS techniques or microkernelification, without using virtualization mechanisms.

ditional VMs in the lower left, at a point with a low interface level and low kernel code usage. Similarly, we depict traditional Unix processes/containers as a point with a high interface level and subsequently high kernel code usage.

However, process mechanisms can be used with dramatically lower-level interfaces than standard Unix processes, depicted by a line on Figure 2.[3] For example, as shown in Figure 3(a), an application can choose to link with its own network stack and use the low-level network tap device as its network interface to the kernel [9, 15, 13, 21]. In general, this approach of moving kernel functionality into libraries is called the *library OS* or *libOS* approach. LibOSes have been well studied [11] and are currently experiencing a rejuvenation in the context of the cloud due to rise of *unikernels* [22, 6, 1] and the ecosystems that support them, such as MirageOS [22].

More generally, kernel functionality can be moved into user space daemons as shown in Figure 3(b). We refer to this process as *microkernelification*, as the resultant architecture approaches a traditional microkernel architecture, with the kernel doing little more than inter-process communication. For example, FUSE [2] is a technique in which the kernel implements a small amount of bridging code that allows a process to consume a filesystem that is implemented entirely in userspace. As a result, the applications (taken as a set) interface with the kernel at a lower level, beneath the filesystem, which will result in a lower kernel code usage. `userfaultfd` [7] is an example of a similar strategy for memory fault handlers.

At the extreme, software approaches can be used to

[3]This observation holds in the other direction as well: virtualization mechanisms can be used for high-level interfaces, as in Dune [4].

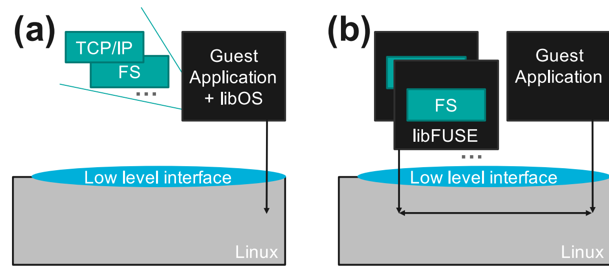

Figure 3: (a) A libOS can have a low-level interface to the host, without the kernel complexity of a VM. For example, the application can implement its own user-level TCP/IP library (like lwip [9]). (b) Kernel functionality can be implemented in user-level components, such as FUSE [2] filesystems through microkernelification.

move the interface level down to the machine level by emulating machine instructions without hardware support (as is possible in QEMU [5]). Full emulation would only need to interact with the kernel for I/O, and in that case, it can minimize the amount of kernel code used by only interacting with the lowest interface levels available: network tap and block devices.

To summarize, as shown in Figure 2, the interface level (and subsequently kernel code usage) is not dependent on using virtualization mechanisms.

4 Dangers of Virtualization Mechanisms

In this section, we examine how lowering the level of the interface of cloud applications via libOS techniques affects isolation indirectly through measuring kernel code usage. We also demonstrate that the use of virtualization mechanisms via KVM has a significant impact on the kernel code usage.

4.1 Methodology

In this experiment, we use the Linux kernel `ftrace` functionality to observe the Linux kernel code usage for a simple HTTP web server on various isolation configurations, described in Table 1.

In order to provide a comparable application in each scenario, we use the MirageOS unikernel ecosystem to build the application. MirageOS is an ecosystem of libOS components built in OCaml. MirageOS tooling allows an OCaml application to be built as standalone VMs or normal unix applications with and without libOS components.

We measure the kernel code usage for cloud applications built using MirageOS under four different scenarios (described in Table 1). The first two configurations are

Name	How much LibOS?	Networking	Description
qemu	all	tap	VM (unikernel) running on QEMU in emulation mode.
kvm	all	tap	VM (unikernel) running on QEMU with KVM.
tap	network stack	tap	Unix process using MirageOS network stack.
socket	none	sockets	Unix process built with MirageOS using host network stack.

Table 1: MirageOS configurations for kernel code usage experimentation.

VMs. Both *qemu* and *kvm* use the same image, however one is run in emulation mode (*qemu*) while the other is run with KVM assistance (*kvm*). Comparing these configurations should allow us to estimate the effects of using KVM on the kernel code usage. The second two configurations are processes. Both *tap* and *socket* are built with MirageOS, but *tap* uses the MirageOS network stack while *socket* uses the Linux host's network stack. Comparing these configurations should allow us to observe the reduction in the kernel code usage achieved by applying a libOS technique, even if it is just on the network stack.

To compute kernel code usage, we count the number of functions in the kernel that are invoked through an interface using the Linux kernel's `ftrace` functionality. We use the function graph tracer set to ignore `irqs` and trace only the `pids` that correspond to our application. In order to avoid contaminating the kernel trace, we run all experiments inside a virtual machine including *kvm*; this is possible via nested virtualization. The test VM is running a stock Ubuntu Linux kernel, version 4.10.0-38-generic. In all scenarios except for *socket*, we bridge the test VM's network device with the tap interface. We start and stop `ftrace` in the test VM entirely through the serial console to avoid any tracing contamination from SSH.

For each scenario, we measure the number of unique kernel functions that were accessed when issuing a single `wget` for a 1354 byte page in each scenario. We do not start the `ftrace` measurement until after the web server is idle and waiting for requests, to avoid polluting the trace with startup-related function calls. We use a single connection in this experiment to ensure that the size of kernel traces remain small.

4.2 Results

Figure 4 shows the total number of kernel functions accessed and also shows a classification of whether each function in the `ftrace` log corresponds to virtualization, networking, or something else. We perform classification based on manual inspection of function names. For example, we classify a function as *virtualization* if its name contains `kvm`, `vmx`, `x86`, `vmcs`, `vcpu`, or `emulator`; *network* if its name contains: `tcp`, `br_`, `skb`, `inet`, `ip`, `sock`, `napi`, `net`, `packet`, `sk_`, or `tun`; and *other* otherwise. This classification is not perfect; in particular, we expect we have missed some functions resulting in them being incorrectly classified as *other*.

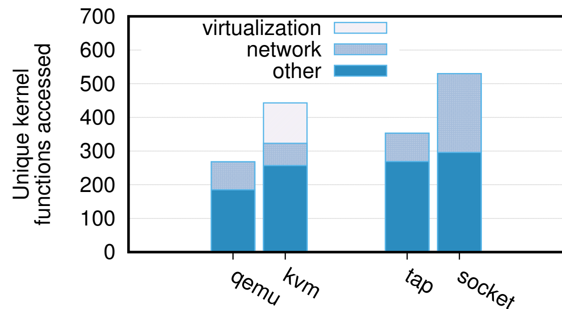

Figure 4: The number of unique kernel functions accessed when serving HTTP.

	tcp	inet	ip	sock	sk_	net	tun	br_	skb	packet	napi	total
qemu	2	1	1	4	8	12	7	11	33	0	4	83
kvm	0	0	3	3	9	9	6	11	25	0	0	66
tap	2	1	1	4	9	12	7	11	33	0	4	84
socket	62	14	16	34	30	25	0	13	35	1	4	234

Table 2: Breakdown of network-related functions.

The first observation we draw from Figure 4 is the effect of libOS techniques on the kernel code usage. Recall that *tap* and *socket* are processes that differ only by the network stack, with the former using a libOS stack. As seen in Table 2, the use of the libOS network stack cuts the total number of network-related kernel functions called by almost two thirds, from 234 to 84. As a result, the total kernel code usage shrinks by a factor of 1.5, from 530 to 353. Table 2 also shows the breakdown of the network-related functions, based on function names that contain each substring. Packet handling and the bridge in the test VM (`br_`, `skb`, `packet`, `napi`) are common to all scenarios, whereas the higher-level network stack functions (leftmost 6 columns) are replaced with lower-level tap functions (`tun`).

Comparing *tap* and *kvm*, we see that even a partial

	kvm	vmx	vcpu	emulator	x86	vmcs	Total
qemu	0	0	0	0	0	0	0
kvm	62	28	18	0	3	1	112
tap	0	0	0	0	0	0	0
socket	0	0	0	0	0	0	0

Table 3: Breakdown of virtualization-related functions.

application of libOS techniques is enough to produce a lower kernel code usage than traditional KVM/QEMU virtualization. In particular, *tap*, through the use of the libOS network stack, has a kernel code usage of about 80% of that of *kvm*, with 353 and 443 total calls respectively. Table 3 shows the breakdown of the virtualization-related functions.

The *qemu* and *kvm* bars correspond to VMs, and as such can be roughly viewed as providing insight into what the kernel code usage would look like if a full libOS was added to a process. The *qemu* result is in some sense a best case, with the smallest kernel code usage, because it uses emulation (in userspace) instead of additional kernel mechanisms (KVM) to implement virtualization. In fact, using virtualization support via KVM increases the kernel code usage by over 1.5 times, from 268 in *qemu* to 443 in *kvm*, 120 of which we have classified as *virtualization*. We expect that a full libOS implementation (instead of just the network stack as in *tap*) would yield similar results to *qemu*.

5 Discussion and Related Work

Generality: The obvious question for libOS approaches is how complete the libOS implementation is. POSIX compatibility is often discussed in the context of libOSes. OSv [17] can support complex applications including those running with the JVM. Furthermore, metrics exist to help design libOSes for completeness [26].

Maintenance: Kernel code is well maintained. Who will maintain the userspace equivalents? One answer could be that language communities with experience maintaining package ecosystems in their language of choice naturally embrace lower-level libraries. This is happening to some extent in the unikernel communities, such as MirageOS [22] and the OCaml community. Another answer could be to reuse existing (maintained) kernels without modification. Anykernels [16], rump kernels and rumprun [1] have demonstrated that, if architected in a particular way, pieces of community-supported kernels like NetBSD can be used directly as libraries with no modifications. As another example, Stackmap [28] uses the Linux kernel's network stack in userspace.

qemu	*kvm*	*tap*	*socket*
385.8	2820.1	2988.0	4250.4

Table 4: Requests/second for HTTPS server under various configurations.

Performance: When the interface to the application changes, performance can change for a few different reasons. Most obviously, although *qemu* shows the smallest kernel code use in Section 4, it is not a practical approach because of the performance implications of full emulation. In a throughput experiment (Table 4) with a MirageOS HTTPS web server under a load of 1000 connections per second we observe similar throughput from *kvm* and *tap* but confirm the inefficiencies of *qemu* losing 91% of the performance. The mechanism itself also affects performance, for example, we have measured improvements from 2355 to 1094 cycles when comparing `vmexits` from the virtualization mechanism to `sysenters` for the process mechanism. Moreover, the level of the interface may affect the number of interface crossings; for example, a libOS may cross for each and every packet, whereas a native process may only perform one crossing for a large stream write. Finally, the quality of implementation may differ between libOS implementations and native kernel implementations (e.g., a highly-optimized network stack like the one in Linux, vs. a home-grown stack). As discussed above, it is possible that native kernel implementations can be reused in libOSes.

Metrics for isolation: We have largely focused on kernel code usage as a metric for isolation. However, there are many other metrics to consider. For example, a better metric for security may not be the size of the code reachable by the interface, but how many vulnerabilities are expected to be found in the code. Lock-in-pop [20] is a system that uses libOS techniques to restrict kernel usage to popular code paths. Alternatively, an isolation mechanism may need to consider data sharing rather than code coverage, in which taint-tracking techniques may help. [12]

6 Conclusion

When thinking about isolation for the cloud, we should not view virtualization mechanisms as necessary but see them for what they are: overly complex ways to achieve a low-level interface. It is time to shed complexity and move on to a safer cloud without virtualization.

References

[1] The rumprun unikernel and toolchain for various platforms. https://github.com/rumpkernel/rumprun, Apr. 2015.

[2] The reference implementation of the linux fuse (filesystem in userspace) interface. https://github.com/libfuse/libfuse, Mar. 2018.

[3] ARPACI-DUSSEAU, R. H., AND ARPACI-DUSSEAU, A. C. *Operating Systems: Three Easy Pieces*, 0.91 ed. Arpaci-Dusseau Books, May 2015.

[4] BELAY, A., BITTAU, A., MASHTIZADEH, A., TEREI, D., MAZIÈRES, D., AND KOZYRAKIS, C. Dune: Safe user-level access to privileged CPU features. In *Proc. of USENIX OSDI* (Hollywood, CA, Oct. 2012).

[5] BELLARD, F. QEMU, a fast and portable dynamic translator. In *Proc. of USENIX Annual Technical Conf. (FREENIX Track)* (Anaheim, CA, Apr. 2005).

[6] BRATTERUD, A., WALLA, A.-A., HAUGERUD, H., ENGELSTAD, P. E., AND BEGNUM, K. Includeos: A minimal, resource efficient unikernel for cloud services.

[7] CORBET, J. Page faults in user space. `https://lwn.net/Articles/615086/`, Oct. 2014.

[8] COSTAN, V., AND DEVADAS, S. Intel sgx explained. *IACR Cryptology ePrint Archive 2016* (2016), 86.

[9] DUNKELS, A. Design and implementation of the lwip tcp/ip stack. *Swedish Institute of Computer Science 2* (2001), 77.

[10] EDWARDS, N., AND CHEN, L. An historical examination of open source releases and their vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 183–194.

[11] ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE, J. W. Exokernel: An operating system architecture for application-level resource management. In *Proc. of ACM SOSP* (Copper Mountain, CO, Dec. 1995).

[12] ERMOLINSKIY, A., KATTI, S., SHENKER, S., FOWLER, L., AND MCCAULEY, M. Towards practical taint tracking. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-92* (2010).

[13] HAN, S., MARSHALL, S., CHUN, B.-G., AND RATNASAMY, S. Megapipe: A new programming interface for scalable network i/o. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 135–148.

[14] HEITMEYER, C. L., ARCHER, M., LEONARD, E. I., AND MCLEAN, J. Formal specification and verification of data separation in a separation kernel for an embedded system. In *Proceedings of the 13th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2006), CCS '06, ACM, pp. 346–355.

[15] JEONG, E., WOO, S., JAMSHED, M. A., JEONG, H., IHM, S., HAN, D., AND PARK, K. mtcp: a highly scalable user-level tcp stack for multicore systems. In *NSDI* (2014), vol. 14, pp. 489–502.

[16] KANTEE, A., ET AL. Flexible operating system internals: the design and implementation of the anykernel and rump kernels.

[17] KIVITY, A., LAOR, D., COSTA, G., ENBERG, P., HAREL, N., MARTI, D., AND ZOLOTAROV, V. OSv optimizing the operating system for virtual machines.

[18] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. seL4: Formal verification of an OS kernel. In *Proc. of ACM SOSP* (Big Sky, MT, Oct. 2009).

[19] LEUNG, F., NEIGER, G., RODGERS, D., SANTONI, A., AND UHLIG, R. Intel virtualization technology: Hardware support for efficient processor virtualzation. *Intel Technology Journal 10*, 3 (Aug. 2006).

[20] LI, Y., DOLAN-GAVITT, B., WEBER, S., AND CAPPOS, J. Lock-in-pop: securing privileged operating system kernels by keeping on the beaten path. In *Annual Technical Conference USENIX ATC 17* (2017), pp. 1–13.

[21] LIN, X., CHEN, Y., LI, X., MAO, J., HE, J., XU, W., AND SHI, Y. Scalable kernel tcp design and implementation for short-lived connections. *SIGOPS Oper. Syst. Rev. 50*, 2 (Mar. 2016), 339–352.

[22] MADHAVAPEDDY, A., MORTIER, R., ROTSOS, C., SCOTT, D., SINGH, B., GAZAGNAIRE, T., SMITH, S., HAND, S., AND CROWCROFT, J. Unikernels: Library operating systems for the cloud. In *Proc. of ACM ASPLOS* (Houston, TX, Mar. 2013).

[23] MANCO, F., LUPU, C., SCHMIDT, F., MENDES, J., KUENZER, S., SATI, S., YASUKATA, K., RAICIU, C., AND HUICI, F. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 218–233.

[24] MCCONNELL, S. *Code complete*. Pearson Education, 2004.

[25] RETO BUERKI, AND ADRIAN-KEN RUEEGSEGGER. Muen: An x86/64 separation kernel for high assurance. `http://muen.sk`, Mar. 2018.

[26] TSAI, C.-C., JAIN, B., ABDUL, N. A., AND PORTER, D. E. A study of modern linux api usage and compatibility: what to support when you're supporting. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), ACM, p. 16.

[27] WEST, R., LI, Y., MISSIMER, E., AND DANISH, M. A virtualized separation kernel for mixed-criticality systems. *ACM Trans. Comput. Syst. 34*, 3 (June 2016), 8:1–8:41.

[28] YASUKATA, K., HONDA, M., SANTRY, D., AND EGGERT, L. Stackmap: Low-latency networking with the os stack and dedicated nics. In *USENIX Annual Technical Conference* (2016), pp. 43–56.