# The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers

ALLISON RANDAL, University of Cambridge

The common perception in both academic literature and industry today is that virtual machines offer better security, whereas containers offer better performance. However, a detailed review of the history of these technologies and the current threats they face reveals a different story. This survey covers key developments in the evolution of virtual machines and containers from the 1950s to today, with an emphasis on countering modern misperceptions with accurate historical details and providing a solid foundation for ongoing research into the future of secure isolation for multitenant infrastructures, such as cloud and container deployments.

CCS Concepts: • **Software and its engineering** → **Distributed systems organizing principles**; • **Security and privacy** → **Operating systems security**; • **General and reference** → *Surveys and overviews*;

Additional Key Words and Phrases: Containers, virtual machines, virtualization

**ACM Reference format:**
Allison Randal. 2020. The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers. *ACM Comput. Surv.* 53, 1, Article 5 (February 2020), 31 pages.
https://doi.org/10.1145/3365199

## 1 INTRODUCTION

Many modern computing workloads run in multitenant environments, where each physical machine is split into hundreds or thousands of smaller units of computing, generically called *guests*. Cloud and containers are currently the leading approaches to implementing multitenant environments. The guests in a cloud deployment are commonly called *virtual machines* or *cloud instances*, whereas the guests in a container deployment are commonly called *containers*. Typically, a single *tenant* (a user or group of users) is granted access to deploy guests in an orchestrated fashion across a cloud or cluster made up of hundreds or thousands of physical machines located in the same data center or across multiple data centers, to facilitate operational flexibility in areas such as capacity planning, resiliency, and reliable performance under variable load. Each guest runs its own (often minimal) operating system and application workloads, and maintains the illusion of being a physical machine, both to the end users who interact with the services running in the guests, and to developers who are able to build those services using familiar abstractions, such as programming languages, libraries, and operating system features. The illusion, however, is not perfect, because ultimately the guests do share the hardware resources (CPU, memory, cache, devices) of the

Author's address: A. Randal, Department of Computer Science and Technology, University of Cambridge, 15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK; email: allison.randal@cl.cam.ac.uk.

0360-0300/2020/02-ART5 $15.00
https://doi.org/10.1145/3365199

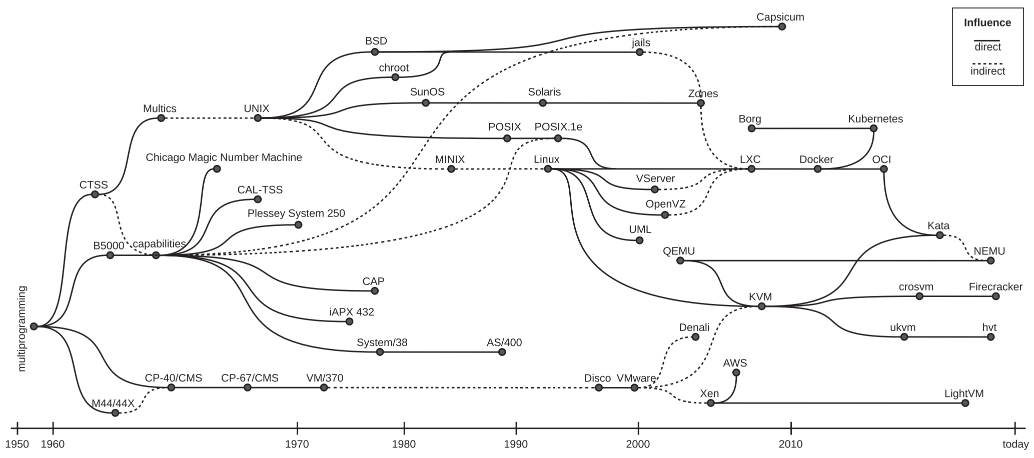

Fig. 1. The evolution of virtual machines and containers.

underlying physical host machine, and consequently also have greater access to the host's privileged software (kernel, operating system) than a physically distinct machine would have.

Ideally, multitenant environments would offer strong isolation of the guest from the host, and between guests on the same host, but reality falls short of the ideal. The approaches that various implementations have taken to isolating guests have different strengths and weaknesses. For example, containers share a kernel with the host, whereas virtual machines may run as a process in the host operating system or a module in the host kernel, so they expose different attack surfaces through different code paths in the host operating system. Fundamentally, however, all existing implementations of virtual machines and containers are leaky abstractions, exposing more of the underlying software and hardware than is necessary, useful, or desirable. New security research in 2018 delivered a further blow to the ideal of isolation in multitenant environments, demonstrating that certain hardware vulnerabilities related to speculative execution—including Spectre, Meltdown, Foreshadow, L1TF, and variants—can easily bypass the software isolation of guests.

Because multitenancy has proven to be useful and profitable for a large sector of the computing industry, it is likely that a significant percentage of computing workloads will continue to run in multitenant environments for the foreseeable future. This is not a matter of naïveté but of pragmatism: these days, the companies who provide and make use of multitenant environments are generally fully aware of the security risks, but they do so anyway because the benefits—such as flexibility, resiliency, reliability, performance, cost, or any of a dozen other factors—outweigh the risks for their particular use cases and business needs. That being the case, it is worthwhile to take a step back and examine how the past 60 years of evolution led to the current tension between secure ideals and flawed reality, and what lessons from the past might help us build more secure software and hardware for the next 60 years.

This survey is divided into sections following the evolutionary paths of the technologies behind virtual machines and containers, generally in chronological order, as illustrated in Figure 1. Section 3 explores the common origins of virtual machines and containers in the late 1950s and early 1960s, driven by the architectural shift toward multitasking and multiprocessing, and motivated by a desire to securely isolate processes, efficiently utilize shared resources, improve portability, and minimize complexity. Section 4 examines the first virtual machines in the mid-1960s to 1970s, which primarily aimed to improve resource utilization in time-sharing systems. Section 5 delves into the capability systems of the early 1960s to 1970s—the precursors of modern containers—which evolved along a parallel track to virtual machines, with similar motivations but different implementations. Section 6 outlines the resurgence of virtual machines in the late 1990s and 2000s. Section 7 traces the emergence of containers in the 2000s and 2010s. Section 8 investigates the impact of recent security research on both virtual machines and containers. Section 9 briefly looks at the relationship between virtual machines and containers and the related terms *cloud*, *serverless*, and *unikernels*.

## 2 TERMINOLOGY

For the sake of clarity, this survey consistently uses certain modern or common terms, even when discussing literature that used various other terms for the same concepts:

- *Container*: The term *container* does not have a single origin, but some early relevant examples of use are Banga et al. [25] in 1999, Lottiaux and Morin [127] in 2001, Morin et al. [145] in 2002, and Price and Tucker [164] in 2004. Early literature on containers confusingly referred to them as a kind of virtualization [45, 48, 104, 142, 164, 182], or even called them *virtual machines* [182]. As containers grew more popular, the confusion shifted to virtual machines being called *containers* [37, 220]. This survey uses the term *container* for multitenant deployment techniques involving process isolation on a shared kernel (in contrast with *virtual machine*, as defined in the following). However, in practice, the distinction between containers and virtual machines is more of a spectrum than a binary divide. Techniques common to one can be effectively applied to the other, such as using system call filtering with containers, or using seccomp sandboxing or user namespaces with virtual machines.
- *Complexity*: There are many dimensions to complexity in computing, but in the context of multitenant infrastructures, some uniquely relevant dimensions are keeping each guest, the interactions between guests, and the host's management of the guests as small and simple as possible. The implementation technique of isolation supports minimizing complexity by restricting access to internal knowledge of the guests and host, and providing well-defined interfaces to reduce the complexity of interactions between them.
- *Guest*: The term *guest* had some early usage in the 1980s for the operating system image running inside a virtual machine [147] but was not common until the early 2000s [26, 197]. This survey uses *guest* as a general term for operating system images hosted on multitenant infrastructures but occasionally distinguishes between virtual machine guests and container guests.
- *Kernel*: A variety of different terms appear in the early literature, including *supervisory program* [52], *supervisor program* [20], *control program* [15, 149, 153], *coordinating program* [153], *nucleus* [1, 43], *monitor* [209], and ultimately *kernel* around the mid-1970s [123, 161]. This survey uses the modern term *kernel*.
- *Performance*: There are many dimensions to performance in computing, but in the context of multitenant infrastructures, some uniquely relevant dimensions are the performance impact of added layers of abstraction separating the guest application workload from the host, balanced against the performance benefits of sharing resources between guests and reducing wasted resources from unused capacity. At the level of a single machine, this involves running multiple guests on the same machine at the same time, with potential for intelligent, dynamic scheduling to extract more work from the same resource pool. Across multiple machines, this involves a larger pool of shared resources, more flexibility to balance work, and options for heterogenous hardware with resource-affinity configurations (e.g., a mixture of some CPU-heavy machines and some storage-heavy machines, with workload allocation

determined by resource needs). The implementation technique of breaking down machines into smaller guests and their resources into smaller, sharable units, supports performance by allowing finer-grained and distributed control over resource management.

- *Portability*: There are many dimensions to portability in computing, but in the context of multitenant infrastructures, some uniquely relevant dimensions are developing guests in a standardized way—without any special knowledge of the environment where they will be deployed—and abstracting deployment and management across physical machines, limiting dependence on low-level hardware details. For example, a container guest can be deployed anywhere in the cluster, or a virtual machine guest can be deployed on any compute machine in the cloud. The implementation techniques of standardizing interfaces so guests are substitutable and hiding implementation and hardware details behind well-defined interfaces both support portability.
- *Process*: The early literature tended to use the terms *job* [171] or *program* [20, 52, 153], and *process* only appeared around the mid-1960s [14, 65]. This survey uses the modern term *process*. The early use of *multiprogramming* meaning "multiprocessing" was derived from the early use of *program* meaning "process."
- *Security*: There are many dimensions to security in computing, but in the context of multitenant infrastructures, some uniquely relevant dimensions are limiting access between guests, from guests to the host, and from the host to the guests. The implementation technique of isolation supports security, at both the software level and the hardware level, by reducing the likelihood of a breach and limiting the scope of damage when a breach occurs.
- *Virtual machine*: This survey uses the term *virtual machine* for multitenant deployment techniques involving the replication/emulation of real hardware architectures in software (in contrast with *container*, as defined earlier). The code responsible for managing virtual machine guests on a physical host machine is often called a *hypervisor* or *virtual machine monitor*, both derived from two early terms for the kernel, *supervisor* and *monitor*. In many early implementations of virtual machines, the host kernel managed both guests and ordinary processes.

## 3 COMMON ORIGINS

The origins of both virtual machines and containers can be traced to a fundamental shift in hardware and software architectures toward the late 1950s. The hardware of the time introduced the concept of *multiprogramming*, which included both basic multitasking in the form of simple context-switching and basic multiprocessing in the form of dedicated I/O processors and multiple CPUs. Codd [51] attributed the earliest known use of the term *multiprogramming* to Rochester [171] in 1955, describing the ability of an IBM 705 system to interrupt an I/O process (tape read), run a process (calculation) on the data found, and then return to the I/O process. The concept of multiprogramming evolved over the remainder of the decade through work on the EDSAC [211], UNIVAC LARC [70], STRETCH (IBM 7030) [52, 69], TX-2 [77], and an influential and comprehensive review by Gill [82]. Key trade-offs discussed in the literature on multiprogramming—around security, performance, portability, and complexity—continue to echo through modern literature on virtual machines and containers.

### 3.1 Security

Multiprogramming increased the complexity of the system software—due to simultaneous and interleaved processes interacting with other processes and shared hardware resources—and also increased the consequences of misbehaving system software—since any process had the potential to disrupt any other process on the same machine. Codd et al. [52] discussed secure isolation as a requirement for "noninterference" between processes regarding errors, in the core design principles for STRETCH. Codd [51] later expanded on the requirement as a need to prevent processes from making "accidental or fraudulent" changes to another process. Buzen and Gagliardi [43] called out the risk of one process modifying memory allocated to other processes or privileged system operations.

In response to the increase in complexity and risk, system software of the time introduced a familiar form of isolation, granting a small privileged kernel of system software unrestricted access to all hardware resources and running processes, as well as responsibility for potentially disruptive operations such as memory and storage allocation, process scheduling, and interrupt handling while restricting access to such features from any software outside the kernel. Codd et al. [52] described the structure and function of the STRETCH kernel in detail, including concurrency, interrupts, memory protection, and time limits (an early form of resource usage control). Amdahl et al. [20] touched on the separation of the kernel in the IBM System/360, including appendices of relevant opcodes and protected storage locations. Opler and Baird [153] weighed trade-offs around having the kernel take responsibility for coordinating the parallel operation of processes and judged the approach to have potential to improve portability of programs not written for parallel operation, as well as potential to minimize complexity for programmers who would no longer be responsible to manually coordinate the parallel operation of each program.

## 3.2 Performance

One of the fundamental goals of adding multiprogramming to hardware and operating systems in the late 1950s was to improve performance through more efficient utilization of available resources by sharing them across parallel processes. Codd et al. [52] described performance as a requirement for "noninterference" between processes regarding "undue delay." Opler and Baird [153] explored the trade-offs between the performance advantages of increasing utilization through multiprocessing, versus the increased complexity of developing for such systems. Codd [49, 50] published two further papers in 1960 about performance considerations for process scheduling algorithms in multiprogramming. Amdahl et al. [20, p. 89] explored the trade-offs between performance and portability in the architecture design of the IBM System/360. Dennis [64, p. 590] noted the performance advantages of dynamic memory allocation for multiprogramming.

## 3.3 Portability

In the 1950s, it was common for specialized system software to be developed for each new model of hardware, requiring programs to be rewritten to run on even closely related machines. As the system software and programs grew larger and more complex, the porting effort grew more costly, motivating a desire for programs to be portable across different machines. Codd et al. [52] discussed portability as a requirement for "independence of preparation" and "flexible allocation of space and time." Amdahl et al. [20, p. 97] emphasized portability as one of the primary design goals of the IBM System/360, specifically allowing machine-language programs to run unmodified across six different hardware models, with a variety of different configurations of peripheral devices. Buzen and Gagliardi [43] noted that the introduction of a privileged kernel compounded the problem of portability, since a program might have to be rewritten to run on two different kernels, even when the underlying hardware was compatible or completely identical.

## 3.4 Minimizing Complexity

Another early realization after the introduction of multiprogramming was that it was unreasonable to expect the developer of each process to directly manage all of the complexity of interacting with every other process running on the machine, so the privileged kernel approach had the advantage of allowing processes to maintain a more minimal focus on their own internals. Codd et al. [52] described minimizing complexity as a requirement for "minimum information from programmer." Nearly a decade before Rushby [173] first wrote about the idea of a Trusted Computing Base, Buzen and Gagliardi [43, p. 291] argued for minimizing complexity within the privileged kernel, noting that such separation was effective when the privileged code base was kept small, so it could be maintained in a relatively stable state, with limited changes over time, by a few expert developers.

## 4 EARLY VIRTUAL MACHINES

The early work on virtual machines grew directly out of the work on multiprogramming, continuing the goal of safely sharing the resources of a physical machine across multiple processes. Initially, the idea was no more than a refinement on memory protection between processes, but it expanded into a much bigger idea: that small isolated bundles of shared resources from the host machine could present the illusion of being a physical machine running a full operating system.

### 4.1 M44/44X

In 1964, Nelson [149] published an internal research report at IBM outlining plans for an experimental machine based on the IBM 7044, called the *M44*. The project built on earlier work in multiprogramming, improving process isolation and scheduling in the privileged kernel with an early form of virtual memory. They called the memory mapped for a particular process a *virtual machine* [149, p. 14]. The 44X part of the name stood for the virtual machines (also based on the IBM 7044) running on top of the M44 host machine.

Nelson [149, pp. 4–6] identified the performance advantages of dynamically allocated shared resources (especially memory and CPU) as one of the primary motivators for the M44/44X experiments. Portability was another central consideration, allowing software to run unmodified across single process, multiprocess, and debugging contexts [149, pp. 9–10].

The M44/44X lacked almost all of the features we would associate with virtual machines today, but it played an important, although largely forgotten, part in the history of virtual machines. Denning [63] reflected that the M44/44X was central to significant theoretical and experimental advances in memory research around paging, segmentation, and virtual memory in the 1960s.

### 4.2 Cambridge Monitor System

The IBM System/360 was explicitly designed for portability of software across different models and different hardware configurations [20]. In the mid-1960s, IBM's *Control Program-40 Cambridge Monitor System* (CP-40/CMS) project running on a modified IBM System/360 (model 40) took the idea a few steps further—initially calling the work a *pseudo machine* but later adopting the term *virtual machine* [61, p. 485]. The CP-40/CMS and later CP-67/CMS[1] projects improved on earlier approaches to portability, making it possible for software written for a bare metal machine to run unmodified in a virtual machine, which could simulate the appearance of various different hardware configurations [15, pp. 1–2]. It also improved isolation by introducing privilege separation for interrupts [15, pp. 6–7], paged memory within virtual machine guests [43, 155], and simulated devices [1, 43]. IBM's work on the CP-40/CMS focused on improving performance through efficient utilization of shared memory [15, pp. 3–5] and explictly did not target efficient utilization of CPU through sharing [15, p. 1]. Kogut [112] developed a variant of CP-67/CMS to improve performance through dynamic allocation of storage (physical disk) to virtual machines.

[1]For the IBM System/360 model 67.

### 4.3 VM/370

IBM's VM/370 running on the System/370 hardware followed in the early 1970s and included virtual memory hardware [61, p. 485]. Madnick and Donovan [130, p. 214] estimated the overhead of the VM/370 at 10% to 15% but deemed the performance trade-off to be worthwhile from a security perspective. Goldberg [85, pp. 39–40] identified the source of overhead as primarily: maintaining state for virtual processors, trapping and emulating privileged instructions, and memory address translation for virtual machine guests (especially when paging was supported in the guests). In retrospect, Creasy [61] noted that efficient execution was never a primary goal of IBM's work on the CP-40, CP-67, or VM/370 (p. 487), and the focus was instead on efficient utilization of available resources (p. 484).

### 4.4 Trade-Offs

In their formal requirements for virtual machines in the mid-1970s, Popek and Goldberg [162, p. 413] stated that ideally virtual machines should "show at worst only minor decreases in speed" compared to running on bare metal. In 2017, Bugnion et al. [41] explained Popek and Goldberg's requirements in modern terms, exploring the performance impact for hardware architectures that do not fully meet the requirements.

Buzen and Gagliardi [43, p. 291], Madnick and Donovan [130, p. 212], Goldberg [84, p. 75], and Creasy [61, p. 486] all observed that the portability offered by virtual machines was also an advantage for development purposes, since it allowed development and testing of multiple different versions of the kernel/operating systems—and programs targeting those kernels/operating systems—in multiple different virtual hardware configurations, on the same physical machine at the same time.

Buzen and Gagliardi [43] considered one of the key advantages of the virtual machine approach to be that "virtual machine monitors typically do not require a large amount of code or a high degree of logical complexity." Popek and Kline [161, p. 294] discussed the advantage of virtual machines being smaller and less complex than a kernel and complete operating system, improving their potential to be secure. Goldberg [85, p. 39] suggested minimizing complexity as a way to improve performance: selectively disabling more expensive features (e.g., memory paging in guests) for virtual machines that would not use the features. Creasy [61, p. 488] discussed the advantages of minimizing interdependencies between virtual machines, giving preference to standard interfaces on the host machine.

A frequently cited group of papers in the early 1970s, by Lauer and Snow [118], Lauer and Wyeth [119], and Srodawa and Bates [185], suggested that virtual machines offered a sufficient level of isolation that it was no longer necessary to maintain a privilege-separated kernel in the host operating system. However, by that point in time, the concept of a privileged kernel was well enough established that the idea of eliminating it was unlikely to be widely accepted. Buzen and Gagliardi [43, p. 297] observed that the proposal depended heavily on the ability of the virtual machine implementation to handle all virtual memory mapping directly, but since the papers failed to take memory segmentation into account, the approach could not be implemented as initially proposed.

### 4.5 Decline

As companies like DEC, Honeywell, HP, Intel, and Xerox introduced smaller hardware to the market in the 1970s, they did not include hardware support for features such as virtual memory and the ability to trap all sensitive instructions, which made it challenging to implement strong isolation using virtual machine techniques on such hardware [66, 78]. Creasy [61, p. 484] observed in the early 1980s that the advent of the personal computer decreased interest in the early forms of virtual machines—which were largely developed for the purpose of isolating users in time-sharing systems on mainframes—but he recognized potential for virtual machines to serve "the future's network of personal computers."[2]

## 5 EARLY CAPABILITIES

The origin of containers is often attributed [31, 54, 114, 121, 166] to the addition of the `chroot` system call in the seventh edition of UNIX released by Bell Labs in 1979 [108]. The simple form of filesystem namespace isolation that `chroot` provides was certainly one influence on the development of containers, although it lacked any concept of isolation for process namespaces [105, 165]. However, containers are not a single technology; they are a collection of technologies combined to provide secure isolation, including namespaces, cgroups, seccomp, and capabilities. Combe et al. [54], Jian and Chen [102], Kovács [114], Priedhorsky and Randles [165], and Raho et al. [166] describe how these different technologies combine to provide secure isolation for containers. It is more accurate to attribute the origin of containers to the earliest of these technologies—capabilities—that began decades before `chroot` and several years before the first work on virtual machines. Like containers, capabilities took the approach of building secure isolation into the hardware and the operating system, without virtualization.

### 5.1 Descriptors

In the early 1960s, inspired by the need to isolate processes, the Burroughs B5000 hardware architecture introduced an improvement to memory protection called *descriptors*, which flagged whether a particular memory segment held code or data, and protected the system by ensuring it could only execute code (and not data), and could only access data appropriately (a single element scalar, or bounds-checked array) [120, 136]. A process on the B5000 could only access its own code and data segments through a private Program Reference Table, which held the descriptors for the process [120, p. 23]. A descriptor also flagged whether a segment was actively in main memory or needed to be loaded from drum [120, p. 24].

### 5.2 Dennis and Van Horn

In the mid-1960s, Dennis and Van Horn [65] introduced the term *capability* in theoretical work directly inspired by both the Burroughs B5000 and MIT's Compatible Time-Sharing System (CTSS) [65, p. 154]. Like the B5000 descriptors, capabilities defined the set of memory segments a process was permitted to read, write, or execute [120, p. 42]. These early capabilities introduced several important refinements: a process executed within a protected *domain* with an associated capability list; multiple processes could share the same capability list; and a process could `FORK` a parallel process with the same capabilities (but no greater), or create a subprocess with a subset of its own capabilities (but no greater) [120, pp. 42–44]. These theoretical capabilities also had a concept of ownership (by a process or a user) [120, p. 42] and of persistent data "directories" (but not files) that survived beyond the execution of a process and could be private to a user or accessible to any user [120, pp. 44–45].

Soon after Dennis and Van Horn published their theoretical capabilities, Ackerman and Plummer [14] implemented some aspects of capabilities relating to resource control on a modified PDP-1 at MIT and added a file capability in addition to the directory capability—a precursor to filesystem namespaces.

[2]It was a reasonable prediction for the time: HTTP was introduced much later in the 1980s, but the RFC for the Internet Protocol (IP) [163] was published in the same month as Creasy's article, and TCP had already been around since the mid-1970s.

### 5.3 Chicago Magic Number Machine

In 1967, the University of Chicago launched the first attempt at designing and building a general-purpose hardware and software capability system, which they later called the *Chicago Magic Number Machine*[3] [73, 74]. The Chicago machine pushed the concept of separation between capabilities and data further, to protect against users altering the capabilities that limited their access to memory on the system [120, pp. 49–50]. The machine had a set of physical registers for capabilities, which were distinct from the usual set of registers for data. It also flagged whether each memory segment stored capabilities or data, and prevented processes from performing data operations like reading or writing on capability segments or capability registers. Inter-process communication also sent both a capability segment and a data segment [120, p. 51].

The University of Chicago project ran out of funding and was never completed, but it inspired subsequent work on CAL-TSS [120, p. 49].

### 5.4 CAL-TSS

In 1968, the University of California at Berkeley launched the CAL-TSS project [120, pp. 52–57], which aimed to produce a general-purpose capability-based operating system, to run on a Control Data Corporation 6400 model (RISC architecture) mainframe machine, without any special customization to the hardware. Like previous implementations, CAL-TSS confined a process to a domain, restricting access to hardware registers, memory, executable code, system calls to the kernel, and inter-process communication. The project introduced a concept of unique and non-reusable identifiers for objects, to protect against reuse of dangling pointers to access and modify memory that has been reallocated after being freed.

The CAL-TSS project encountered difficulties implementing the operating system as designed and was terminated in 1971. Levy [120, p. 57] identified the memory management features of the CDC 6400 as a particularly troublesome obstacle to the implementation. In postmortem analysis, Sturgis [186] and Lampson and Sturgis [116] reflected that CAL-TSS ended up being large, overly complex, and slow, and attributed this primarily to a poor match between the hardware they selected and the design of mapped address spaces, and also to their design choice of distributing privileged code for manipulating global system data across individual processes rather than consolidating it in a privileged kernel.

### 5.5 Plessey System 250

In the early 1970s, the Plessey System 250 [72] was a commercially successful real-time multiprocessing telephone-switch controller. It implemented capabilities for memory protection and process isolation [120, p. 65], and expanded capabilities into the I/O system [120, p. 77].

### 5.6 Provably Secure Operating System

Also in the early 1970s, the Stanford Research Institute began a project to explore the potential of formal proofs applied to a capability-based operating system design, which they called the *Provably Secure Operating System* (PSOS) [150]. The design was completed in 1980 but never fully formally proven and never implemented [151].

### 5.7 CAP

In the late 1970s, the University of Cambridge's CAP machine [148, 210] successfully implemented capabilities as general-purpose hardware combined with a complementary operating system. The

[3]The unusual name was emblematic of the decade, from Ken Kesey's "Magic Bus" to the Beatles' "Magical Mystery Tour." At the level of physical memory, capabilities are effectively a "magic" number.

CAP introduced a refinement replacing the privileged kernel with an ordinary process, so the special control the "root" process had over the entire system was really just the normal ability of any process to create subprocesses and grant a subset of its own capabilities to those subprocesses [120, pp. 80–81].

## 5.8 Object Systems

Several software offshoots of the early capability systems generalized the idea by treating processes and shared resources as typed objects with associated capabilities, including Carnegie-Mellon's Hydra [217, 218], StarOS [103], and Gnosis later renamed as KeyKOS [92].

## 5.9 IBM System/38

In 1978, IBM announced plans for a capability-based hardware architecture, the System/38, which they shipped in 1980 [120, p. 137]. Berstis [32] characterized the primary goal of the System/38 as improving memory protection without sacrificing performance. Houdek et al. [96] described the implementation of capabilities as protected pointers in detail. The System/38 introduced a concept of user profiles associated with protected process domains [32, pp. 249–250], which were vaguely reminiscent of modern user namespaces, although implemented differently. User profiles allowed for revocation of capabilities but at the cost of significantly increased complexity in the implementation [120, pp. 155–156].

The System/38 was succeeded by the AS/400 in the late 1980s, which removed capability-based addressing [183, p. 119]. The AS/400 later adopted the concept of logical partitioning from the IBM System/370 [176, pp. 1–2], to divide the physical resources of the host machine between multiple guests at the hardware level[4] [183, pp. 240, 328].

## 5.10 Intel iAPX 432

In 1975, Intel began designing the iAPX 432 [2] capability-based hardware architecture, which they originally intended to be their next-generation, market-leading CPU, replacing the 8080 [137, p. 79]. The project finally shipped in 1981, but it was significantly delayed and significantly over budget [137, p. 79].

Mazor [137, p. 75] recorded that performance was not considered as a goal in the design of the iAPX 432. Hansen et al. [91] measured the performance of the iAPX 432 against the Intel 8086, Motorola 68000, and the VAX-11/780 in 1982, with results as poor as 95 times slower on some benchmarks. Norton [152, p. 27] assessed the poor performance and unoptimized compiler offered by the iAPX 432 as the leading cause of its commercial failure. Levy [120, p. 186] blamed the commercial failure on both poor performance and overhyped marketing.

In a move that Mazor [137] described as "a crash program . . . to save Intel's market share" (p. 75), Intel launched a parallel project to develop the 8086 architecture (the first in a long line of x86 CPUs), which became Intel's leading product line by default rather than by design (p. 79).[5]

## 5.11 Trade-Offs

The early capability systems in the 1960s and 1970s sacrificed performance for the sake of security, although Levy speculated in the mid-1980s that this was partly due to "hardware poorly matched

[4]Unlike virtual machines, capabilities, or containers, which divide physical resources at the software level.

[5]In hindsight, the commercial failure of the iAPX 432 probably influenced Intel's single-minded focus on performance and disinterest in memory protection techniques in the decades that followed, which ultimately contributed to the vulnerabilities discussed in Section 8.

to the task" [120, p. 205]. Wilkes [209, pp. 49–59] contrasted the memory protection features of capabilities with other systems of the time, including detailed descriptions of hardware implementations.

Levy [120, p. 205] also observed that the early capability systems significantly increased complexity for the sake of security. Patterson and Séquin [157] and Patterson and Ditzel [156] judged this sacrifice as a major reason the capability machines were surpassed by simpler architectures, such as RISC.

Kirk McKusick recalled that the primary reason Bill Joy ported `chroot` from UNIX into BSD in 1982 was for portability, so he could build different versions of the system in an isolated build directory [105, p. 11].

### 5.12 Decline

As with virtual machines, interest in the early capability systems sharply declined in the 1980s, influenced by several independent factors. Several early attempts to implement capabilities were terminated uncompleted—notably the Chicago Magic Number Machine, CAL-TSS, and the PSOS—contributing to a reputation that capability systems were difficult to implement and perhaps overly ambitious, despite the successful implementations that followed. The commercial failure of Intel's iAPX 432 raised further doubts on the feasibility of capability-based architectures. In 2003, Neumann and Feiertag [151, p. 6] looked back on the early capability systems, expressing disappointment that "the demand for meaningfully secure systems has remained surprisingly small until recently."

Perhaps the most significant factor in the decline of capabilities was the rise of the general-purpose operating system, which was a third important technology that evolved from multiprogramming. MIT's CTSS [55, 209] laid the foundation for Multics [56], which later inspired UNIX [168] and its robust mutation, the Berkeley Software Distribution (BSD)[6] [138, 139]. Saltzer and Schroeder [174, p. 1294] contrasted capabilities with the access control list models adopted by Multics and its descendants, calling out revocation of access as one major area where capabilities fell short.

Although none of the early capability systems remain in use today, they have not been entirely forgotten. In 2003, Miller et al. [143] reviewed capability systems from a historical perspective, addressing common misconceptions about capabilities related to revocation, confinement, and equivalence to access control lists. Section 7 traces the evolution of a feature called *capabilities* in the modern Linux Kernel. FreeBSD took a different approach for the feature it calls capabilities and integrated the Capsicum framework [140, p. 30], which was more directly derived from the classic capability systems [21, 199]. In 2012, the CHERI project [200, 202, 203, 215] expanded on the ideas of the Capsicum framework, pushing its capability model down into an RISC-based hardware architecture. Since 2016, Google has been exploring a revival of capability systems with the Fuchsia operating system and Zircon microkernel [87]. In a 2018 plenary session about Spectre/Meltdown, Hennessy [94] pointed to future potential for capabilities, reflecting that the early capability systems "probably weren't the right match for what software designers thought they needed and they were too inefficient at the time" but suggested "those are all things we know how to fix now . . . so it's time, I think, to begin re-examining some of those more sophisticated [protection] mechanisms and see if they'll work."

[6]One noteworthy connection between these factors is Robert Fabry, who worked on the Chicago Magic Number Machine in the 1960s [73, 74] while working on a Ph.D. at the University of Chicago [75], and was also the catalyst for Berkeley's interest in UNIX and substantial investment in the BSD project, while he was a professor at Berkeley in the 1970s [138].

## 6 MODERN VIRTUAL MACHINES

Virtual machines still existed in the 1980s and 1990s but garnered only a bare minimum of activity and interest. IBM's line of VM products, descended from VM/370, continued to have a small but loyal following [194]. DOS, OS/2, and Windows all offered a limited form of DOS virtual machines during that time, although it might be more fair to categorize those as emulation. The rise of programming languages like Smalltalk and Java re-purposing the term *virtual machine*—to refer to an abstraction layer of a language runtime rather than a software replication of a real hardware architecture—may be indicative of how dead the original concept of virtual machines was in that period.

After nearly two decades, the late 1990s brought a resurgence of interest in virtual machines but for a new purpose adapted to the technology of the time.

### 6.1 Disco

In 1997, the Disco research project at Stanford University explored reviving virtual machines as an approach to making efficient use of hardware with multiple CPUs (on the order of "tens to hundreds"), and included a lightweight library operating system for guests (SPLASHOS) as an option, in addition to supporting commodity operating systems as guests. Bugnion et al. [39] cited portability (rather than security or performance) as the primary motivation of the Disco project, which proposed virtual machines as a potential way to allow commodity operating systems (Unix, Windows NT, and Linux) to run on NUMA architectures without extensive modifications.

### 6.2 VMware

A year later, the team behind Disco founded VMware to continue their work, and released a workstation product in 1999 [40], quickly followed by two server products (GSX and ESX) in 2001 [18, 175, 197]. VMware faced a challenge in virtualizing the x86 architectures of the time, because the hardware did not support traditional virtualization techniques—specifically the architecture contained some sensitive instructions that were not also privileged—so a virtual machine monitor could not rely on trapping protection exceptions as the sole means of identifying when to execute emulated instructions as a safe replacement, since some potentially harmful instructions would never be trapped [170, p. 131].[7] To work around this limitation, VMware combined the trap-and-execute technique with a dynamic binary translation technique [40, p. 12:3], which was faster than full emulation but still allowed the guest operating system to run unmodified [40, pp.12:29–12:36].

### 6.3 Denali

The Denali project at the University of Washington in 2002 [207] introduced the term *paravirtualization*,[8] another work-around for the lack of hardware virtualization support in x86, which involved altering the instruction set in the virtualized hardware architecture and then porting the guest operating system to run on the altered instruction set [206].

### 6.4 Xen

The Xen project at the University of Cambridge in 2003 [26] also used paravirtualization techniques and modified guest operating systems but emphasized the importance of preserving the application binary interface (ABI) within the guests so that guest applications could run unmodified. Xen's greatest technical contribution may have been its approach to precise accounting for resource

[7]Popek and Goldberg [162] classically defined such machines as unvirtualizable.
[8]The term was new, but the technique had roots stretching back to IBM's VM/370 [61, 85].

usage, with the explicit intention to individually bill tenants sharing physical machines [26, p. 176], which was a relatively radical idea at the time[9] and directly led to the creation of Amazon's Elastic Compute Cloud (EC2) a couple of years later [28].[10]

Chisnall [47] provided a detailed account of Xen's architecture and design goals. Xen's approach to the problem of untrapped x86 privileged instructions was to substitute a set of *hypercalls* for unsafe system calls [47, pp. 10–13]. Smith and Nair [181, p. 422] highlighted that Xen was able to run unmodified application binaries within the guest, because it ran the guest in ring 1 of the IA-32 privilege levels and the hypervisor in ring 0, so all privileged instructions were filtered through the hypervisor.

### 6.5 x86 Hardware Virtualization Extensions

In 2000, Robin and Irvine [170] analyzed the limitations of the x86 architecture as a host for virtual machine implementations, with reference to the earlier work of Goldberg [83] on the architectural features required to support virtual machines. In the mid-2000s, in response to the growing success of virtual machines, and the challenges of implementing them on x86 hardware, Intel and AMD both added hardware support for virtualization in the form of a less privileged execution mode to execute code for the virtual machine guest directly but selectively trap sensitive instructions, eliminating the need for binary translation or paravirtualization. Rosenblum and Garfinkel [172] discussed the motivations behind the added hardware support for virtualization in x86, before the changes were released. Pearce et al. [158, p. 7] contrasted binary translation, paravirtualization, and the features x86 added for hardware-assisted virtualization, clarifying the x86 virtualization extensions were not full virtualization. Adams and Agesen [16] recounted the difficulties VMware encountered while integrating the x86 hardware virtualization extensions and concluded that the new features offered no performance advantage over binary translation.

In 2007, the KVM subsystem for the Linux Kernel provided an API for accessing the x86 hardware virtualization extensions [110]. Since KVM was only a Kernel subsystem, the developers released a fork of QEMU[11] as the userspace counterpart of KVM, so the combination of QEMU+KVM provided a full virtual machine implementation, including virtual devices [198, pp.128–129]. Eventually, KVM support was merged into mainline QEMU [122].

### 6.6 Hyper-V

In 2008, Microsoft released a beta of Hyper-V [107] for Windows Server. It was built on top of the x86 hardware virtualization extensions and for some virtual devices offered a choice between slower emulation and faster paravirtualization if the guest operating system installed the "Enlightened I/O" extensions. Like Xen's Dom0, Hyper-V granted special privileges to one guest, called the *parent partition*, which hosted the virtual devices and handled requests from the other guests.

In 2010, Bolte et al. [35] incorporated support for Hyper-V into `libvirt`, so it could be managed through a standardized interface, together with Xen, QEMU+KVM, and VMware ESX.

### 6.7 Trade-Offs

Denali and Xen both used paravirtualization techniques, sacrificing portability to gain performance, but their goals for scale were completely different: Denali considered 10,000 virtual

[9]Partially inspired by earlier work, involving some of the same authors, on resource management in the Nemesis operating system [27].
[10]The EC2 beta was launched in 2006, but when I presented at the Amazon Developers Conference in 2005, they were already working on it.
[11]Which was previously only an emulator [29].

machines[12] to be a good result [208]—achieved through a combination of lightweight guests and a minimal host—whereas Xen argued that 100 virtual machines running full operating systems[13] was a more reasonable target [26, p. 165,175]. To some extent, Denali was more in line with modern container implementations than with the virtual machine implementations of its day. Xen has shifted their estimation of required scale upward over the years but still exhibits a tolerance for unnecessary performance degradation. For example, Manco et al. [131] demonstrated that a few small internal changes to the way Xen stores metadata and creates virtual devices improved virtual machine instantiation time by an order of magnitude—a result 50 to 200 times faster than Docker's container instantiation—however, those patches are unlikely to ever make it into mainline Xen.

Xen and KVM have a reputation for sacrificing performance to gain security; however, several independent lines of research have raised questions as to whether those security gains are real or imagined. Perez-Botero et al. [159] analyzed security vulnerabilities in Xen and KVM between 2008 and 2012, categorizing them by source, vector, and target, and observed that the most common vector of attack was device emulation (Xen 34%, KVM 40%), the majority were triggered from within the virtual machine guest (Xen 71%, KVM 66%), and the majority successfully targeted the hypervisor's Ring −1 privileges or slightly less privileged control over Dom0 or the host operating system (Xen 80%, KVM 76%). Chandramouli et al. [46] built on the work of Perez-Botero et al. [159], moving toward a more general framework for forensic analysis of vulnerabilities in virtual machine implementations. Ishiguro and Kono [101] evaluated vulnerabilities in Xen and KVM related to instruction emulation between 2009 and 2017. They demonstrated that a prototype "instruction firewall" on KVM—which denies emulation of all instructions except the small subset deemed legitimate in the current execution context—could have defended against the known instruction emulation vulnerabilities; however, the patches are unlikely to ever make it into mainline KVM.

Szefer et al. [191] demonstrated in the NoHype implementation (based on Xen) that eliminating the hypervisor and running virtual machines with more direct access to the hardware improved security by reducing the attack surface and removing virtual machine exit events as potential attack vectors. However, the approach involved a performance trade-off in resource utilization that was not viable for most real deployments: it pre-allocated processor cores, memory, and I/O devices dedicated to specific virtual machines rather than allowing for oversubscription and dynamic allocation in response to load.

One persistent argument in favor of virtual machines has been that virtual machine implementations have fewer lines of code than a kernel or host operating system, and are therefore easier to code review and secure [39, 81, 131, 158, 178], which is the classic trade-off of minimizing complexity to gain security. However, less code offers only a vague potential for security, and even that potential becomes questionable as modern virtual machine implementations have grown larger and more complex [37, 53, 158, 214].

Recent work on virtual machines—such as `ukvm` [212], LightVM [131], and Kata Containers (formerly Intel Clear Containers) [5]—has shifted back toward an emphasis on improving performance. However, this work appears to be founded on the assumption that the virtual machine implementations under discussion are adequately secure and need only improve performance, which is a dubious assumption at best.

Two notable departures from this complacent attitude to security are Google's crosvm [86] and Amazon's Firecracker [19], which aim to improve both performance and security, by replacing QEMU with a radically smaller and simpler userspace component for KVM, and by choosing Rust

[12]On a 1.7-GHz Pentium 4 with 1 GB of RAM.
[13]On a 2.4-GHz dual-core Xeon with 2 GB of RAM.

as the implementation language for memory safety.[14] Firecracker started as a fork of crosvm, but the two projects are collaborating on generalizing the divergence into a set of Rust libraries they can share.

### 6.8 Decline

Toward the end of the 2000s, the enthusiasm for virtual machines gave way to a growing skepticism. Garfinkel et al. [80] demonstrated that virtual machine environments could reliably be detected on close inspection, reviving the long-running tension between the ideals of strong isolation in virtual machines, and the reality of actual implementations. Buzen and Gagliardi [43] commented on the ideals in the early 1970s, stating "Since a privileged software nucleus has, in principle, no way of determining whether it is running on a virtual or a real machine, it has no way of spying on or altering any other virtual machine that may be coexisting with it in the same system," but in the same work they acknowledged, "In practice no virtual machine is completely equivalent to its real machine counterpart."

In 2010, Bratus et al. [37] criticized the disproportionate focus of systems security research on virtual machines and the resulting neglect of other potentially superior approaches to system security. Vasudevan et al. [195] outlined a set of requirements for protecting the integrity of virtual machines implemented on x86 with hardware virtualization support and evaluated all existing implementations as "unsuitable for use with highly sensitive applications" (p. 141). Colp et al. [53] observed that multitenant environments presented new risks for virtual machine implementations, because they required stronger isolation between guests sharing the same host than was necessary when a single tenant owned the entire physical machine.

Virtual machines such as Xen, QEMU+KVM, Hyper-V, and VMware are still in active use today, but in recent years they have entirely ceded their reputation as the "hot new thing" to containers.

## 7 MODERN CONTAINERS

The collection of technologies that make up modern container implementations started coming together years before anyone used the term *container*. The two decade span surrounding the development of containers corresponded to a major shift in the way information about technological advances was broadcast and consumed. Exploring the socio-economic factors driving this shift is outside the scope of this survey; however, it is worth noting that the academic literature on more recent projects such as Docker and Kubernetes is largely written by outsiders providing external commentary rather than by the primary developers of the technologies. As a result, recent academic publications on containers tend to lack the depth of perspective and insight that was common to earlier publications on virtual machines, capabilities, and security in the Linux Kernel. The dialog driving innovation and improvements to the technology has not disappeared, but it has moved away from the academic literature and into other communication channels.

### 7.1 POSIX Capabilities

In the mid-1990s, the security working group of the POSIX standards project began drafting an extension to the POSIX.1 standard, called *POSIX 1003.1e* [3, 71, 90], which added a feature called *capabilities*. The implementation details of POSIX capabilities were entirely different than the early capability systems [201, p. 97] but had similarities on a conceptual level: POSIX capabilities were a set of flags associated with a process or file, which determined whether a process was permitted

[14]The memory safety features of Rust do not address the security vulnerabilities discussed in Section 8 but can eliminate another common class of memory access vulnerabilities, such as buffer overflows/underflows and use-after-free. Szekeres et al. [192] provide a systematic account of such vulnerabilities and their impact in the C/C++ programming languages.

to perform certain actions, a process could exec a subprocess with a subset of its own capabilities, and the specification attempted to support the principle of least privilege [3]. However, the POSIX capabilities did not adopt the concepts of small access domains and no-privilege defaults, which were crucial elements of secure isolation in the early capability systems [62]. The POSIX.1e draft was withdrawn from the process in 1998 and never formally adopted as a standard [90, p. 259], but it formed the basis of the capabilities feature added to the Linux Kernel in 1999 (release 2.2) [4, 132].

## 7.2 Namespaces and Resource Controls

A second important strand in the evolution of modern container implementations was the isolation of processes via namespaces and resource usage controls. In 2000, FreeBSD added Jails [105], which isolated filesystem namespaces (using chroot) but also isolated processes and network resources in such a way that a process might be granted root privileges inside the jail but blocked from performing operations that would affect anything outside the jail. In 2001, Linux VServer [182] patched the Linux Kernel to add resource usage limits and isolation for filesystems, network addresses, and memory. Around the same time, Virtuozzo (later released as OpenVZ) [98, 135] also patched the Linux Kernel to add resource usage limits and isolation for filesystems, processes, users, devices, and interprocess communication (IPC). In 2003, Nagar et al. [146] proposed a framework for resource usage control and metering called *Class-Based Kernel Resource Management* (CKRM) and later released it as a set of patches to the Linux Kernel.

In 2002, the Linux Kernel (release 2.4.19) introduced a filesystem namespaces feature [109].[15] In 2006, Biederman [33] proposed expanding the idea of namespace isolation in the Linux Kernel beyond the filesystem to process IDs, IPC, the network stack, and user IDs. The Kernel developers accepted the idea, and the patches to implement the features landed in the Kernel between 2006 and 2013 (releases 2.6.19 to 3.8) [109]. The last set of patches to be completed was user namespaces, which allow an unprivileged user to create a namespace and grant a process full privileges for operations inside that namespace while granting it no privileges for operations outside that namespace [11]. The way user namespaces are nested bears a resemblance to those of the capabilities of Dennis and Van Horn [65], where processes created more restricted subprocesses.

In 2004, Solaris added Zones [164] (sometimes also called *Solaris Containers*), which isolated processes into groups that could only observe or signal other processes in the same group, associated each zone with an isolated filesystem namespace, and set limits for shared resource consumption (initially only CPU). Between 2006 and 2007, Rohit Seth and Paul Menage worked on a patch for the Linux Kernel for a feature they called *process containers* [58]—later renamed to *cgroups* for "control groups"—which provided resource limiting, prioritization, accounting,[16] and control features for processes.

## 7.3 Access Control and System Call Filtering

A third set of relevant features in the Linux Kernel evolved around secure isolation of processes through restricted access to system calls. In 2000, Cowan et al. [60] released SubDomain, a Linux Kernel module that added access control checks to a limited set of system calls related to executing processes. In 2001, Loscocco and Smalley [126] published an architectural description of SELinux, which implemented mandatory access control (MAC) for the Linux Kernel. The access control architecture of SELinux was received positively, but the implementation was rejected for being

[15]Partially inspired by the namespaces feature of Plan 9 [160] from Bell Labs.
[16]Similar in idea, although not in implementation, to Xen's resource usage accounting.

too tightly coupled with the kernel. So, in 2002, Wright et al. [216] proposed the Linux Security Module (LSM) framework as a more general approach to extensible security in the Linux Kernel, which made it possible for security policies to be loaded as Kernel modules. LSM is not an access control mechanism, but it provides a set of hooks where other security extensions such as SELinux or AppArmor can insert access control checks. LSM and a modified version of SELinux based on LSM were both merged into the mainline Linux Kernel in 2003. In 2004 to 2005, SubDomain was rewritten to use LSM and rebranded under the name AppArmor.

In 2005, Arcangeli [22] released a set of patches to the Linux Kernel called *seccomp* for "secure computing," which restricted a process so that it could only run an extremely limited set of system calls to exit/return or interact with already open filehandles and terminated a process attempting to run any other system calls. The patches were merged into the mainline Kernel later that year. However, the features of the original seccomp were inadequate and rarely used, and over the years multiple proposals to improve seccomp were unsuccessful. Then, in 2012, Drewry [68] extended seccomp to allow filters for system calls to be dynamically defined using Berkeley Packet Filter (BPF) rules, which provided enough flexibility to make seccomp useful as an isolation technique. In 2013, Krude and Meyer [115] implemented a framework for isolating untrusted workloads on multitenant infrastructures using seccomp system call filter policies written in BPF.

## 7.4 Cluster Management

A fourth relevant strand of technology evolved around resource sharing in large-scale cluster management. In 2001, Lottiaux and Morin [127] used the term *container* for a form of shared, distributed memory that provided the illusion that multiple nodes in an SMP cluster were sharing kernel resources, including memory, disk, and network. In 2002, the Zap project [154] used the term *pod*[17] for a group of processes sharing a private namespace, which had an isolated view of system resources such as process identifiers and network addresses. These pods were self-contained, so they could be migrated as a unit between physical machines. In the mid-2000s, Google deployed a cluster management solution called Borg [42, 196] into production, to orchestrate the deployment of their vast suite of web applications and services. Although the code for Borg has never been seen outside Google, it was the direct inspiration for the Kubernetes project a decade later [196, p.18:13–18:14]—the Borg *alloc* became the Kubernetes *pod*, Borglets became Kubelets, and tasks gave way to containers. Burns et al. [42, p. 70] explained that improving performance through resource utilization was one of the primary motivations for Borg.

## 7.5 Combined Features

The strength of modern containers is not in any one feature but in the combination of multiple features for resource control and isolation. In 2008, Linux Containers (LXC) [6] combined cgroups, namespaces, and capabilities from the Linux Kernel into a tool for building and launching low-level system containers. Miller and Chen [142] demonstrated that filesystem isolation between LXC containers could be improved by applying SELinux policies. Xavier et al. [219] and Raho et al. [166] contrasted LXC's approach to isolation and resource control using standard Linux Kernel features such as cgroups and filesystem, process, IPC, and network namespaces, versus the approaches taken by Linux VServer and OpenVZ using custom patches to the Linux Kernel to provide similar features.[18]

[17] Given as an acronym for a *p*r*o*cess *d*omain abstraction.

[18] In the 2000s, many VM or container implementations relied on custom patches to the Linux Kernel, including VServer, OpenVZ, Xen, VMware, and MetaCluster (an earlier version of LXC). The practice was contentious, as multiple incompatible patch sets competed to be merged upstream [57], and ultimately none were ever accepted.

Docker [141] launched in 2013 as a container management platform built on LXC. In 2014, Docker replaced LXC with `libcontainer`, its own implementation for creating containers, which also used Linux Kernel namespaces, cgroups, and capabilities [99, 166]. Morabito et al. [144] compared the performance of LXC and Docker after the transition to libcontainer and found them to be roughly equivalent on CPU performance, disk I/O, and network I/O; however, LXC performed 30% better on random writes, which may have been related to Docker's use of a union file system. Raho et al. [166] contrasted the implementations of Docker, QEMU+KVM, and Xen on the ARM hardware architecture. Mattetti et al. [134] experimented with dynamically generating AppArmor rules for Docker containers based on the application workload they contained. Catuogno and Galdi [45] performed a case study of Docker using two different models for security assessment. They built on the work of Reshetova et al. [167] in classifying vulnerabilities by the goal of the attack: denial of service, container compromise, or privilege escalation.

In 2015, Docker split the container runtime out into a separate project, `runc`, in support of a vendor-neutral container runtime specification maintained by the Open Container Initiative (OCI). Hykes [100] highlighted that SELinux, AppArmor, and seccomp were all standard supported features in `runc`. Koller and Williams [113] observed that `runc` was more minimal than the Docker runtime while still using the same isolation mechanisms from the Linux Kernel, such as namespaces and cgroups. In 2016, Docker and CoreOS merged their container image formats into a vendor-neutral container image format specification, also at OCI [36].

## 7.6 Orchestration

In 2014, Docker began working on Swarm, described as a clustering system for Docker, which they ultimately released late in 2015 [128]. Also in 2014, Google began developing Kubernetes, an orchestration tool for deploying and managing the lifecycle of containers, which they released in the middle of 2015 [38]. Also in 2014, Canonical began developing LXD, a container orchestration tool for LXC containers, which they released in 2016 [89].

Verma et al. [196] outlined the design goals behind Kubernetes, in the context of lessons learned from Borg. Syed and Fernandez [189, 190] pointed out that the performance advantages of the higher-level container orchestration tools, such as Kubernetes and Docker Swarm, were primarily a matter of improving resource utilization. They also contrasted the portability advantages of managing containers across multiple physical host machines against the increased complexity required for the orchestration tools to advance beyond managing a single machine host. Souppaya et al. [184] systematically reviewed increased security risks and mitigation techniques for container orchestration tools. Bila et al. [34] extended Kubernetes with a vulnerability scanning service and network quarantine for containers.

## 7.7 Trade-Offs

Containers have a reputation for substantially better performance than virtual machines; however, that reputation may not be deserved. In 2015, Felter et al. [76] measured the performance of Docker against QEMU+KVM and determined that neither had significant overhead on CPU and memory usage, but that KVM had a 40% higher overhead in I/O. They observed that the overhead was primarily due to extra cycles on each I/O operation, so the impact could be mitigated for some applications by batching multiple small I/O operations into fewer large I/O operations. In 2017, Kovács [114] compared CPU execution time and network throughput between Docker, LXC, Singularity, KVM, and bare metal, and determined that there was no significant variation between them, as long as Docker and LXC were running in host networking mode, but in Linux bridge mode Docker and LXC exhibited high retransmission rates that negatively impacted their throughput compared to the others. Manco et al. [131] demonstrated that Xen virtual machine instantiation could be 50 to 200 times faster than Docker container instantiation, with a few low-level modifications to Xen's control stack.

Secure isolation technologies have been the core of modern container implementations from the beginning, so it would be reasonable to expect that containers would provide a strong form of isolation. However, early implementations of containers were prone to preventable security vulnerabilities, which may indicate that security was not a primary design consideration, at least not initially. Combe et al. [54] analyzed security vulnerabilities in Docker and `libcontainer` between 2014 and 2015, and determined that the majority were related to filesystem isolation, which led to privilege escalation when Docker was run as the root user. They also suggested that some of Docker's sane default configurations for the isolation features of the Linux Kernel could be easily switched to less secure configurations through standard options to the `docker` command-line tool or the Docker daemon, and so might be prone to user error. Martin et al. [133] surveyed vulnerabilities in Docker images, `libcontainer`, the Docker daemon, and orchestration tools, as well as the unique security challenges of containers in multitenant infrastructures. In addition to security patches for specific privilege escalation vulnerabilities, there has been ongoing work to integrate support for user namespaces into Docker and Kubernetes,[19] so they can run as a non-root user and limit the scope of damage from privilege escalation. However, the user namespaces feature itself has had a series of vulnerabilities[20] related to interfaces in the Kernel that were written with the expectation of being restricted to the root user but are now exposed to unprivileged users.

One significant difference between virtual machine implementations and container implementations is that containers share a kernel with the host operating system, so efforts to secure the kernel greatly impact the security of containers. Reshetova et al. [167] considered the set of secure isolation features offered by the Linux Kernel as of 2014 (in the context of LXC) and judged them to have caught up with the features of FreeBSD Jails and Solaris Zones but highlighted some areas for improvement in support of containers. These improvements included integrating MAC into the Kernel as "security namespaces," providing a way to lock down device hotplug features for containers and extending cgroups to support all resource management features supported by rlimits. Gao et al. [79] discussed the risks of certain types of information that containers can currently access from the Linux Kernel via *procfs* and *sysfs*—which can be exploited to detect co-resident containers and precisely target power consumption spikes to overload servers—and prototyped a power-based namespace to partition the information for containers.

Some more recent approaches to secure isolation for containers have been inspired by virtual machine implementations. Kata Containers (formerly Intel Clear Containers) [5] wraps each Docker container or Kubernetes pod in a QEMU+KVM virtual machine [12]. They realized that QEMU was not ideal for the purpose—since it introduces a substantial performance hit compared to running bare containers, and the majority of the code relates to emulation that is not useful for wrapping containers—so a group at Intel started working on a stripped-down version of QEMU called *NEMU* [8]. X-Containers [179] used Xen's paravirtualization features to improve isolation between containers and the host but made an unfortunate trade-off of removing isolation between containers running on the same host. Nabla Containers [7] and gVisor [88] have both taken an approach of improving isolation by heavily filtering system calls from containers to the host kernel, which is a common technique for modern virtual machines.

Bratus et al. [37] noted that the "self-protection" techniques employed by container implementations are a necessary path for future research, since even virtual machines depend on those techniques to protect themselves. Hosseinzadeh et al. [95] explored the possibility that container

[19]Such as Suda and Scrivano [188] and Suda [187].
[20]Such as CVE-2018-6559, CVE-2018-18955, CVE-2014-9717, and CVE-2014-4014.

implementations might directly adapt earlier work (primarily Berger et al. [30]) for virtual machine implementations to integrate a Trusted Platform Module (TPM) as a virtual device.

Container implementations have a potential advantage over virtual machine implementations in addressing the problem of secure isolation over the long-term, not because any existing implementations are inherently superior but because containers take a modular approach to implementation that permits them to be more flexible over time and across different underlying software[21] and hardware architectures, as new ideas for secure isolation evolve.

## 8 SECURITY OUTLOOK

A series of vulnerabilities related to speculative execution and side-channel attacks rose to attention early in 2018. These vulnerabilities collectively upend traditional notions of secure isolation. The current reactionary approach—patching up each vulnerability as it is revealed—works in the short term but is a losing battle in the long term.[22]

Early in 2018, Kocher et al. [111] and Lipp et al. [124] published a set of vulnerabilities, respectively called *Spectre* and *Meltdown*, using techniques involving speculative execution and out-of-order execution. Spectre affects Intel, AMD, and ARM [111, p. 3], can be launched from any user process (including JavaScript code run in a browser) [111, p. 3], and grants access to any memory an attacked process could normally access [111, p. 5]. Meltdown affects Intel x86 architecture, can be launched from any user process, and grants full access to any physical memory on the same machine including kernel memory and memory allocated to any other process [124, p. 1]. In July 2018, Schwarz et al. [177] published a remote variant of Spectre, nicknamed NetSpectre, which is launched through packets over the network and grants access to any physical memory accessible to the attacked process. In August 2018, Van Bulck et al. [193] published a variant of Meltdown, nicknamed Foreshadow or more broadly "L1 Terminal Fault" (L1TF), which is launched from unprivileged user space, and grants access to the L1 data cache, including encrypted data from Intel's Software Guard eXtensions (SGX). In November 2018, Canella et al. [44] reviewed the broad range of speculative execution vulnerabilities and proposed a comprehensive classification of the known variants and mitigations, which also revealed several previously unknown variants.

The models of secure isolation employed by virtual machines and containers offer little protection from the speculative execution vulnerabilities. Containers are vulnerable to Meltdown, although virtual machines are not because they run a different kernel than the host [124, p. 12]. Both virtual machines and containers are vulnerable to Spectre [10, pp. 3, 5, 6], NetSpectre [177, p. 11], and L1TF [205], with varying degrees of compromise. Variants of L1TF[23] are especially troublesome for virtual machines, because they allow an unprivileged process in the user space of a guest to access any memory on the physical machine, including memory allocated to other guests, the host operating system, and host kernel [13]. Multitenant infrastructures generally allow any tenant to deploy a virtual machine or container on any physical machine in the cloud or cluster, which means that it is viable to exploit these vulnerabilities by simply creating an account with a public provider and deploying malicious guests repeatedly, until one of them lands on a physical host with interesting secrets to steal.

The techniques behind the speculative execution vulnerabilities were not new, but the combined application of the techniques was more sophisticated, and the security impact more severe, than previously considered possible. Although these vulnerabilities were only recently discovered and

[21] Such as `pledge` and `unveil` on OpenBSD versus capabilities and namespaces on Linux.
[22] Metaphorically reminiscent of the proverbial small Dutch child attempting to protect the village from flooding by inserting a tiny finger in each leak that springs in the floodbank wall.
[23] Notably CVE-2018-3646.

published by defensive security researchers,[24] it is possible that offensive security researchers[25] discovered and exploited them much earlier, and continue to exploit additional unpublished variants. Although mitigation patches have typically been applied quickly for the known variants of these vulnerabilities [9, 10], it is not feasible to entirely disable speculative execution [111, p. 11] and out-of-order execution [124, p. 14], which are the primary vectors of the attacks, because the performance penalty is prohibitive, and in some cases the hardware simply has no mechanism to disable the features. The probability of further variants being discovered in the coming years is high. A substantial rethink of the fundamental hardware architecture could potentially eliminate the entire class of vulnerabilities, but in the research, development, and production timelines common to hardware vendors, such a significant change could take decades.

Two notable alternative hardware architectures, CHERI and RISC-V, were already under development before the flood of speculative execution vulnerabilities were published. CHERI [215] combines concepts from classic capability systems and RISC architectures, with a strong emphasis on memory protection. RISC-V [24] is a RISC-based hardware architecture, aimed at providing an extensible open source instruction set architecture (ISA) used as an industry standard by a broad array of hardware vendors. Neither CHERI nor RISC-V were designed with speculative execution vulnerabilities in mind, but Watson et al. [204] observed that CHERI mitigates some aspects of Spectre and Meltdown but is vulnerable to speculative memory access, whereas Asanović and O'Connor [23] announced that RISC-V is not vulnerable because it does not perform speculative memory access. In August 2018, Google announced that the open source implementation of its Titan project, providing a hardware root of trust, will likely be based on RISC-V [169]. MIT's Sanctum processor [59] was also based on RISC-V and demonstrated potential for secure hardware partitioning by adding a small secure CPU to the side of the main CPU. Hardware partitioning might provide a way to mitigate the speculative execution vulnerabilities in multitenant environments while avoiding major changes to the kernel and operating system. However, genuinely delivering the level of physical isolation that x86 promised would likely require logical partitioning of the main CPU, RAM, and cache of the machine, so the guests and the host operating system could share resources at the hardware level but be far more restricted at the software level than is currently possible.

The problem of providing secure isolation for containers and virtual machines extends beyond simple refinements to their implementations. When the fundamental assumptions of a system are proven false, then any theorems built on those assumptions may also be false. The secure isolation features of the full stack—from the kernel and operating system, through to virtual machines, containers, and application workloads—are all built on false assumptions about the behavior of the hardware and will need to be re-examined.

## 9 RELATED IMPLEMENTATIONS

Implementation approaches that adopt the label "cloud" [67, 97, 125, 180] are typically virtual machines with added orchestration features to enhance portability. Cloud implementations also tend to favor lighter-weight guest images, which enhances performance and reduces complexity, although cloud images are generally not quite as minimal as container images.

Implementation approaches that adopt the label "unikernel" [117, 129, 212] take minimalist guest images to an extreme, by replacing the kernel and operating system of the guest with a set of highly optimized libraries that provide the same functionality. The code for an application workload is

[24] Also known as "white hat hackers."
[25] Also known as "black hat hackers."

compiled together with the small subset of unikernel libraries required by the application, resulting in a very small binary that runs directly as a guest image. Historically, unikernels have sacrificed portability of guest images, by targeting only a limited set of virtual machine implementations as their host, but recent work has begun exploring running unikernels as containers [213]. The unikernel approach also reduces the portability of application code, since unikernel frameworks tend to require the application code to be written in the same language as the unikernel libraries.

Implementation approaches that adopt the label "serverless" [17, 93, 106, 113] tend to emphasize portability and minimizing complexity. They rely on the underlying infrastructure—typically some combination of bare metal, virtual machines, and/or containers—for whatever secure isolation and performance they provide.

## 10 CONCLUSION

A detailed examination of the history of virtual machines and containers reveals that the two have evolved in tandem from the very beginning. It also reveals that both families of technology are facing significant challenges in providing secure isolation for modern multitenant infrastructures. In light of recent vulnerabilities, patching up existing tools is a necessary and valuable activity in the short term but is not sufficient for the long term. In the coming decades, the computing industry as a whole will need to embrace more radical alternatives in both hardware and software. Current researchers and developers can benefit from a deeper understanding of how virtual machines and containers evolved—and the trade-offs made along the way—to make more informed choices for tomorrow, avoid repeating past mistakes, and build on a solid foundation toward new paths of exploration.

## ACKNOWLEDGMENTS

Thanks to Ravi Nair for help locating and scanning copies of several pivotal IBM papers on virtual machines from the 1960s that were no longer (or perhaps never) available in libraries or online. Thanks also to the reviewers (alphabetically): Clint Adams, Matthew Allen, Ross Anderson, Alastair Beresford, James Bottomley, Peter Capek, Damian Conway, Kees Cook, Jon Crowcroft, Mike Dodson, Tony Finch, Greg Kroah-Hartman, Anil Madhavapeddy, Ronald Minnich, Richard Mortier, Davanum Srinivas, Tom Sutcliffe, Zahra Tarkhani, and Daniel Thomas. Their feedback was greatly appreciated.

## REFERENCES

[1] IBM Corporation. 1971. *Control Program-67 Cambridge Monitor System.* IBM Corporation, Hawthorne, NY.
[2] Intel Corporation. 1981. *iAPX 432 General Data Processor Architecture Reference Manual.* Intel Corporation, Aloha, OR.
[3] IEEE. 1997. *Protection, Audit and Control Interfaces.* Draft POSIX Standard 1003.1e. IEEE, Los Alamitos, CA.
[4] Man7.org. 2018. Capabilities(7) Man Page. Retrieved December 18, 2019 from http://man7.org/linux/man-pages/man7/capabilities.7.html.
[5] Kata Containers. 2018. Home Page. Retrieved December 18, 2019 from https://katacontainers.io/.
[6] Linux Containers. 2018. LXC Introduction. Retrieved December 18, 2019 from https://linuxcontainers.org/lxc/introduction/.
[7] Nabla Containers. 2018. Home Page. Retrieved December 18, 2019 from https://nabla-containers.github.io/.
[8] Github. 2018. NEMU—Modern Hypervisor for the Cloud. Retrieved December 18, 2019 from https://github.com/intel/nemu.
[9] Intel Corporation. 2018. *Retpoline: A Branch Target Injection Mitigation.* White Paper 337131-003. Intel Corporation.
[10] Intel Corporation. 2018. *Speculative Execution Side Channel Mitigations.* Technical Report 336996-003. Intel Corporation.
[11] Man7.org. 2018. User_namespaces(7) Man Page. Retrieved December 18, 2019 from http://man7.org/linux/man-pages/man7/user_namespaces.7.html.

[12] Github. 2019. Kata Containers Architecture. Retrieved December 18, 2019 from https://github.com/kata-containers/documentation.
[13] Kernel.org. 2019. L1TF–L1 Terminal Fault–The Linux Kernel Documentation. Retrieved December 18, 2019 from https://www.kernel.org/doc/html/latest/admin-guide/l1tf.html.
[14] William B. Ackerman and William W. Plummer. 1967. An implementation of a multiprocessing computer system. In *Proceedings of the 1st ACM Symposium on Operating System Principles (SOSP'67)*. ACM, New York, NY, 5.1–5.10.
[15] R. J. Adair, R. U. Bayles, L. W. Comeau, and R. J. Creasy. 1966. *A Virtual Machine System for the 360/40*. Technical Report 36.010. IBM Cambridge Scientific Center, Cambridge, MA.
[16] Keith Adams and Ole Agesen. 2006. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, New York, NY, 2–13.
[17] Gojko Adzic and Robert Chatley. 2017. Serverless computing: Economic and architectural impact. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*. ACM, New York, NY, 884–889.
[18] I. Ahmad, J. M. Anderson, A. M. Holler, R. Kambo, and V. Makhija. 2003. An analysis of disk performance in VMware ESX server virtual machines. In *Proceedings of the 2003 IEEE International Conference on Communications (Cat. No. 03CH37441)*. 65–76.
[19] Amazon. 2019. Firecracker. Retrieved December 18, 2019 from https://firecracker-microvm.github.io/.
[20] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks. 1964. Architecture of the IBM system/360. *IBM Journal of Research and Development* 8, 2 (April 1964), 87–101.
[21] Jonathan Anderson, Stanley Godfrey, and Robert N. Watson. 2017. Towards oblivious sandboxing with Capsicum. *FreeBSD Journal*.
[22] Andrea Arcangeli. 2005. [PATCH] Seccomp: Secure Computing Support. Retrieved December 18, 2019 from https://git.kernel.org/pub/scm/linux/kernel/git/tglx/history.git/commit/?id=d949d0ec9c601f2b148bed3cdb5f87c052968554.
[23] Krste Asanović and Rick O'Connor. 2018. Building a More Secure World with the RISC-V ISA. Retrieved December 18, 2019 from https://riscv.org/2018/01/more-secure-world-risc-v-isa/.
[24] Krste Asanović and David A. Patterson. 2014. *Instruction Sets Should Be Free: The Case for RISC-V*. Technical Report UCB/EECS-2014-146. EECS Department, University of California, Berkeley.
[25] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. 1999. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*. 45–58.
[26] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*. ACM, New York, NY, 164–177.
[27] P. R. Barham. 1997. A fresh approach to file system quality of service. In *Proceedings of 7th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV'97)*. 113–122.
[28] Jeff Barr. 2006. Amazon EC2 Beta. https://aws.amazon.com/blogs/aws/amazon_ec2_beta/.
[29] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference (ATEC'05)*. 41.
[30] Stefan Berger, Ramon Caceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. 2006. vTPM: Virtualizing the Trusted Platform Module. In *Proceedings of the 15th USENIX Security Symposium*. 305–320.
[31] D. Bernstein. 2014. Containers and cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing* 1, 3 (Sept. 2014), 81–84.
[32] Viktors Berstis. 1980. Security and protection of data in the IBM system/38. In *Proceedings of the 7th Annual Symposium on Computer Architecture (ISCA'80)*. ACM, New York, NY, 245–252.
[33] Eric W. Biederman. 2006. Multiple instances of the global linux namespaces. In *Proceedings of the Linux Symposium*, Vol. 1. 101–112.
[34] N. Bila, P. Dettori, A. Kanso, Y. Watanabe, and A. Youssef. 2017. Leveraging the serverless architecture for securing Linux containers. In *Proceedings of the 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW'17)*. 401–404.
[35] Matthias Bolte, Michael Sievers, Georg Birkenheuer, Oliver Niehörster, and André Brinkmann. 2010. Non-intrusive virtualization management using libvirt. In *Proceedings of the Conference on Design, Automation, and Test in Europe (DATE'10)*. 574–579.
[36] Jonathan Boulle. 2016. Celebrating the Open Container Initiative Image Specification. Retrieved December 18, 2019 from https://coreos.com/blog/oci-image-specification.html.
[37] Sergey Bratus, Michael E. Locasto, Ashwin Ramaswamy, and Sean W. Smith. 2010. VM-based security overkill: A lament for applied systems security research. In *Proceedings of the 2010 New Security Paradigms Workshop (NSPW'10)*. ACM, New York, NY, 51–60.

[38] Eric A. Brewer. 2015. Kubernetes and the path to cloud native. In *Proceedings of the 6th ACM Symposium on Cloud Computing (SoCC'15)*. ACM, New York, NY, 167.

[39] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. 1997. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems* 15, 4 (Nov. 1997), 412–447.

[40] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y. Wang. 2012. Bringing virtualization to the x86 architecture with the original VMware workstation. *ACM Transactions on Computer Systems* 30, 4 (Nov. 2012), Article 12, 51 pages.

[41] E. Bugnion, J. Nieh, and D. Tsafrir. 2017. *Hardware and Software Support for Virtualization*. Morgan & Claypool.

[42] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Queue* 14, 1 (Jan. 2016), 10:70–10:93.

[43] J. P. Buzen and U. O. Gagliardi. 1973. The evolution of virtual machine architecture. In *Proceedings of the 1973 National Computer Conference and Exposition (AFIPS'73)*. ACM, New York, NY, 291–299.

[44] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2018. A systematic evaluation of transient execution attacks and defenses. arXiv:1811.05441.

[45] L. Catuogno and C. Galdi. 2016. On the evaluation of security properties of containerized systems. In *Proceedings of the 2016 15th International Conference on Ubiquitous Computing and Communications and 2016 International Symposium on Cyberspace and Security (IUCC-CSS'16)*. 69–76.

[46] Ramaswamy Chandramouli, Anoop Singhal, Duminda Wijesekera, and Changwei Liu. 2018. *A Methodology for Determining Forensic Data Requirements for Detecting Hypervisor Attacks*. Technical Report NISTIR 8221 (Draft). National Institute of Standards and Technology. https://csrc.nist.gov/publications/detail/nistir/8221/draft.

[47] David Chisnall. 2007. *The Definitive Guide to the Xen Hypervisor*. Prentice Hall Press, Upper Saddle River, NJ.

[48] J. Claassen, R. Koning, and P. Grosso. 2016. Linux containers networking: Performance and scalability of kernel modules. In *Proceedings of the 2016 IEEE/IFIP Network Operations and Management Symposium (NOMS'16)*. 713–717.

[49] E. F. Codd. 1960. Multiprogram scheduling: Parts 1 and 2. Introduction and theory. *Communications of the ACM* 3, 6 (June 1960), 347–350.

[50] E. F. Codd. 1960. Multiprogram scheduling: Parts 3 and 4. Scheduling algorithm and external constraints. *Communications of the ACM* 3, 7 (July 1960), 413–418.

[51] E. F. Codd. 1962. Multiprogramming. In *Advances in Computers*, Vol. 3, F. L. Alt and M. Rubinoff (Eds.). Elsevier, 77–153.

[52] E. F. Codd, E. S. Lowry, E. McDonough, and C. A. Scalzi. 1959. Multiprogramming STRETCH: Feasibility considerations. *Communications of the ACM* 2, 11 (Nov. 1959), 13–17.

[53] Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. 2011. Breaking up is hard to do: Security and functionality in a commodity hypervisor. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*. ACM, New York, NY, 189–202.

[54] T. Combe, A. Martin, and R. Di Pietro. 2016. To Docker or not to Docker: A security perspective. *IEEE Cloud Computing* 3, 5 (Sept. 2016), 54–62.

[55] Fernando J. Corbató, Marjorie Merwin-Daggett, and Robert C. Daley. 1962. An experimental time-sharing system. In *Proceedings of the Spring Joint Computer Conference (AIEE-IRE'62 (Spring))*. ACM, New York, NY, 335–344.

[56] F. J. Corbató, J. H. Saltzer, and C. T. Clingen. 1972. Multics: The first seven years. In *Proceedings of the Spring Joint Computer Conference (AFIPS'72 (Spring))*. ACM, New York, NY, 571–583.

[57] Jonathan Corbet. 2006. Kernel Summit 2006: Paravirtualization and Containers. Retrieved December 18, 2019 from https://lwn.net/Articles/191923/.

[58] Jonathan Corbet. 2007. Process Containers. Retrieved December 18, 2019 from https://lwn.net/Articles/236038/.

[59] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal hardware extensions for strong software isolation. In *Proceedings of the 25th USENIX Security Symposium*. 857–874.

[60] Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Virgil Gligor. 2000. SubDomain: Parsimonious server security. In *Proceedings of the 14th USENIX Conference on System Administration (LISA'00)*. 355–368.

[61] R. J. Creasy. 1981. The origin of the VM/370 time-sharing system. *IBM Journal of Research and Development* 25, 5 (Sept. 1981), 483–490.

[62] Peter J. Denning. 1976. Fault tolerant operating systems. *ACM Computing Surveys* 8, 4 (Dec. 1976), 359–389.

[63] Peter J. Denning. 1981. Performance modeling: Experimental computer science as its best. *Communications of the ACM, President's Letter* 24, 11 (Nov. 1981), 725–727.

[64] Jack B. Dennis. 1965. Segmentation and the design of multiprogrammed computer systems. *Journal of the ACM* 12, 4 (Oct. 1965), 589–602.

[65] Jack B. Dennis and Earl C. Van Horn. 1966. Programming semantics for multiprogrammed computations. *Communications of the ACM* 9, 3 (March 1966), 143–155.

[66] Lloyd I. Dickman. 1973. Small virtual machines: A survey. In *Proceedings of the Workshop on Virtual Computer Systems*. ACM, New York, NY, 191–202.

[67] M. S. Dildar, N. Khan, J. B. Abdullah, and A. S. Khan. 2017. Effective way to defend the hypervisor attacks in cloud computing. In *Proceedings of the 2017 2nd International Conference on Anti-Cyber Crimes (ICACC'17)*. 154–159.

[68] Will Drewry. 2012. Dynamic Seccomp Policies (Using BPF Filters). Retrieved December 18, 2019 from https://lwn.net/Articles/475019/.

[69] S. W. Dunwell. 1957. Design objectives for the IBM stretch computer. In *Papers and Discussions Presented at the Eastern Joint Computer Conference: New Developments in Computers (AIEE-IRE'56 (Eastern))*. ACM, New York, NY, 20–22.

[70] J. P. Eckert. 1957. UNIVAC-Larc, the next step in computer design. In *Papers and Discussions Presented at the Eastern Joint Computer Conference: New Developments in Computers (AIEE-IRE'56 (Eastern))*. ACM, New York, NY, 16–20.

[71] Heiko Eißfeldt. 1997. POSIX: A developer's view of standards. In *Proceedings of the USENIX Annual Technical Conference (ATEC'97)*. 24.

[72] D. M. England. 1974. Capability concept mechanism and structure in system 250. In *Proceedings of the International Workshop on Protection in Operating Systems*. 63–82.

[73] R. S. Fabry. 1967. *A User's View of Capabilities*. ICR Quarterly Report 15. University of Chicago.

[74] R. S. Fabry. 1968. *Preliminary Description of a Supervisor for a Machine Oriented Around Capabilities*. ICR Quarterly Report 18. University of Chicago.

[75] R. S. Fabry. 1971. *List-Structured Addressing*. Ph.D. Dissertation. University of Chicago.

[76] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. 2015. An updated performance comparison of virtual machines and Linux containers. In *Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'15)*. 171–172.

[77] J. M. Frankovich and H. P. Peterson. 1957. A functional description of the Lincoln TX-2 computer. In *Papers Presented at the Western Joint Computer Conference: Techniques for Reliability (IRE-AIEE-ACM'57 (Western))*. ACM, New York, NY, 146–155.

[78] S. W. Galley. 1973. PDP-10 virtual machines. In *Proceedings of the Workshop on Virtual Computer Systems*. ACM, New York, NY, 30–34.

[79] X. Gao, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang. 2017. ContainerLeaks: Emerging security threats of information leakages in container clouds. In *Proceedings of the 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'17)*. 237–248.

[80] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. 2007. Compatibility is not transparency: VMM detection myths and realities. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems (HOTOS'07)*. 6.

[81] Tal Garfinkel and Mendel Rosenblum. 2003. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Network and Distributed Systems Security Symposium*, Vol. 1. 253–285.

[82] S. Gill. 1958. Parallel programming. *Computer Journal* 1, 1 (Jan. 1958), 2–10.

[83] R. P. Goldberg. 1972. *Architectural Principles for Virtual Computer Systems*. Ph.D. Dissertation. Harvard University, Cambridge, MA.

[84] R. P. Goldberg. 1973. Architecture of virtual machines. In *Proceedings of the Workshop on Virtual Computer Systems*. ACM, New York, NY, 74–112.

[85] Robert P. Goldberg. 1974. Survey of virtual machine research. *Computer* 7, 6 (June 1974), 34–45.

[86] Google. 2018. Chrome OS Virtual Machine Monitor. Retrieved December 18, 2019 from https://chromium.googlesource.com/chromiumos/platform/crosvm/.

[87] Google. 2018. Fuchsia Is Not Linux: A Modular, Capability-Based Operating System. Retrieved December 18, 2019 from https://fuchsia.dev/fuchsia-src.

[88] Google. 2019. gVisor: Runtime Sandbox. Retrieved December 18, 2019 from https://github.com/google/gvisor.

[89] Stéphane Graber. 2016. LXD 2.0. Retrieved December 18, 2019 from https://stgraber.org/2016/03/11/lxd-2-0-blog-post-series-012/.

[90] Andreas Grünbacher. 2003. POSIX access control lists on Linux. In *Proceedings of the 2003 USENIX Annual Technical Conference*. 259–272.

[91] Paul M. Hansen, Mark A. Linton, Robert N. Mayo, Marguerite Murphy, and David A. Patterson. 1982. A performance evaluation of the Intel iAPX 432. *SIGARCH Computer Architecture News* 10, 4 (June 1982), 17–26.

[92] Norman Hardy. 1985. KeyKOS architecture. *SIGOPS Operating Systems Review* 19, 4 (Oct. 1985), 8–25.

[93] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2018. Serverless computing: One step forward, two steps back. arXiv:1812.03651.

[94] John Hennessy. 2018. The era of security: Introduction. In *Proceedings of the 2018 IEEE Hot Chips Symposium*. IEEE, Los Alamitos, CA. https://youtu.be/d5XzVF0sAZo.

[95] Shohreh Hosseinzadeh, Samuel Laurén, and Ville Leppänen. 2016. Security in container-based virtualization through vTPM. In *Proceedings of the 9th International Conference on Utility and Cloud Computing (UCC'16)*. ACM, New York, NY, 214–219.

[96] Merle E. Houdek, Frank G. Soltis, and Roy L. Hoffman. 1981. IBM system/38 support for capability-based addressing. In *Proceedings of the 8th Annual Symposium on Computer Architecture (ISCA'81)*. IEEE, Los Alamitos, CA, 341–348.

[97] Wei Huang, Afshar Ganjali, Beom Heyn Kim, Sukwon Oh, and David Lie. 2015. The state of public infrastructure-as-a-service cloud security. *ACM Computing Surveys* 47, 4 (June 2015), Article 68, 31 pages.

[98] Yih Huang, Angelos Stavrou, Anup K. Ghosh, and Sushil Jajodia. 2008. Efficiently tracking application interactions using lightweight virtualization. In *Proceedings of the 1st ACM Workshop on Virtual Machine Security (VMSec'08)*. ACM, New York, NY, 19–28.

[99] Solomon Hykes. 2014. Docker 0.9: Introducing Execution Drivers and Libcontainer. Retrieved December 18, 2019 from https://blog.docker.com/2014/03/docker-0-9-introducing-execution-drivers-and-libcontainer/.

[100] Solomon Hykes. 2015. Introducing runC: A Lightweight Universal Container Runtime. Retrieved December 18, 2019 from https://blog.docker.com/2015/06/runc/.

[101] Kenta Ishiguro and Kenji Kono. 2018. Hardening hypervisors against vulnerabilities in instruction emulators. In *Proceedings of the 11th European Workshop on Systems Security (EuroSec'18)*. ACM, New York, NY, Article 7, 6 pages.

[102] Zhiqiang Jian and Long Chen. 2017. A defense method against docker escape attack. In *Proceedings of the 2017 International Conference on Cryptography, Security, and Privacy (ICCSP'17)*. ACM, New York, NY, 142–146.

[103] Anita K. Jones, Robert J. Chansler Jr., Ivor Durham, Karsten Schwans, and Steven R. Vegdahl. 1979. StarOS, a multiprocessor operating system for the support of task forces. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP'79)*. ACM, New York, NY, 117–127.

[104] A. M. Joy. 2015. Performance comparison between Linux containers and virtual machines. In *Proceedings of the 2015 International Conference on Advances in Computer Engineering and Applications*. 342–346.

[105] Poul-Henning Kamp and Robert N. M. Watson. 2000. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*.

[106] Ali Kanso and Alaa Youssef. 2017. Serverless: Beyond the cloud. In *Proceedings of the 2nd International Workshop on Serverless Computing (WoSC'17)*. ACM, New York, NY, 6–10.

[107] Jason A. Kappel, Anthony Velte, and Toby Velte. 2009. *Microsoft Virtualization with Hyper-V: Manage Your Datacenter with Hyper-V, Virtual PC, Virtual Server, and Application Virtualization*. McGraw Hill Professional.

[108] B. W. Kernighan and M. D. McIlroy. 1979. *UNIX Time-Sharing System: UNIX Programmer's Manual* (7th ed.). Vol. 1. Bell Telephone Laboratories Incorporated, Murray Hill, NJ.

[109] Michael Kerrisk. 2013. Namespaces in Operation, Part 1: Namespaces Overview. Retrieved December 18, 2019 from https://lwn.net/Articles/531114/.

[110] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. KVM: The Linux virtual machine monitor. In *Proceedings of the 2007 Ottawa Linux Symposium (OLS'07)*.

[111] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre attacks: Exploiting speculative execution. arXiv:1801.01203.

[112] Richard M. Kogut. 1973. The segment based file support system. In *Proceedings of the Workshop on Virtual Computer Systems*. ACM, New York, NY, 35–42.

[113] Ricardo Koller and Dan Williams. 2017. Will serverless end the dominance of Linux in the cloud? In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS'17)*. ACM, New York, NY, 169–173.

[114] Ákos Kovács. 2017. Comparison of different Linux containers. In *Proceedings of the 2017 40th International Conference on Telecommunications and Signal Processing (TSP'17)*. 47–51.

[115] Johannes Krude and Ulrike Meyer. 2013. A versatile code execution isolation framework with security first. In *Proceedings of the 2013 ACM Workshop on Cloud Computing Security Workshop (CCSW'13)*. ACM, New York, NY, 1–10.

[116] Butler W. Lampson and Howard E. Sturgis. 1976. Reflections on an operating system design. *Communications of the ACM* 19, 5 (May 1976), 251–265.

[117] Stefan Lankes, Simon Pickartz, and Jens Breitbart. 2016. HermitCore: A unikernel for extreme scale computing. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS'16)*. ACM, New York, NY, Article 4, 8 pages.

[118] Hugh C. Lauer and C. R. Snow. 1972. Is supervisor-state necessary? In *Proceedings of the ACM AICA International Computing Symposium*.

[119] Hugh C. Lauer and David Wyeth. 1973. A recursive virtual machine architecture. In *Proceedings of the Workshop on Virtual Computer Systems*. ACM, New York, NY, 113–116.

[120] Henry M. Levy. 1984. *Capability-Based Computer Systems*. Digital Press, Newton, MA.
[121] Z. Li, M. Kihl, Q. Lu, and J. A. Andersson. 2017. Performance overhead comparison between hypervisor and container based virtualization. In *Proceedings of the 2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA'17)*. 955–962.
[122] Anthony Liguori. 2012. QEMU 1.3.0 Release. Retrieved December 18, 2019 from https://lists.gnu.org/archive/html/qemu-devel/2012-12/msg00123.html.
[123] Steven B. Lipner, William A. Wulf, Roger R. Schell, Gerald J. Popek, Peter G. Neumann, Clark Weissman, and Theodore A. Linden. 1974. Security kernels. In *Proceedings of the AFIPS National Computer Conference (AFIPS'74)*. ACM, New York, NY, 973–980.
[124] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. arXiv:1801.01207.
[125] Flavio Lombardi and Roberto Di Pietro. 2011. Secure virtualization for cloud computing. *Journal of Network and Computer Applications* 34, 4 (July 2011), 1113–1122.
[126] Peter Loscocco and Stephen Smalley. 2001. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*. 29–42.
[127] R. Lottiaux and C. Morin. 2001. Containers: A sound basis for a true single system image. In *Proceedings of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid*. 66–73.
[128] Andrea Luzzardi. 2015. Announcing Swarm 1.0: Production-Ready Clustering at Any Scale. Retrieved December 18, 2019 from https://blog.docker.com/2015/11/swarm-1-0/.
[129] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library operating systems for the cloud. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. ACM, New York, NY, 461–472.
[130] Stuart E. Madnick and John J. Donovan. 1973. Application and analysis of the virtual machine approach to information system security and isolation. In *Proceedings of the Workshop on Virtual Computer Systems*. ACM, New York, NY, 210–224.
[131] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. ACM, New York, NY, 218–233.
[132] David Margery, Renaud Lottiaux, and Christine Morin. 2004. *Capabilities for per Process Tuning of Distributed Operating Systems*. Research Report RR-5411. INRIA.
[133] A. Martin, S. Raponi, T. Combe, and R. Di Pietro. 2018. Docker ecosystem—Vulnerability analysis. *Computer Communications* 122 (June 2018), 30–43.
[134] M. Mattetti, A. Shulman-Peleg, Y. Allouche, A. Corradi, S. Dolev, and L. Foschini. 2015. Securing the infrastructure and the workloads of Linux containers. In *Proceedings of the 2015 IEEE Conference on Communications and Network Security (CNS'15)*. 559–567.
[135] Jeanna Neefe Matthews, Wenjin Hu, Madhujith Hapuarachchi, Todd Deshane, Demetrios Dimatos, Gary Hamilton, Michael McCabe, and James Owens. 2007. Quantifying the performance isolation properties of virtualization systems. In *Proceedings of the 2007 Workshop on Experimental Computer Science (ExpCS'07)*. ACM, New York, NY.
[136] Alastair J. W. Mayer. 1982. The architecture of the Burroughs B5000: 20 years later and still ahead of the times? *SIGARCH Computer Architecture News* 10, 4 (June 1982), 3–10.
[137] S. Mazor. 2010. Intel's 8086. *IEEE Annals of the History of Computing* 32, 1 (Jan. 2010), 75–79.
[138] Marshall Kirk McKusick. 1999. Twenty years of Berkeley Unix—From AT&T-owned to freely redistributable. In *Open Sources: Voices from the Open Source Revolution*. O'Reilly Media Inc.
[139] Marshall K. McKusick, Michael J. Karels, Keith Sklower, Kevin Fall, Marc Teitelbaum, and Keith Bostic. 1989. Current research by the computer systems research group of Berkeley. In *Proceedings of the European UNIX Users Group*.
[140] Marshall Kirk McKusick, George V. Neville-Neil, and Robert N. M. Watson. 2014. *The Design and Implementation of the FreeBSD Operating System* (2nd ed.). Addison-Wesley Professional.
[141] Dirk Merkel. 2014. Docker: Lightweight Linux containers for consistent development and deployment. *Linux Journal* 2014, 239 (March 2014), Article 2.
[142] Adam Miller and Lei Chen. 2012. An exercise in secure high performance virtual containers. In *Proceedings of the 2012 International Conference on Security and Management (SAM'12)*. 5.
[143] Mark S. Miller, Ka-Ping Yee, and Jonathan Shapiro. 2003. *Capability Myths Demolished*. Technical Report SRL2003-02. Johns Hopkins University, Systems Research Laboratory, MD.
[144] R. Morabito, J. Kjällman, and M. Komu. 2015. Hypervisors vs. lightweight virtualization: A performance comparison. In *Proceedings of the 2015 IEEE International Conference on Cloud Engineering*. 386–393.

[145] C. Morin, P. Gallard, R. Lottiaux, and G. Vallee. 2002. Towards an efficient single system image cluster operating system. In *Proceedings of the 2002 5th International Conference on Algorithms and Architectures for Parallel Processing*. 370–377.

[146] Shailabh Nagar, Hubertus Franke, Jonghyuk Choi, Chandra Seetharaman, Scott Kaplan, Nivedita Singhvi, Vivek Kashyap, and Mike Kravetz. 2003. Class-based prioritized resource control in Linux. In *Proceedings of the Linux Symposium*. 21.

[147] S. Nanba, N. Ohno, H. Kubo, H. Morisue, T. Ohshima, and H. Yamagishi. 1985. VM/4: ACOS-4 virtual machine architecture. In *Proceedings of the 12th Annual International Symposium on Computer Architecture (ISCA'85)*. IEEE, Los Alamitos, CA, 171–178.

[148] R. M. Needham and R. D. H. Walker. 1977. The Cambridge CAP computer and its protection system. In *Proceedings of the 6th ACM Symposium on Operating Systems Principles*. ACM, New York, NY, 1–10.

[149] R. A. Nelson. 1964. *Mapping Devices and the M44 Data Processing System*. Research Report RC-1303. IBM Thomas J. Watson Research Center, Yorktown Heights, NY.

[150] P. G. Neumann. 1980. *A Provably Secure Operating System: The System, Its Applications, and Proofs*. Technical Report. Computer Science Laboratory, SRI International.

[151] P. G. Neumann and R. J. Feiertag. 2003. PSOS revisited. In *Proceedings of the 19th Annual Computer Security Applications Conference*. 208–216.

[152] Robert M. Norton. 2016. *Hardware Support for Compartmentalisation*. Technical Report UCAM-CL-TR-887. University of Cambridge, Computer Laboratory.

[153] Ascher Opler and Norma Baird. 1959. Multiprogramming: The programmer's view. In *Preprints of Papers Presented at the 14th National Meeting of the Association for Computing Machinery (ACM'59)*. ACM, New York, NY, 1–4.

[154] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. 2002. The design and implementation of Zap: A system for migrating computing environments. In *Proceedings of the 5th Operating Systems Design and Implementation Conference (OSDI'02)*.

[155] R. P. Parmelee, T. I. Peterson, C. C. Tillman, and D. J. Hatfield. 1972. Virtual storage and virtual machine concepts. *IBM Systems Journal* 11, 2 (1972), 99–130.

[156] David A. Patterson and David R. Ditzel. 1980. The case for the reduced instruction set computer. *SIGARCH Computer Architecture News* 8, 6 (Oct. 1980), 25–33.

[157] David A. Patterson and Carlo H. Sequin. 1981. RISC I: A reduced instruction set VLSI computer. In *Proceedings of the 8th Annual Symposium on Computer Architecture (ISCA'81)*. IEEE, Los Alamitos, CA, 443–457.

[158] Michael Pearce, Sherali Zeadally, and Ray Hunt. 2013. Virtualization: Issues, security threats, and solutions. *ACM Computing Surveys* 45, 2 (Feb. 2013), 1–39.

[159] Diego Perez-Botero, Jakub Szefer, and Ruby B. Lee. 2013. Characterizing hypervisor vulnerabilities in cloud computing servers. In *Proceedings of the 2013 International Workshop on Security in Cloud Computing (Cloud Computing'13)*. ACM, New York, NY, 3–10.

[160] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. 1993. The use of name spaces in plan 9. *SIGOPS Operating Systems Review* 27, 2 (April 1993), 72–76.

[161] Gerald Popek and Charles Kline. 1975. A verifiable protection system. *ACM SIGPLAN Notices* 10, 6 (June 1975), 294–304.

[162] Gerald J. Popek and Robert P. Goldberg. 1974. Formal requirements for virtualizable third generation architectures. *Communications of the ACM* 17, 7 (July 1974), 412–421.

[163] Jon Postel. 1981. *Internet Protocol*. Request for Comments 791. Internet Engineering Task Force (IETF), Defense Advanced Research Projects Agency (DARPA), Marina del Rey, California.

[164] Daniel Price and Andrew Tucker. 2004. Solaris zones: Operating system support for consolidating commercial workloads. In *Proceedings of the 18th USENIX Conference on System Administration (LISA'04)*. 241–254.

[165] Reid Priedhorsky and Tim Randles. 2017. Charliecloud: Unprivileged containers for user-defined software stacks in HPC. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'17)*. ACM, New York, NY, Article 36, 10 pages.

[166] M. Raho, A. Spyridakis, M. Paolino, and D. Raho. 2015. KVM, Xen and Docker: A performance analysis for ARM based NFV and cloud computing. In *Proceedings of the 2015 IEEE 3rd Workshop on Advances in Information, Electronic, and Electrical Engineering (AIEEE'15)*. 1–8.

[167] Elena Reshetova, Janne Karhunen, Thomas Nyman, and N. Asokan. 2014. Security of OS-level virtualization technologies. In *Secure IT Systems*. Lecture Notes in Computer Science, Vol. 8788. Springer, 77–93.

[168] Dennis Ritchie. 1980. The evolution of the unix time-sharing system. In *Language Design and Programming Methodology*. Lecture Notes in Computer Science, Vol. 79. Springer, 25–35.

[169] Dominic Rizzo and Parthasarathy Ranganathan. 2018. Titan: Google's root-of-trust security silicon. In *Proceedings of the IEEE Hot Chips Symposium*. IEEE, Los Alamitos, CA.

[170] John Scott Robin and Cynthia E. Irvine. 2000. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th USENIX Security Symposium.* 129–144.

[171] Nathaniel Rochester. 1955. The computer and its peripheral equipment. In *Papers and Discussions Presented at the the Eastern Joint AIEE-IRE Computer Conference: Computers in Business and Industrial Systems (AIEE-IRE'55 (Eastern)).* ACM, New York, NY, 64–69.

[172] Mendel Rosenblum and Tal Garfinkel. 2005. Virtual machine monitors: Current technology and future trends. *Computer* 38, 5 (May 2005), 39–47.

[173] J. M. Rushby. 1981. Design and verification of secure systems. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP'81).* ACM, New York, NY, 12–21.

[174] J. H. Saltzer and M. D. Schroeder. 1975. The protection of information in computer systems. *Proceedings of the IEEE* 63, 9 (Sept. 1975), 1278–1308.

[175] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. 2002. Optimizing the migration of virtual computers. *SIGOPS Operating Systems Review* 36, SI (Dec. 2002), 377–390.

[176] Gottfried Schimunek, Danny Dupuche, Tim Fung, Paul Kirkdale, Erik Myhra, and Helmut Stein. 1999. *Slicing the AS/400 with Logical Partitioning: A How to Guide.* IBM Corporation.

[177] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. 2018. NetSpectre: Read arbitrary memory over network. arXiv:1807.10535.

[178] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. 2007. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07).* ACM, New York, NY, 335–350.

[179] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. 2019. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19) [Preprint].* ACM, New York, NY, 15.

[180] Jatinder Singh, Jean Bacon, Jon Crowcroft, Anil Madhavapeddy, Thomas Pasquier, W. Kuan Hon, and Christopher Millard. 2014. *Regional Clouds: Technical Considerations.* Technical Report UCAM-CL-TR-863. Computer Laboratory, University of Cambridge.

[181] J. E. Smith and Ravi Nair. 2005. The architecture of virtual machines. *Computer* 38, 5 (May 2005), 32–38.

[182] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. 2007. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proceedings of the 2007 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems.* ACM, New York, NY, 275–287.

[183] Frank G. Soltis. 2001. *Fortress Rochester: The Inside Story of the IBM ISeries.* System iNetwork.

[184] Murugiah Souppaya, John Morello, and Karen Scarfone. 2017. *Application Container Security Guide.* Technical Report NIST SP 800-190. National Institute of Standards and Technology, Gaithersburg, MD.

[185] Ronald J. Srodawa and Lee A. Bates. 1973. An efficient virtual machine implementation. In *Proceedings of the Workshop on Virtual Computer Systems.* ACM, New York, NY, 43–73.

[186] Howard E. Sturgis. 1973. *A Postmortem for a Time Sharing System.* Ph.D. Dissertation. University of California at Berkeley, Berkeley, CA.

[187] Akihiro Suda. 2019. Allow Running Dockerd as a Non-Root User (Rootless Mode). Retrieved December 18, 2019 from https://github.com/moby/moby/pull/38050.

[188] Akihiro Suda and Giuseppe Scrivano. 2019. Rootless Kubernetes. Retrieved December 18, 2019 from https://fosdem.org/2019/schedule/event/containers_k8s_rootless/.

[189] Madiha H. Syed and Eduardo B. Fernandez. 2017. The container manager pattern. In *Proceedings of the 22nd European Conference on Pattern Languages of Programs (EuroPLoP'17).* ACM, New York, NY, Article 28, 9 pages.

[190] Madiha H. Syed and Eduardo B. Fernandez. 2018. A reference architecture for the container ecosystem. In *Proceedings of the 13th International Conference on Availability, Reliability, and Security (ARES'18).* ACM, New York, NY, Article 31, 6 pages.

[191] Jakub Szefer, Eric Keller, Ruby B. Lee, and Jennifer Rexford. 2011. Eliminating the hypervisor attack surface for a more secure cloud. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11).* ACM, New York, NY, 401–412.

[192] L. Szekeres, M. Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy.* IEEE, Los Alamitos, CA, 48–62.

[193] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18).* 991–1008.

[194] Melinda Varian. 1989. VM and the VM Community: Past, Present, and Future. Retrieved December 18, 2019 from http://www.leeandmelindavarian.com/Melinda/neuvm.pdf.

[195] Amit Vasudevan, Jonathan M. McCune, Ning Qu, Leendert Van Doorn, and Adrian Perrig. 2010. Requirements for an integrity-protected hypervisor on the x86 hardware virtualized architecture. In *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing (TRUST'10)*. 141–165.

[196] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15)*. ACM, New York, NY, Article 18, 17 pages.

[197] Carl A. Waldspurger. 2002. Memory resource management in VMware ESX server. *SIGOPS Operating Systems Review* 36, SI (Dec. 2002), 181–194.

[198] Zhi Wang, Chiachih Wu, Michael Grace, and Xuxian Jiang. 2012. Isolating commodity hosted hypervisors with hyperlock. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*. ACM, New York, NY, 127–140.

[199] Robert Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. 2010. Capsicum: Practical capabilities for UNIX. In *Proceedings of the 19th USENIX Security Symposium*, Vol. 19. ACM, New York, NY.

[200] Robert Watson, Peter Neumann, Jonathan Woodruff, Jonathan Anderson, Ross Anderson, Nirav Dave, Ben Laurie, et al. 2012. CHERI: A research platform deconflating hardware virtualization and protection. In *The Unpublished Workshop Paper for RESoLVE'12*.

[201] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. 2012. A taste of capsicum: Practical capabilities for UNIX. *Communications of the ACM* 55, 3 (March 2012), 97–104.

[202] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Simon W. Moore, Steven J. Murdoch, and Michael Roe. 2014. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture*. Technical Report UCAM-CL-TR-864. Computer Laboratory, University of Cambridge.

[203] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, et al. 2015. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. 20–37.

[204] Robert N. M. Watson, Jonathan Woodruff, Michael Roe, Simon W. Moore, and Peter G. Neumann. 2018. *Capability Hardware Enhanced RISC Instructions (CHERI): Notes on the Meltdown and Spectre Attacks*. Technical Report UCAM-CL-TR-916. Computer Laboratory, University of Cambridge.

[205] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. 2018. *Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution*. Technical Report. Foreshadow.

[206] Andrew Whitaker, Marianne Shaw, and Steven Gribble. 2002. *Denali: Lightweight Virtual Machines for Distributed and Networked Applications*. Technical Report. University of Washington.

[207] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. 2002. Denali: A scalable isolation kernel. In *Proceedings of the 10th ACM SIGOPS European Workshop*. ACM, New York, NY, 10–15.

[208] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. 2002. Scale and performance in the Denali isolation kernel. *SIGOPS Operating Systems Review* 36, SI (Dec. 2002), 195–209.

[209] Maurice V. Wilkes. 1968. *Time-Sharing Computer Systems* (2nd ed.). Number 5 in MacDonald Computer Monographs. MacDonald & Co.

[210] M. V. Wilkes. 1979. *The Cambridge CAP Computer and Its Operating System. Operating and Programming Systems Series)*. North-Holland Publishing Co., Amsterdam, The Netherlands.

[211] M. V. Wilkes and D. W. Willis. 1956. A magnetic-tape auxiliary storage system for the EDSAC. *Proceedings of the IEE—Part B: Radio and Electronic Engineering* 103, 2 (April 1956), 337–345.

[212] Dan Williams and Ricardo Koller. 2016. Unikernel monitors: Extending minimalism outside of the box. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'16)*. 6.

[213] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. 2018. Unikernels as processes. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'18)*. ACM, New York, NY, 199–211.

[214] Dan Williams, Ricardo Koller, and Brandon Lum. 2018. Say goodbye to virtualization for a safer cloud. In *Proceedings of the 10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'18)*.

[215] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceedings of the 41st Annual International Symposium on Computer Architecuture (ISCA'14)*. IEEE, Los Alamitos, CA, 457–468.

[216] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. 2002. Linux Security Module framework. In *Proceedings of the Ottawa Linux Symposium*. 604–617.

[217] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. 1974. HYDRA: The kernel of a multiprocessor operating system. *Communications of the ACM* 17, 6 (June 1974), 337–345.

[218] William Allan Wulf, Roy Levin, and Samuel P. Harbison. 1981. *HYDRA-C. Mmp: An Experimental Computer System.* McGraw-Hill.

[219] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. De Rose. 2013. Performance evaluation of container-based virtualization for high performance computing environments. In *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 233–240.

[220] Yan Zhai, Lichao Yin, Jeffrey Chase, Thomas Ristenpart, and Michael Swift. 2016. CQSTR: Securing cross-tenant applications with cloud containers. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC'16).* ACM, New York, NY, 223–236.

Received March 2019; revised August 2019; accepted September 2019