X-Containers: Breaking Down Barriers to Improve Performance and Isolation of Cloud-Native Containers

Zhiming Shen
Cornell University

Zhen Sun
Cornell University

Gur-Eyal Sela*
University of California, Berkeley

Eugene Bagdasaryan
Cornell University

Christina Delimitrou
Cornell University

Robbert Van Renesse
Cornell University

Hakim Weatherspoon
Cornell University

Abstract

"Cloud-native" container platforms, such as Kubernetes, have become an integral part of production cloud environments. One of the principles in designing cloud-native applications is called *Single Concern Principle*, which suggests that each container should handle a single responsibility well. In this paper, we propose X-Containers as a new security paradigm for isolating single-concerned cloud-native containers. Each container is run with a Library OS (LibOS) that supports multi-processing for concurrency and compatibility. A minimal exokernel ensures strong isolation with small kernel attack surface. We show an implementation of the X-Containers architecture that leverages Xen paravirtualization (PV) to turn Linux kernel into a LibOS. Doing so results in a highly efficient LibOS platform that does not require hardware-assisted virtualization, improves inter-container isolation, and supports binary compatibility and multi-processing. By eliminating some security barriers such as seccomp and Meltdown patch, X-Containers have up to 27× higher raw system call throughput compared to Docker containers, while also achieving competitive or superior performance on various benchmarks compared to recent container platforms such as Google's gVisor and Intel's Clear Containers.

*Work conducted at Cornell University.

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304016

CCS Concepts • **Security and privacy → Virtualization and security**; • **Software and its engineering → Operating systems**.

Keywords Containers; X-Containers; Cloud-Native; Library OS; exokernel

ACM Reference Format:
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 *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3297858.3304016

1 Introduction

An important recent trend in cloud computing is the rise of "cloud-native" container platforms, such as Kubernetes [38], which have become an integral part of production environments. Such platforms support applications designed specifically for cloud infrastructures that consist of loosely-coupled microservices [62] running in containers, enabling automatic orchestration and agile *DevOps* practices [33]. In cloud-native platforms, container design is similar to object design in object-oriented (OO) software systems: each container should have a single responsibility and handle that responsibility well [39]. By focusing on a single concern, cloud-native containers are easier to scale horizontally, and replace, reuse, and upgrade transparently. Similar to the *Single Responsibility Principle* in OO-languages, this has been termed the "Single Concern Principle" [51], and is recommended by Docker [8].

Running multiple containers on the same host does not come without problems. From a security perspective, if one container is compromised, all containers on the same Operating System (OS) kernel are put under risk. Due to the concern of application isolation, containers are generally not allowed to install their own kernel modules, a limitation for applications that require kernel customization. Nor can

the OS kernel be easily tuned and optimized for a particular container since it is shared by other containers.

There have been several proposals to address the issue of container isolation. Hypervisor-based container runtimes [17], such as Clear Containers [15], Kata Containers [16], and Hyper Containers [13], wrap containers with a dedicated OS kernel running in a virtual machine (VM). These platforms require hardware-assisted virtualization support to reduce the overhead of adding another layer of indirection. However, many public and private clouds, including Amazon EC2, do not support nested hardware virtualization. Even in clouds like Google Compute Engine where nested hardware virtualization is enabled, its performance overhead is high (Section 5 and [5]). LightVM [60] wraps a container in a paravirtualized Xen instance without hardware virtualization support. Unfortunately, it introduces a significant performance penalty in x86-64 platforms (Section 4.1 and 5). Finally, Google gVisor [12] is a user-space kernel written in Go that supports container runtime sandboxing, but it only offers limited system call compatibility [55] and incurs significant performance overheads (Section 5).

The trend of running a single application in its own VM for enhanced security has led to a renewed interest in LibOSes, as suggested by the Unikernel [58] model. LibOSes avoid the overhead of security isolation between the application and the OS, and allow each LibOS to be carefully optimized for the application at hand. Designing a container architecture inspired by the exokernel+LibOS [43] model can improve both container isolation and performance. However, existing LibOSes, such as MirageOS [58], Graphene [69], and OS^v [53], lack features like full binary compatibility or multi-processing support. This makes porting containerized applications very challenging.

In this paper, we propose a new LibOS platform called *X-Containers* that improves container isolation without requiring hardware virtualization support. An X-Container can support one or more user processes that all run at the same privilege level as the LibOS. Different processes inside an X-Container still have their own address spaces for resource management and compatibility, but they no longer provide secure isolation from one another; in this new security paradigm processes are used for concurrency, while X-Containers provide isolation between containers. We show an implementation of the X-Containers architecture that leverages Xen's paravirtualization (PV) architecture [32] and turns the Linux kernel into a LibOS that supports both binary compatibility and multi-processing.

Without hardware virtualization support, executing system call instructions is expensive, as they are first trapped into the exokernel and then redirected to the LibOS. The X-Container platform automatically optimizes the binary of an application during runtime to improve performance by rewriting costly system calls into much cheaper function calls in the LibOS. Furthermore, by avoiding overheads of seccomp filters and Meltdown [57] patch, X-Containers have up to 27× higher raw system call throughput compared to native Docker containers running in the cloud. X-Containers also achieve competitive or superior performance compared to recent container platforms such as gVisor and Clear Containers, as well as other LibOSes like Unikernel and Graphene on various benchmarks.

The X-Container architecture, however, also imposes several limitations. For example, the change in the threat model makes it unsuitable for running some containers that still require process and kernel isolation. Due to the requirement of running a LibOS with each container, X-Containers take longer time to boot and have bigger memory footprint. X-Containers also face challenges of page table operation efficiency and dynamic memory management. We discuss these limitations in the paper.

This paper makes the following contributions:

- We present X-Containers, a new exokernel-inspired container architecture that is designed specifically for single-concerned cloud-native applications. We discuss the new threat model and the trade-offs it introduces, the advantages and limitations of the proposed design, including those related to running unmodified applications in X-Containers.
- We demonstrate how the Xen paravirtualization architecture and the Linux kernel can be turned into a secure and efficient LibOS platform that supports both binary compatibility and multi-processing.
- We present a technology for automatically changing system calls into function calls to optimize applications running on a LibOS.
- We evaluate the efficacy of X-Containers against Docker, gVisor, Clear Container, and other LibOSes (Unikernel and Graphene), and demonstrate competitive or superior performance.

2 X-Containers as a New Security Paradigm

2.1 Single-Concerned Containers

Cloud-native applications are designed to fully exploit the potential of cloud infrastructures. Although legacy applications can be packaged in containers and run in a cloud, these applications cannot take full advantage of the automated deployment, scaling, and orchestration offered by systems like Kubernetes, which are designed for single-concerned containers [30, 31]. The shift to single-concerned containers is already apparent in many popular container clouds, such as Amazon Elastic Container Service (ECS), and Google Container Engine, both of which propose different mechanisms for grouping containers that need to be tightly coupled, e.g., using a "pod" in Google Kubernetes [21], and a "task" in Amazon ECS [7]. It is important to note that single-concerned

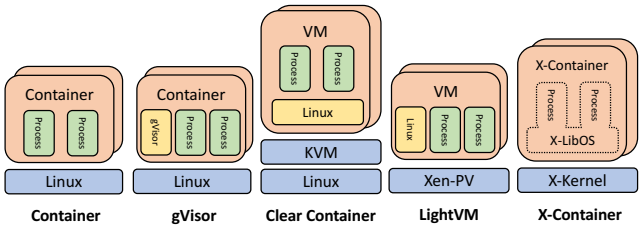


Figure 1. Comparison of different container architectures.

containers are not necessarily single-process. Many containers require to run bash scripts or third-party tools. Some applications might spawn multiple worker processes for concurrency [8], such as NGINX and Apache webserver.

2.2 Rethinking the Isolation Boundary

Modern operating systems (OS) that support multiple users and processes provide various types of isolation, including *Kernel Isolation*, which ensures that a process cannot compromise the integrity of the kernel nor read confidential information that is kept in the kernel; and *Process Isolation*, which ensures that one process cannot easily access or compromise another.

The cost of kernel isolation can be significant. System call handlers are forced to perform various security checks. The recently disclosed Meltdown attack [57] imposes kernel page-table isolation (PTI) [4], which further increases system call overheads. Moreover, data copies are often performed in the I/O stack for eliminating data dependencies between the kernel and user mode code. Meanwhile, there is a trend to push more and more functionality into the OS kernel, making it increasingly harder to defend against attacks on the kernel [46]. Modern monolithic OS kernels like Linux have become a large code base with complicated services, device drivers, and system call interfaces, resulting in a mounting number of newly discovered security vulnerabilities [19].

Process isolation is similarly problematic [49]. For one, this type of isolation typically depends on kernel isolation due to the way in which it is implemented and enforced. But perhaps more importantly, processes are not intended solely for security isolation. They are often used for resource sharing and concurrency support, and to support this modern OSes provide interfaces that transcend isolation, including shared memory, shared file systems, signaling, user groups, and debugging hooks. These mechanisms lay out a big attack surface, which causes many vulnerabilities for applications that rely on processes for security isolation.

In this work, we are revisiting the question of what functionality belongs in the kernel, and where the security boundaries should be built. An exokernel architecture [43] is essential for ensuring a minimal kernel attack surface while providing good performance. Beyond that, processes are useful for resource management and concurrency, but security isolation could be decoupled from the process model. Indeed, due to the coarse granularity of the process model, applications often implement their own security properties in the application logic, isolating different users within the same application. Many popular containerized applications (for instance Redis) mainly use either a single-threaded event-driven model or multi-threading instead of multi-processing for serving different clients. These applications implement client isolation inside the application logic through mechanisms such as a strongly-typed language runtime, role-based access control, authentication, and encryption.

In this paper, we propose the *X-Container* as a new paradigm for isolating single-concerned containers. In the X-Container architecture, each single-concerned container runs with its own LibOS called X-LibOS, and can have multiple processes—for resource management and concurrency—but not isolation. Inter-container isolation is guarded by the *X-Kernel*, an exokernel that ensures both a small kernel attack surface (*i.e.*, a small number of well-documented system calls) and a small Trusted Computing Base (TCB). The X-Containers architecture is different from existing container architectures, as shown in Figure 1. gVisor [12] has a user-space kernel isolated in its own address space. Clear Container [15] and LightVM [60] run each container in its own virtual machine (using KVM and Xen-PV resp.), but they don't reduce the cost of kernel and process isolation.

2.3 Threat Model and Design Trade-offs

X-Containers are single-concerned cloud-native containers that are either single-process, or use multiple processes for concurrency. Processes within the same X-Container are mutually trusting, and additionally trust the X-LibOS, and underlying X-Kernel. The most significant threat in this case comes from external probes designed to corrupt the application logic. This threat is countered by application and OS logic and is identical for standard containers and X-Containers. Another class of external threat may attempt to break through the isolation barrier of a container. In the case of standard containers, this isolation barrier is provided by the underlying general purpose OS kernel, which has a large TCB, and due to the large number of system calls, a large attack surface. X-Containers, in contrast, rely on a small X-Kernel that is specifically dedicated to providing isolation. The X-Kernel has a small TCB and a small number of system calls that lead to a smaller number of vulnerabilities in practice.

Running an application over a LibOS removes security isolation between a process and the OS kernel, but it does not affect other security mechanisms implemented in the application logic. For example, internal sand-boxing and protection are also possible by leveraging programming language type safety and verification tools for isolation, similar to Software-Isolated Processes [50], Software-based Fault Isolation [70], Nooks [68], and SPIN [37].

The X-Containers model involves multiple trade-offs. Intra-container isolation is significantly reduced for improving performance and inter-container isolation. While many cloud-native applications are compatible with X-Container's security model, there exist some applications that still rely on strong process and kernel isolation, and some widely-used security and fault-tolerance mechanisms are no longer working, for example:

- A regular OpenSSH server isolates different users in their own shell processes. Similarly, when using SSL certificates, NGINX stores them in the parent process running as root, isolated from worker processes running with fewer privileges. Running these applications directly in X-Containers cannot provide the same security guarantees.
- Applications using processes for fault tolerance expect that a crashed process does not affect others. However, in X-Containers, a crashed process might compromise the X-LibOS and the whole application.
- Kernel-supported security features such as seccomp, file permissions, and network filter are no longer effective in securing a particular thread or process within the container.

We can modify the application for some of these scenarios to work in X-Containers. For example, applications that still rely on process isolation and fault-tolerance can put each process in its own X-Container. However, this can increase overheads of inter-process communications.

3 X-Container Design

3.1 Challenges of Running Containers with LibOSes

Designing an exokernel-inspired container architecture can improve both container isolation and performance. However, there are two features that are necessary for supporting containers, but are particularly challenging for LibOSes:

- **Binary compatibility**: A container packages an application with all dependencies including third-party tools and libraries. A LibOS without binary level compatibility can make the porting of many containers infeasible. Even for containers that have the source code of all dependencies, substantial changes in implementation, compilation, debugging, and testing can potentially introduce security or compatibility issues that are not acceptable in production environments. Further, incompatibility causes the loss of opportunities to leverage existing, mature development and deployment infrastructures that have been optimized and tested for years.
- **Concurrent multi-processing**: While binary compatibility ensures support for spawning multiple processes, concurrent multi-processing refers to the capability of running multiple processes in different address spaces concurrently. As an example of the distinction, User Mode Linux (UML) [42] supports spawning multiple processes, but they can only run a single process at a time even when multiple CPU cores are available [24]. Without concurrent multi-processing, the performance of many applications would be dramatically impacted due to the reduced concurrency.

To the best of our knowledge, no existing LibOS provides both of these features. Unikernel [58] and related projects, such as Dune [35, 36], EbbRT [66], OS^v [53], and ClickOS [61], only support single-process applications, and involve substantial source code and compilation changes. Graphene [69] supports concurrent multiprocessing, but provides only one third of the Linux system calls.

3.2 Why Use Linux as the X-LibOS?

We believe that the best way to develop an X-LibOS that is fully compatible with Linux is to leverage Linux itself for the primitives needed in the LibOS. Starting from the Linux kernel when designing the X-LibOS enables binary compatibility and multiprocessing. Additionally, although the Linux kernel is widely referred to as a "general-purpose" OS kernel, in fact it is highly customizable and supports different layers of abstraction [52]. It has hundreds of booting parameters, thousands of compilation configurations, and many fine-grained runtime tuning knobs. Since most kernel functions can be configured as kernel modules and loaded during runtime, a customized Linux kernel can be very small and highly optimized. For example, for single-threaded applications, such as many event-driven applications, disabling Symmetric Multi-Processing (SMP) support can optimize spinlock operations [1], which greatly improves performance. Depending on the workload, applications can set different policies in the Linux scheduler [54]. Compared to user-level load balancers like HAProxy, kernel-level load balancing solutions based on IPVS (IP Virtual Server) have better performance and scalability [44]. Many applications do not currently reach the Linux kernel's full potential, either because of lack of control over kernel configurations, or because the kernel is shared across multiple diverse applications, complicating the process of tuning its many configuration parameters. Turning the Linux kernel into a LibOS and dedicating it to a single application can unlock its full potential.

3.3 Why Use Xen as the X-Kernel?

There have been previous attempts to turn an existing feature-rich monolithic OS kernel into a LibOS [64, 65]. However, these projects also use a monolithic OS kernel to serve as the host kernel. For example, the Linux Kernel Library (LKL) project [65] compiles the kernel code into an object file that can be linked directly into a Linux application. However, LKL does not support running multiple processes. This technical obstacle comes from the design choice of relying on the host kernel, instead of the LibOS itself, to handle page

Table 1. Comparison of Linux and Xen.

	Linux	Xen
Number of Syscalls or Hypercalls	300+	40+
Lines of Code	17+ Million	2+ Million
Number of CVE Reports in 2018	170	22
Number of CVE Reports in 2017	454	62

table mapping and scheduling. The same design choice was made by Drawbridge [34, 64], which turns the Windows 7 OS kernel into a LibOS running on Windows and only supports a single address space.

Graphene [69] is based on Linux, and addressed the challenge of supporting multiple processes by having processes use IPC calls to maintain the consistency of multiple LibOS instances, at a significant performance penalty. In addition, it is difficult for Graphene to support full compatibility with all Linux interfaces, such as shared memory, due to lack of control over memory mapping.

Rather than trying to run the Linux kernel as a LibOS inside a Linux process, X-Containers leverage the already mature support for running Linux in Xen's paravirtualization (PV) architecture [32] (Section 4.1). There are five reasons that make Xen ideal for implementing an X-Kernel with binary compatibility and concurrent multi-processing.

- *Compared to Linux, Xen is a much smaller kernel with simpler interfaces.* Although it is hard to directly compare the security of two different systems, the number of reports in Common Vulnerabilities and Exposures (CVE) [9] allow for an empirical comparison between the standard Linux kernel and Xen. Table 1 shows that the Linux kernel has an order of magnitude more vulnerabilities than Xen, roughly consistent with the ratio in code size and number of system calls.
- *Xen provides a clean separation of functions in kernel mode (Xen) and user mode (Linux).* In the Xen PV architecture, all operations that require root privileges are handled by Xen, while the Linux kernel is re-structured to run with fewer privileges. This clean separation eliminates the requirement of any hardware virtualization support.
- *Xen supports portability of guest kernels.* Xen hides the complexity of the underlying hardware, so that guest kernels only need to provide PV device drivers, which are portable across different platforms.
- *Multi-processing support is implemented in guest kernels.* Xen only provides facilities for managing page tables and context switching, while memory and process management policies are completely implemented in the guest kernel. This makes it much easier to support concurrent multi-processing when turning the guest kernel into a LibOS.

Table 2. Pros and cons of the X-Container architecture.

	Container	gVisor	Clear Container	LightVM	X-Container
Inter-container Isolation	Poor	Good	Good	Good	Good
System call Performance	Limited	Poor	Limited	Poor	Good
Portability	Good	Good	Limited	Good	Good
Compatibility	Good	Limited	Good	Good	Good
Intra-container Isolation	Good	Good	Good	Good	Reduced
Memory Efficiency	Good	Good	Limited	Limited	Limited
Spawning Time	Short	Short	Moderate	Moderate	Moderate
Software Licensing	Clean	Clean	Clean	Clean	Needs Discussion

- *There is a mature ecosystem around Xen infrastructures.* The Linux community maintains support for Xen PV architectures, which is critical for providing binary compatibility even for future versions of Linux. In addition, there are many mature technologies in Xen's ecosystem enabling features such as live migration, fault tolerance, and checkpoint/restore, which are hard to implement with traditional containers.

3.4 Limitations and Open Questions

In this paper, we focus on addressing some of the key challenges of turning the Xen PV architecture into an efficient X-Containers platform. As shown in Table 2, there are some challenges remaining for future work, as discussed below.

Memory management: In the Xen PV architecture, the separation of policy and mechanism in page table management greatly improves compatibility and avoids the overhead of shadow page table [2]. However, it still incurs overheads for page table operations (Section 5.4). This affects workloads like Apache webserver which frequently create and destroy memory mappings. Furthermore, the memory footprint of an X-Container is larger than a Docker container due to the requirement of running an X-LibOS (Section 5.7). To reduce memory footprint, multiple X-Containers can share read-only binary pages (supported since Xen 4.0) of X-LibOS, but it can be complicated when running different versions of X-LibOSes. Another challenge comes from dynamic memory management. In our prototype, each X-Container is configured with a static memory size. Dynamic memory allocation and over-subscription face problems of determining correct memory requirement and the efficiency of changing memory allocation with ballooning [48, 63].

Spawning speed of new instances: An important benefit of containers is that they can be spawned much faster than an ordinary VM. X-Containers require extra time for bootstrapping the whole software stack including the X-LibOS (Section 5.7). This overhead mainly comes from Xen's "xl" toolstack for creating new VMs. We are implementing a

new version of X-Containers integrated with the "runV" runtime [23] from HyperContainer [13], which can bring the toolstack overhead down to 460ms. LightVM has proposed a solution to further reduce this overhead to 4ms [60], which can be also applied to X-Containers.

GPL license contamination: The Linux kernel uses the GNU General Public License (GPL) [11], which requires that software using GPL-licensed modules must carry a license no less restrictive. While there exist no official rules on distinguishing two separate programs and one program with two parts, a widely used criteria is based on whether the components are linked together in a shared address space, and whether they communicate through function calls [26]. This raises concerns for running proprietary software in X-Containers. It is a question that deserves more discussion, but we believe that GPL contamination should not be applied to software running in X-Containers if the identical software binary is legally viable when running on ordinary Linux systems.

4 Implementation

We have implemented a prototype of the X-Containers platform based on Xen 4.2 and Linux kernel 4.4.44. We leveraged Xen-Blanket [71] drivers to run the platform efficiently in public clouds. We focused on applications running in x86-64 long mode. For Linux, the patch included 264 lines of C code and 1566 lines of assembly, which are mostly in the architecture-dependent layer and transparent to other layers in the kernel. For Xen, the C code patch size was 686 lines and the assembly code patch size was 153 lines. In this section, we present the implementation of X-Containers.

4.1 Background: Xen Paravirtualization

The Xen PV architecture enables running multiple concurrent Linux VMs (PV guests) on the same physical machine without support for hardware-assisted virtualization, but it requires guest kernels to be modestly modified to work with the underlying hypervisor. Below, we review key technologies in Xen's PV architecture and its limitations on x86-64 platforms.

In the PV architecture, Xen runs in kernel mode, and both the host OS (a.k.a "Domain-0") and guest OS (a.k.a Domain-U) run with fewer privileges. All sensitive system instructions that could affect security isolation, such as installing new page tables and changing segment selectors, are executed by Xen. Guest kernels request those services via hypercalls, which are validated by Xen before being served. Exceptions and interrupts are virtualized through efficient event channels. For device I/O, instead of emulating hardware, Xen defines a simpler split driver model. The Domain-U installs a front-end driver, which is connected to a corresponding back-end driver in the Driver Domain which gets access to real hardware, and data is transferred using shared memory (asynchronous buffer descriptor rings). Importantly, while Domain-0 runs a Linux kernel and has the supervisor privilege to control other domains, it does not run any applications, and can effectively isolate device drivers in unprivileged Driver Domains. Therefore, bugs in Domain-0 kernel are much harder to exploit, and in their majority do not affect security isolation of other VMs.

Xen's PV interface has been supported by the mainline Linux kernel—it was one of the most efficient virtualization technologies on x86-32 platforms. However, the PV architecture faces a fundamental challenge on x86-64 platforms. Due to the elimination of segment protection in x86-64 long mode, we can only run the guest kernel and user processes in user mode. To protect the guest kernel from user processes, the guest kernel needs to be isolated in another address space. Each system call needs to be forwarded by the Xen hypervisor as a virtual exception, and incurs a page table switch and a TLB flush. This causes significant overheads, and is one of the main reasons why 64-bit Linux VMs opt to run with hardware-assisted full virtualization instead of PV.

4.2 Eliminating Kernel Isolation

We modified the application binary interface (ABI) of the Xen PV architecture so that it no longer provides isolation between the guest kernel (i.e., the X-LibOS) and user processes. X-LibOS is mapped into user processes' address space with the same page table privilege level and segment selectors, so that kernel access no longer incurs a switch between (guest) user mode and (guest) kernel mode, and system calls can be performed with function calls.

This leads to a complication: Xen needs to know whether the CPU is in guest user mode or guest kernel mode for correct syscall forwarding and interrupt delivery. Since all user-kernel mode switches are handled by Xen, this can easily be done via a flag. However, in X-LibOS, with lightweight system calls (Section 4.4) guest user-kernel mode switches do not involve the X-Kernel anymore. Instead, the X-Kernel determines whether the CPU is executing kernel or user process code by checking the location of the current stack pointer. As in the normal Linux memory layout, X-LibOS is mapped into the top half of the virtual memory address space and is shared by all processes. The user process memory is mapped to the lower half of the address space. Thus, the most significant bit in the stack pointer indicates whether it is in guest kernel mode or guest user mode.

In the Xen PV architecture, interrupts are delivered as asynchronous events. There is a variable shared by Xen and the guest kernel that indicates whether there is any event pending. If so, the guest kernel issues a hypercall into Xen to have those events delivered. In the X-Container architecture, the X-LibOS can emulate the interrupt stack frame when it sees any pending events and jump directly into interrupt handlers without trapping into the X-Kernel first.

To return from an interrupt handler, one typically uses the `iret` instruction to reset code and stack segments, flags, the stack and instruction pointers, while also atomically enabling interrupts. However, in the Xen PV architecture virtual interrupts can only be enabled by writing to a memory location, which cannot be performed atomically with other operations. To guarantee atomicity and security when switching privilege levels, Xen provides a hypercall for implementing `iret`. To implement `iret` completely in user mode, the code must support reentrancy so that virtual interrupts can be enabled before restoring all registers.

We implement `iret` by considering two cases. When returning to a place running on the kernel mode stack, the X-LibOS pushes return address on the destination stack, and switches the stack pointer before enabling interrupts so preemption is safe. Then the code jumps to the return address by using a lightweight `ret` instruction. When returning to the user mode stack, the user mode stack pointer might not be valid, so X-LibOS saves register values in the kernel stack for system call handling, enables interrupts, and then executes the more expensive `iret` instruction.[1] Similar to `iret`, the `sysret` instruction, which is used for returning from a system call handler, is optimized without trapping in the kernel by leveraging spare registers (`rcx` and `r11`) which are clobbered according to the `syscall` ABI.

4.3 Concurrent Multi-Processing Support

X-Containers inherit support for concurrent multiprocessing from the Xen PV architecture. Xen provides an abstraction of paravirtualized CPUs, and the Linux kernel can leverage this abstraction in the architecture-dependent layer using customized code for handling interrupts, maintaining page tables, flushing TLBs, etc. The Linux kernel has full control over how processes are scheduled with virtual CPUs, and Xen determines how virtual CPUs are mapped to physical CPUs for execution.

For security isolation, in paravirtualized Linux the "global" bit in the page table is disabled so that switching between different processes causes a full TLB flush. This is not needed for X-LibOS, thus the mappings for the X-LibOS and X-Kernel both have the global bit set in the page table. Switching between different processes running on the same X-LibOS (by updating the `CR3` register) does not trigger a full TLB flush, which greatly improves the performance of address translation. When switching between different X-Containers, a full TLB flush is performed by updating the `CR4` register to drop all TLB entries regardless of the global bit.

Because the kernel code is no longer protected, kernel routines would not need a dedicated stack if the X-LibOS only supported a single process. However, since the X-LibOS supports multiple forked processes with overlapping address spaces, using user-mode stack for kernel routines causes problems after context switch. Therefore, we still use dedicated kernel stacks in the kernel context, and when performing a system call, a switch from user stack to kernel stack is necessary.

4.4 Automatic Lightweight System Calls

In the x86-64 architecture, user mode programs perform system calls using the `syscall` instruction, which transfers control to a routine in kernel mode. The X-Kernel immediately transfers control to the X-LibOS, guaranteeing binary level compatibility so that existing applications can run on the X-LibOS without any modification.

Because the X-LibOS and the process both run in the same privilege level, it is more efficient to invoke system call handlers using function call instructions. X-LibOS stores a *system call entry table* in the `vsyscall` page, which is mapped to a fixed virtual memory address in every process. Updating X-LibOS will not affect the location of the system call entry table. Using this entry table, applications can optimize their libraries and binaries for X-Containers by patching the source code to change system calls into function calls, as most existing LibOSes do. However, this significantly increases deployment complexity, and it cannot handle third-party tools and libraries whose source code is not available.

To avoid re-writing or re-compiling the application, we implemented an online Automatic Binary Optimization Module (ABOM) in the X-Kernel. It automatically replaces `syscall` instructions with function calls on the fly when receiving a syscall request from user processes, avoiding scanning the entire binary file. Before forwarding the syscall request, ABOM checks the binary around the syscall instruction and sees if it matches any pattern that it recognizes. If it does, ABOM temporarily disables interrupts and enables writing to any memory page even if it is mapped read-only in the page table. ABOM then performs the binary patch with atomic `lock cmpxchg` instructions[2]. Since each `cmpxchg` instruction can handle at most eight bytes, if we need to modify more than eight bytes, we need to make sure that any intermediate state of the binary is still valid for the sake of multicore concurrency safety. The patch is mostly transparent to X-LibOS, except that the page table dirty bit will be set for read-only pages. X-LibOS can choose to either ignore those dirty pages, or flush them to disk so that the same patch is not needed in the future.

Figure 2 illustrates three patterns of binary code that ABOM recognizes. To perform a system call, programs typically set the system call number in the `rax` or `eax` register with a `mov` instruction, and then execute the `syscall` instruction. The `syscall` instruction is two bytes, and the `mov` instruction is 5 or 7 bytes depending on the size of operands.

[1] A potential stack overflow can happen, which we leave for future work.

[2] This unsynchronized cross-modifying code (XMC) can potentially cause some non-deterministic behavior [3], which we will address in future work.

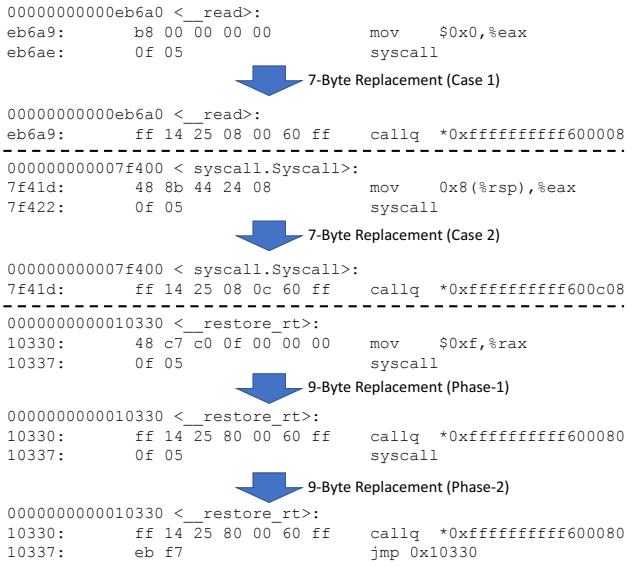


Figure 2. Examples of binary replacement.

We replace these two instructions with a single `call` instruction with an absolute address stored in memory, which can be implemented with 7 bytes. The memory address of the entry points is retrieved from the system call entry table stored in the `vsyscall` page. The binary replacement only needs to be performed once for each place.

With 7-byte replacements, we merge two instructions into one. There is a rare case that the program jumps directly to the location of the original `syscall` instruction after setting the `rax` register somewhere else or after an interrupt. After the replacement, this will cause a jump into the last two bytes of our `call` instruction, which are always "`0x60 0xff`". These two bytes cause an invalid opcode trap into the X-Kernel. To provide binary level equivalence, we add a special trap handler in the X-Kernel to redirect the program to the system call handler.

9-byte replacements are performed in two phases, each one generating results equivalent to the original binary. Since the `mov` instruction takes 7 bytes, we replace it directly with a call into the syscall handler. We leave the original syscall instruction unchanged, in case the program jumps directly to it, and we further optimize it with a jump into the previous `call` instruction. The syscall handler in X-LibOS will check if the instruction on the return address is either a `syscall` or a specific `jmp` to the `call` instruction again. If it is, the syscall handler modifies the return address to skip this instruction.

Our online binary replacement solution only handles the case when the `syscall` instruction immediately follows a `mov` instruction. For more complicated cases, it is possible to inject code into the binary and re-direct a bigger chunk of code. We also provide a tool to do this offline. For most standard libraries, such as `glibc`, the default system call wrappers typically use the pattern illustrated in Figure 2, making our current solution sufficient for optimizing most system call wrappers on the critical path (see evaluations in Section 5.2).

4.5 Lightweight Bootstrapping of Docker Images

X-Containers do not have a VM disk image and do not go through the same bootstrapping processes that a VM does. When creating a new X-Container, a special bootloader is loaded with an X-LibOS, which initializes virtual devices, configures IP addresses, and spawns processes of the container directly. Because X-Containers support binary level compatibility, we can run any existing Docker image without modification. We connect our X-Container architecture to the Docker platform with a *Docker Wrapper*. An unmodified Docker engine running in the Host X-Container is used to pull and build Docker images. We use `devicemapper` as the storage driver, which stores different layers of Docker images as thin-provisioned copy-on-write snapshot devices. The Docker Wrapper then retrieves the meta-data from Docker, creates the thin block device and connects it to a new X-Container. The processes in the container are then spawned with a dedicated X-LibOS. We are also integrating X-Containers with the "runV" runtime [23], which supports Open Container Initiative (OCI) and can run as a standard backend for Docker and Kubernetes.

5 Evaluation

In this section, we address the following questions:

- How effective is the Automatic Binary Optimization Module (ABOM)?
- What is the performance overhead of X-Containers, and how does it compare to Docker and other container runtimes in the cloud?
- How does the performance of X-Containers compare to other LibOS designs?
- How does the scalability of X-Containers compare to Docker Containers and VMs?

5.1 Experiment Setup

We conducted experiments on VMs in both Amazon Elastic Compute Cloud (EC2) and Google Compute Engine (GCE). In EC2, we used `c4.2xlarge` instances in the North Virginia region (4 CPU cores, 8 threads, 15GB memory, and 2×100GB SSD storage). To make the comparison fair and reproducible, we ran the VMs with different configurations on a dedicated host. In Google GCE, we used a customized instance type in the South Carolina region (4 CPU cores, 8 threads, 16GB memory, and 3×100GB SSD storage). Google does not support dedicated hosts, so we attached multiple boot disks to a single VM, and rebooted it with different configurations.

We used the Docker platform on Ubuntu 16.04 and gVisor as baselines for our evaluation. Docker containers were

Table 3. Evaluation of the Automatic Binary Optimization Module (ABOM).

Application	Description	Implementation	Benchmark	Syscall Reduction
memcached	Memory caching system	C/C++	memtier_benchmark	100%
Redis	In-memory database	C/C++	redis-benchmark	100%
etcd	Key-value store	Go	etcd-benchmark	100%
MongoDB	NoSQL Database	C/C++	YCSB	100%
InfluxDB	Time series database	Go	influxdb-comparisons	100%
Postgres	Database	C/C++	pgbench	99.80%
Fulentd	Data collector	Ruby	fluentd-benchmark	99.40%
Elasticsearch	Search engine	JAVA	elasticsearch-stress-test	98.80%
RabbitMQ	Message broker	Erlang	rabbitmq-perf-test	98.60%
Kernel Compilation	Code Compilation	Various tools	Linux kernel with tiny config	95.30%
Nginx	Webserver	C/C++	Apache ab	92.30%
MySQL	Database	C/C++	sysbench	44.60% (92.2% manual)

run with the default seccomp filters. `gVisor` used the default `Ptrace` platform in Amazon, and the KVM platform in Google with nested virtualization. In Google GCE, we also installed Clear Containers in Ubuntu 16.04 with KVM. We also implemented `Xen-Containers`, a platform similar to LightVM [60] that packages containers with a Linux kernel in para-virtualized Xen instances. Xen-Containers use exactly the same software stack (including the Domain-0 tool stack, device drivers, and Docker wrapper) as X-Containers. The only difference between Xen-Containers and X-Containers is the underlying hypervisor (unmodified Xen vs. X-Kernel) and guest kernel (unmodified Linux vs. X-LibOS). Xen-Containers are similar to Clear Containers except that they can run in public clouds that do not support nested hardware virtualization, such as Amazon EC2.

Due to the disclosure of Meltdown attacks on Intel CPUs, both Amazon EC2 and Google GCE provision VMs with patched Linux kernels by default. This patch protects the kernel by isolating page tables used in user and kernel mode. The same patch exists for Xen and we ported it to both Xen-Container and X-Container. These patches can cause significant overheads, and ultimately new Intel hardware will render them unnecessary. It is thus important to compare both the patched and unpatched code bases. We therefore used ten configurations: `Docker`, `Xen-Container`, `X-Container`, `gVisor`, and `Clear-Container`, each with an `-unpatched` version. Due to the threat model of single-concerned containers, for `Clear-Containers` only the host kernel is patched; the guest kernel running in nested VMs is unpatched in our setup.

The VMs running native Docker, gVisor, and Clear Containers had Ubuntu 16.04-LTS installed with Docker engine 17.03.0-ce and Linux kernel 4.4. We used Linux kernel 4.14 as the guest kernel for Clear Containers since its current tool stack is no longer compatible with Linux 4.4. The VMs running Xen-Containers had CentOS-6 installed as Domain-0 with Docker engine 17.03.0-ce and Xen 4.2, and used Linux kernel 4.4 for running containers. X-Containers used the same setup as Xen-Containers except that we modified Xen and Linux as described in this paper. All configurations used `device-mapper` as the back-end storage driver.

For each set of experiments, we used the same Docker image for all configurations. When running network benchmarks, we used separate VMs for the client and server. Unless otherwise noted we report the average and standard deviation of five runs for each experiment.

5.2 Automatic Binary Optimization

To evaluate the efficacy of ABOM, we added a counter in the X-Kernel to calculate how many system calls were forwarded to X-LibOS. We then ran a wide range of popular container applications with ABOM enabled and disabled. The applications include the top 10 most popular containerized applications [41], and are written in a variety of programming languages. For each application, we used open-source workload generators as the clients.

Table 3 shows the applications we tested and the reduction in system call invocations that ABOM achieved. For all but one application we tested, ABOM turns more than 92% of system calls into function calls. The exception is MySQL, which uses cancellable system calls implemented in the `libpthread` library that are not recognized by ABOM. However, using our offline patching tool, two locations in the `libpthread` library can be patched, reducing system call invocations by 92.2%.

5.3 Macrobenchmarks

We evaluated the performance of X-Containers with four macrobenchmarks: NGINX, Memcached, Redis, and Apache httpd. The corresponding Docker images were `nginx:1.13`, `memcached:1.5.7`, `redis:3.2.11`, and `httpd:2.4`, with the default configurations. For X-Containers the applications were optimized only by ABOM, without any manual binary patching. Since Amazon EC2 and Google GCE do not support

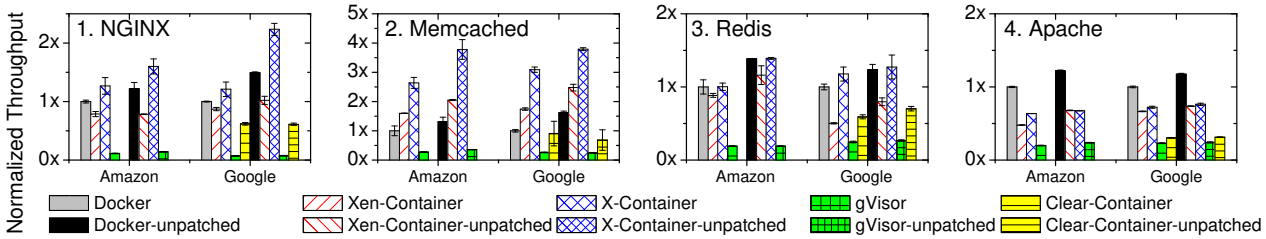


Figure 3. Normalized performance of macrobenchmarks (higher is better).

bridged networks natively, the servers were exposed to clients via port forwarding in `iptables`. We used a separate VM as the client for generating workloads. For NGINX and Apache we used the `Apache ab` benchmark which benchmarks webserver throughput by sending concurrent requests. For Memcached and Redis, we used the `memtier_benchmark` which simulates multiple clients generating operations to the database with a 1:10 SET:GET ratio.

Figure 3 shows the relative performance of the macrobenchmarks normalized to native `Docker` (patched). gVisor performance suffers significantly from the overhead of using `ptrace` for intercepting system calls. Clear Containers and gVisor in Google suffer a significant performance penalty for using nested hardware virtualization (also measured by Google [5]). X-Containers improve throughput of Memcached from 134% to 208% compared to native Docker, achieving up to 307K Ops/sec in Amazon and 314K Ops/sec in Google. For NGINX, X-Containers achieve 21% to 50% throughput improvement over Docker, up to 32K Req/sec in Amazon and 40K Req/sec in Google. For Redis, the performance of X-Containers is comparable to Docker, about 68K Ops/sec in Amazon and 72K Ops/sec in Google, but note that this is achieved with stronger inter-container isolation. For Apache, X-Containers incur 28% to 45% performance overhead (11K Req/sec in Amazon and 12K Req/sec in Google), which is caused by context switches between multiple worker processes. Note that Xen-Containers performed worse than Docker in most cases, thus performance gains achieved by X-Containers are due to our modifications to Xen and Linux.

5.4 Microbenchmarks

To better understand the gains and losses in performance, we also evaluated X-Containers with a set of microbenchmarks. We ran `UnixBench` and `iperf` in the Ubuntu 16.04 container. The *System Call* benchmark tests the speed of issuing a series of nonblocking system calls, including `dup`, `close`, `getpid`, `getuid`, and `umask`. The *Execl* benchmark measures the speed of the `exec` system call, which overlays a new binary on the current process. The *File Copy* benchmarks test the throughput of copying files with a 1KB buffer. The *Pipe Throughput* benchmark measures the throughput

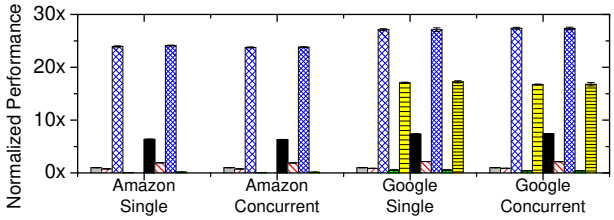


Figure 4. Normalized system call throughput (higher is better; legend is the same as Figure 3).

of a single process reading and writing in a pipe. The *Context Switching* benchmark tests the speed of two processes communicating with a pipe. The *Process Creation* benchmark measures the performance of spawning new processes with the `fork` system call. Finally, *iperf* tests the performance of TCP transfer. We ran our tests both in Google GCE and Amazon EC2. We ran tests both isolated and concurrently. For concurrent tests, we ran 4 copies of the benchmark simultaneously. For each configuration, we saw similar trends.

Figure 4 shows the relative system call throughput normalized to `Docker`. X-Containers dramatically improve system call throughput (up to 27× compared to `Docker`, and up to 1.6× compared to `Clear Containers`). This is because X-Containers avoid the overhead of seccomp filters and Meltdown patch for Docker, and the overhead of nested virtualization for Clear Containers. The throughput of gVisor is only 7 to 55% of Docker due to the high overhead of `ptrace` and nested virtualization, so can be barely seen in the figure. Clear Containers achieve better system call throughput than Docker because the guest kernel is optimized by disabling most security features within a Clear container. Also, note that the Meltdown patch does not affect performance of X-Containers and Clear Containers because for X-Containers the system calls do not trap into kernel mode, and for Clear Containers the guest kernel is always unpatched.

Figure 5 shows the relative performance for other microbenchmarks, also normalized to patched `Docker`. Similar to the system call throughput benchmark, the Meltdown patch does not affect X-Containers and Clear Containers. In contrast, patched Docker containers and Xen-Containers suffer significant performance penalties. X-Containers have

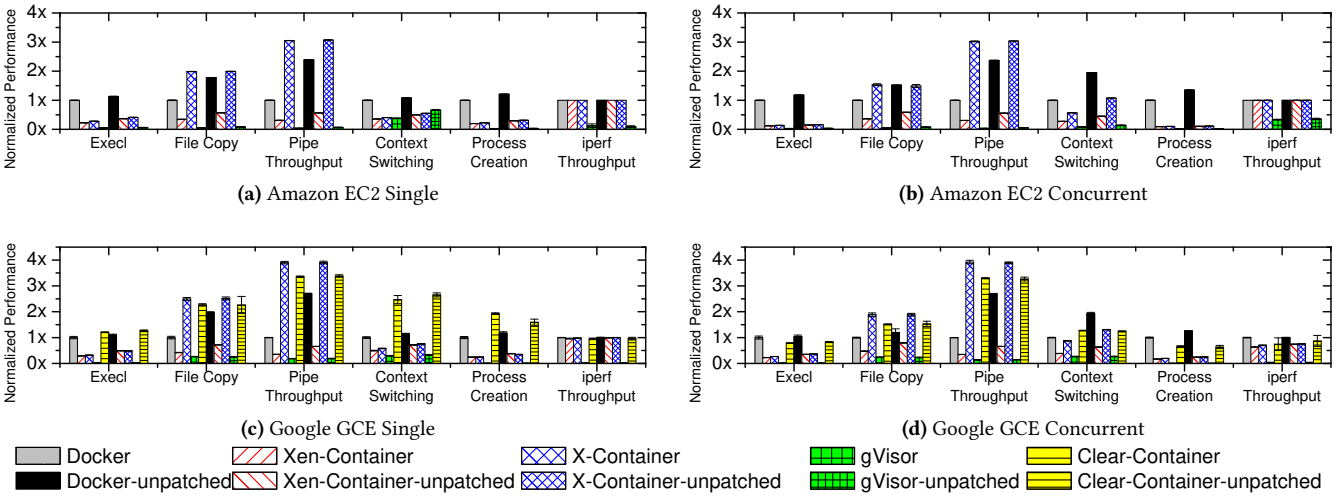


Figure 5. Normalized performance of microbenchmarks (higher is better).

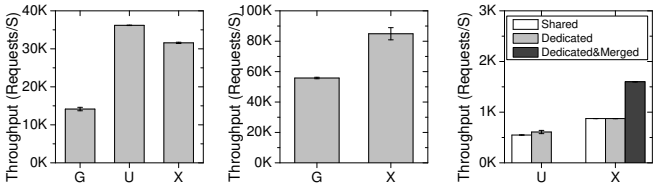


Figure 6. Throughput comparison for Unikernel (U), Graphene (G), and X-Container (X).

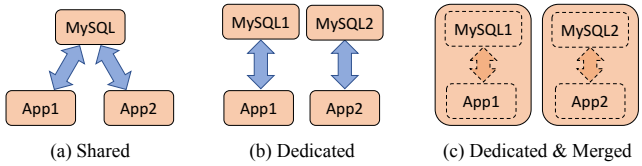


Figure 7. Alternate configurations of two applications that use MySQL.

noticeable overheads in process creation and context switching. This is because these benchmarks involve many page table operations that must be done in the X-Kernel.

5.5 Unikernel and Graphene

We also compared X-Containers to Graphene and Unikernel. For these experiments, we used four Dell PowerEdge R720 servers in our local cluster (two 2.9 GHz Intel Xeon E5-2690 CPUs, 16 cores, 32 threads, 96GB memory, 4TB disk), connected to one 10Gbit switch. We ran the `wrk` benchmark with the NGINX webserver, PHP, and MySQL. Graphene ran on Linux with Ubuntu-16.04, and was compiled *without* the security isolation module (which should improve its performance). For Unikernel, we used *Rumprun* [22] because it can run the benchmarks with minor patches (running with MirageOS [58] requires rewriting the application in OCaml).

Figure 6a compares throughput of the NGINX webserver serving static webpages with a single worker process. As there was only one NGINX server process running, we dedicated a single CPU core for X-Containers and Unikernel. X-Containers achieve throughput comparable to Unikernel, and over twice that of Graphene.

For Figure 6b, we ran 4 worker processes of a single NGINX webserver. This is not supported by Unikernel, so we only compared with Graphene. X-Containers outperform Graphene by more than 50%, since in Graphene, processes use IPC calls to coordinate access to a shared POSIX library, which incurs high overheads.

For Figure 6c we evaluated the scenario where two PHP CGI servers were connected to MySQL databases. We enabled the built-in webserver of PHP, and used the `wrk` client to access a page that issued requests to the database with equal probability for read and write. Graphene does not support the PHP CGI server, so we only compared to Unikernel. As illustrated in Figure 7, the PHP servers can have either shared or dedicated databases, so there are three possible configurations depending on the threat model. Figure 6c shows the total throughput of two PHP servers with different configurations. All VMs were running a single process with one CPU core. With `Shared` and `Dedicated` configurations, X-Containers outperform Unikernel by over 40%. Furthermore, X-Containers support running PHP and MySQL in a single container (the `Dedicated & Merged` configuration), which is not possible for Unikernel. Using this setup, X-Container throughput is about three times that of the Unikernel `Dedicated` configuration.

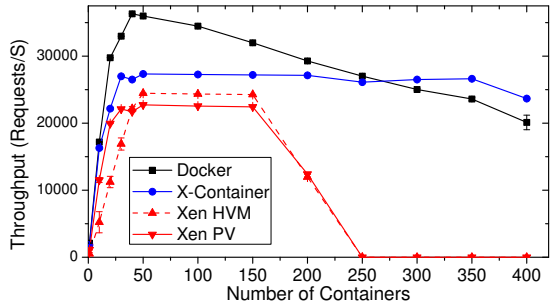
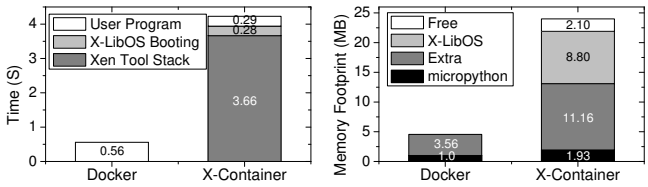


Figure 8. Throughput scalability.



(a) Spawning Time **(b)** Memory Footprint

Figure 9. Spawning time and memory footprint.

5.6 Scalability

We evaluated scalability of the X-Containers architecture by running up to 400 containers on one physical machine. For this experiment, we used an NGINX server with a PHP-FPM engine. We used the `webdevops/PHP-NGINX` Docker image and configured NGINX and PHP-FPM with a single worker process. We ran the `wrk` benchmark to measure the total throughput of all containers. Each container had a dedicated `wrk` thread with 5 concurrent connections—thus the total number of `wrk` threads and concurrent connections increased linearly with the number of containers.

Each X-Container was configured with 1 vCPU and 128MB memory. We also evaluated `Xen HVM` and `Xen PV` configurations that ran Docker containers in regular Xen HV and PV instances respectively. Each Xen VM had 1 vCPU and 512MB memory. Note that 512MB is the recommended minimum memory size for Ubuntu 16.04. When using smaller memory size like 256MB, the VMs can still boot, but the network starts dropping packets. Since the physical machine only had 96GB memory, we can run 200 VMs at most.

Figure 8 shows the aggregated throughput of all containers or VMs. `Docker` achieves higher throughput for small numbers of containers since context switching between Docker containers is cheaper than between X-Containers and between Xen VMs. However, as the number of containers increases, the performance of `Docker` drops faster. This is because each NGINX+PHP container runs 4 processes: with N containers, the Linux kernel running Docker containers is scheduling $4N$ processes, while X-Kernel is scheduling N vCPUs, each running 4 processes. This hierarchical scheduling turns out to be a more scalable way of co-scheduling many containers. With $N = 400$, `X-Containers` outperform `Docker` by 18%. Note that in this experiment, we avoided over-subscribing memory. If memory becomes a bottleneck, due to the lack of dynamic memory allocation, X-Containers will have higher overhead than Docker.

5.7 Spawning Time and Memory Footprint

We evaluated the overhead of X-Containers on spawning time and memory footprint, comparing to the same version of Docker engine as we used for X-Containers running on an unmodified Linux kernel. These experiments were performed in Amazon EC2 `c4.2xlarge` instances.

To measure the spawning time, we instrumented the system to record timestamps of some critical events. We then spawned a Ubuntu 16.04 container and ran the `date` command to print out a timestamp as the point of finishing spawning the container. Figure 9a shows the detailed breakdown of the time spent on different phases when spawning a new container. Docker takes 558ms to finish spawning, which we counted as all for "User Program" although some of them is actually spent in kernel for security and resource configurations. For X-Containers, it takes 277ms to boot a new X-LibOS, and another 287ms for spawning the user program. We spawn the user program faster than Docker since we skip most of the security and resource configurations. However, the overhead of Xen's "xl" toolstack adds another 3.66 seconds to the total spawning time. We are integrating X-Containers with the "runV" runtime [23] which brings the toolstack overhead down to 460ms.

To evaluate the memory footprint, we used the `micropython` container, which runs a python script to check memory utilization from the `/proc` file system. Figure 9b shows the breakdown of memory usage. The micropython process itself consumes 1 to 2 MB memory. However, the `docker stats` command reports 3.56MB extra memory consumption (counted from cgroups) used for page cache and container file system. For X-Containers, the "Extra" 11.16MB memory includes page cache for the whole system and all other user processes such as the initial `bash` and `switch_root` tools. The 8.8MB memory for "X-LibOS" includes kernel code and text segments, kernel stack, page tables, and slab memory. Note that there is some "free" memory in an X-Container that can be reduced by ballooning, but cannot be totally eliminated due to the minimal requirement of the Linux kernel.

6 Related Work

6.1 Application Containers

OS-level virtualization [67] provides a lightweight mechanism of running multiple OS instances. Docker [10], LXC [18], OpenVZ [20], and Solaris Zones [6] are different

implementations of OS-level virtualization. Generally, these solutions provide poor kernel customization support, and application isolation is a concern due to the sharing of a large OS kernel. Although there are mitigations such as `seccomp` and `SELinux` which allow specification of system call filters for each container, in practice it is difficult to define a good policy for arbitrary, previously unknown applications [12].

Various runtimes have been proposed to address the problem of security isolation in containers. Clear Containers [15], Kata Containers [16], Hyper Containers [13], VMware vSphere Integrated Containers [25], and Hyper-V containers [14] all leverage hardware virtualization support to wrap containers with a dedicated OS kernel running in a VM. However, deploying these platforms in virtualized clouds requires nested hardware virtualization support, which is not available everywhere, and can cause significant performance penalties even when it is available [5]. Google gVisor [12] is a user-space kernel written in Go that supports container runtime sandboxing, but it provides limited compatibility [55] and incurs significant performance overhead.

LightVM with TinyX [60] creates minimalistic Linux VM images targeted at running a single application container. Similar to X-Containers, LightVM leverages the Xen hypervisor to reduce the TCB running in kernel mode, and can leverage Xen-Blanket [71] to run in public clouds. However, this can introduce significant performance overheads, as we saw in Section 5. LightVM focuses on improving Xen's toolstack for scalability and performance, which can be integrated with X-Containers.

SCONE [29] implements secure containers using Intel SGX, assuming a threat model different from X-Container's where even the host OS or hypervisor cannot be trusted. Due to hardware limitations, SCONE cannot support full binary compatibility and multi-processing.

6.2 Library OS

The insight of a Library OS [28, 40, 43, 56, 64] is to keep the kernel small and link applications to a LibOS containing functions that are traditionally performed in the kernel. Most Library OSes [27, 43, 47, 64, 65] focus exclusively on single-process applications, which is not sufficient for multi-process container environments, and cannot support more complicated cloud applications that rely on Linux's rich primitives. Graphene [69] is a Library OS that supports multiple Linux processes, but provides only one third of the Linux system calls. Moreover, multiple processes use IPC calls to access a shared POSIX implementation, which limits performance and scalability. Most importantly, the underlying host kernel of Graphene is a full-fledged Linux kernel, which does not reduce the TCB and attack surface.

Unikernel [58] and related projects, such as EbbRT [66], OS^v [53], ClickOS [61], and Dune [35, 36], proposed compiling an application with a Library OS into a lightweight VM, using the VM hypervisor as the exokernel. These systems also only support single-process applications, and require re-writing or re-compiling the application. In contrast, X-Containers supports binary level compatibility and multiple processes. In addition, X-Container supports all debugging and profiling features that are available in Linux.

Usermode Kernel [45] is an idea similar to X-Containers that runs parts of the Linux kernel in userspace in VMs. However, some parts of the Usermode Kernel still run in a higher privilege level than user mode processes, and it is not integrated with application container environments. Moreover, Usermode Kernel currently only works for x86-32 architectures. Kernel Mode Linux (KML) [59] allows executing user programs inside kernel space. However, KML requires hardware virtualization in order to isolate different applications.

As a final point, none of the previous work on containers and LibOSes are specifically designed for cloud-native applications, either incurring high performance overheads, or sacrificing compatibility. As more applications switch from monolithic designs to large graphs of loosely-coupled microservices, it is important for container technologies to also evolve to fully exploit the potential of cloud-native systems.

7 Conclusion

In this paper, we propose X-Containers as a new security paradigm for isolating single-concerned cloud-native containers: minimal exokernels can securely isolate mutually untrusting containers, and LibOSes allow for customization and performance optimization. X-Containers introduce new trade-offs in container design: intra-container isolation is significantly reduced for improving performance and inter-container isolation. We demonstrate an implementation of the X-Containers architecture that uses Xen as the X-Kernel and Linux as the X-LibOS, achieving binary compatibility and concurrent multi-processing without the requirement of hardware virtualization support. We show that X-Containers offer significant performance improvements in cloud environments, and discuss the advantages and limitations of the current design, including those pertaining to running unmodified applications in X-Containers.

Availability

The code of the X-Containers project is publicly available at http://x-containers.org.

Acknowledgments

We thank our anonymous reviewers and shepherd, Nadav Amit, for their useful comments and suggestions. This work was in part supported by the National Science Foundation (NSF) CSR-1422544, NSF NeTS CSR-1704742, the U.S. Department of Commerce, National Institute of Standards and Technology (NIST) award number 70NANB17H181, Cisco, and Huawei.

References

[1] 2005. SMP alternatives. https://lwn.net/Articles/164121/

[2] 2015. X86 Paravirtualised Memory Management. https://wiki.xen.org/wiki/X86_Paravirtualised_Memory_Management

[3] 2016. Intel 64 and IA-32 Architectures Software Developer's Manual. *Volume 3: System programming Guide* (2016).

[4] 2018. Kernel page-table isolation. https://en.wikipedia.org/wiki/Kernel_page-table_isolation

[5] 2018. Performance of Nested Virtualization – Google Compute Engine. https://cloud.google.com/compute/docs/instances/enable-nested-virtualization-vm-instances#performance

[6] 2018. Solaris Containers. https://en.wikipedia.org/wiki/Solaris_Containers

[7] 2019. Amazon ECS Task Definitions. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task_definitions.html

[8] 2019. Best practices for writing Dockerfiles. https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

[9] 2019. CVE: list of cybersecurity vulnerabilities. https://cve.mitre.org/

[10] 2019. Docker. https://www.docker.com/

[11] 2019. GNU General Public License. https://www.gnu.org/copyleft/gpl.html

[12] 2019. gVisor: Container Runtime Snadbox. https://github.com/google/gvisor

[13] 2019. Hyper Containers. https://hypercontainer.io

[14] 2019. Hyper-V Containers. https://docs.microsoft.com/en-us/virtualization/windowscontainers/manage-containers/hyperv-container

[15] 2019. Intel Clear Containers. https://clearlinux.org/containers

[16] 2019. Kata Containers. https://katacontainers.io

[17] 2019. Kubernetes Frakti. https://github.com/kubernetes/frakti

[18] 2019. Linux LXC. https://linuxcontainers.org/

[19] 2019. List of Security Vulnerabilities in the Linux Kernel. https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/cvssscoremin-7/cvssscoremax-7.99/Linux-Linux-Kernel.html

[20] 2019. OpenVZ Containers. https://openvz.org/Main_Page

[21] 2019. Pods in Kubernetes. https://kubernetes.io/docs/concepts/workloads/pods/pod/

[22] 2019. Rumprun Unikernel. https://github.com/rumpkernel/rumprun

[23] 2019. runV: Hypervisor-based Runtime for OCI. https://github.com/hyperhq/runv

[24] 2019. User Mode Linux FAQ. http://uml.devloop.org.uk/faq.html

[25] 2019. vSphere Integrated Containers. https://www.vmware.com/products/vsphere/integrated-containers.html

[26] 2019. What is the difference between an "aggregate" and other kinds of "modified versions"? https://www.gnu.org/licenses/gpl-faq.en.html#MereAggregation

[27] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. 1986. Mach: A new kernel foundation for UNIX development. (1986).

[28] T. E. Anderson. 1992. The case for application-specific operating systems. In *[1992] Proceedings Third Workshop on Workstation Operating Systems*. 92–94. https://doi.org/10.1109/WWOS.1992.275682

[29] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. USENIX Association, GA, 689–703. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov

[30] A. Balalaie, A. Heydarnoori, and P. Jamshidi. 2016. Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. *IEEE Software* 33, 3 (May 2016), 42–52. https://doi.org/10.1109/MS.2016.64

[31] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. 2016. Migrating to Cloud-Native Architectures Using Microservices: An Experience Report. In *Advances in Service-Oriented and Cloud Computing*. Springer International Publishing, Cham, 201–215.

[32] 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 Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. ACM, New York, NY, USA, 164–177. https://doi.org/10.1145/945445.945462

[33] Len Bass, Ingo Weber, and Liming Zhu. 2015. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional.

[34] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 267–283. http://dl.acm.org/citation.cfm?id=2685048.2685070

[35] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 335–348. http://dl.acm.org/citation.cfm?id=2387880.2387913

[36] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 49–65. http://dl.acm.org/citation.cfm?id=2685048.2685053

[37] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. 1995. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM, New York, NY, USA, 267–283. https://doi.org/10.1145/224056.224077

[38] Eric A. Brewer. 2015. Kubernetes and the Path to Cloud Native. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15)*. ACM, New York, NY, USA, 167–167. https://doi.org/10.1145/2806777.2809955

[39] Brendan Burns and David Oppenheimer. 2016. Design Patterns for Container-based Distributed Systems. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. USENIX Association, Denver, CO. https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/burns

[40] David R. Cheriton and Kenneth J. Duda. 1994. A Caching Model of Operating System Kernel Functionality. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation (OSDI '94)*. USENIX Association, Berkeley, CA, USA, Article 14. http://dl.acm.org/citation.cfm?id=1267638.1267652

[41] DATADOG. 2018. 8 Surprising Facts About Real Docker Adoption. https://www.datadoghq.com/docker-adoption/

[42] Jeff Dike. 2000. A user-mode port of the Linux kernel. In *Annual Linux Showcase & Conference*.

[43] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. 1995. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM, New York, NY, USA, 251–266. https://doi.org/10.1145/224056.224076

[44] Nick Gauthier. 2017. Kernel Load-Balancing for Docker Containers Using IPVS. https://blog.codeship.com/kernel-load-balancing-for-docker-containers-using-ipvs/

[45] Sharath George. 2008. *Usermode Kernel: running the kernel in userspace in VM environments*. Master's thesis. University of British Columbia. https://doi.org/10.14288/1.0051274

[46] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. 2012. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. USENIX, Bellevue, WA, 475–490. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/giuffrida

[47] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Jean Wolter, and Sebastian Schönberg. 1997. The Performance of μ-kernel-based Systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*. ACM, New York, NY, USA, 66–77. https://doi.org/10.1145/268998.266660

[48] Jin Heo, X. Zhu, P. Padala, and Z. Wang. 2009. Memory overbooking and dynamic control of Xen virtual machines in consolidated environments. In *2009 IFIP/IEEE International Symposium on Integrated Network Management*. 630–637. https://doi.org/10.1109/INM.2009.5188871

[49] Galen Hunt, Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, James Larus, Steven Levi, Bjarne Steensgaard, David Tarditi, and Ted Wobber. 2007. Sealing OS Processes to Improve Dependability and Safety. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07)*. ACM, New York, NY, USA, 341–354. https://doi.org/10.1145/1272996.1273032

[50] Galen C. Hunt and James R. Larus. 2007. Singularity: Rethinking the Software Stack. *SIGOPS Oper. Syst. Rev.* 41, 2 (April 2007), 37–49. https://doi.org/10.1145/1243418.1243424

[51] Bilgin Ibryam. 2017. Principles of Container-Based Application Design. *Redhat Consulting Whitepaper* (2017). https://www.redhat.com/en/resources/cloud-native-container-design-whitepaper

[52] Sandra K Johnson, Gerrit Huizenga, and Badari Pulavarty. 2005. *Performance Tuning for Linux Servers*. IBM.

[53] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. 2014. OSv: Optimizing the Operating System for Virtual Machines. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 61–72. http://dl.acm.org/citation.cfm?id=2643634.2643642

[54] R. Krishnamurthy and G. N. Rouskas. 2015. On the impact of scheduler settings on the performance of multi-threaded SIP servers. In *2015 IEEE International Conference on Communications (ICC)*. 6175–6180. https://doi.org/10.1109/ICC.2015.7249307

[55] Nicolas Lacasse. 2018. Open-sourcing gVisor, a sandboxed container runtime. https://cloudplatform.googleblog.com/2018/05/Open-sourcing-gVisor-a-sandboxed-container-runtime.html

[56] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. 2006. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE J.Sel. A. Commun.* 14, 7 (Sept. 2006), 1280–1297. https://doi.org/10.1109/49.536480

[57] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01207

[58] 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 Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 461–472. https://doi.org/10.1145/2451116.2451167

[59] Toshiyuki Maeda. 2003. Kernel Mode Linux. https://www.linuxjournal.com/article/6516

[60] 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, USA, 218–233. https://doi.org/10.1145/3132747.3132763

[61] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the Art of Network Function Virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. USENIX Association, Berkeley, CA, USA, 459–473. http://dl.acm.org/citation.cfm?id=2616448.2616491

[62] Sam Newman. 2015. *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.".

[63] Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. 2009. Automated Control of Multiple Virtualized Resources. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys '09)*. ACM, New York, NY, USA, 13–26. https://doi.org/10.1145/1519065.1519068

[64] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. 2011. Rethinking the Library OS from the Top Down. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 291–304. https://doi.org/10.1145/1950365.1950399

[65] O. Purdila, L. A. Grijincu, and N. Tapus. 2010. LKL: The Linux kernel library. In *9th RoEduNet IEEE International Conference*. 328–333.

[66] Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, and Jonathan Appavoo. 2016. EbbRT: A Framework for Building Per-Application Library Operating Systems. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, GA, 671–688. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/schatzberg

[67] 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 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07)*. ACM, New York, NY, USA, 275–287. https://doi.org/10.1145/1272996.1273025

[68] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. 2003. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. ACM, New York, NY, USA, 207–222. https://doi.org/10.1145/945445.945466

[69] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. 2014. Cooperation and Security Isolation of Library OSes for Multi-process Applications. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York, NY, USA, Article 9, 14 pages. https://doi.org/10.1145/2592798.2592812

[70] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP '93)*. ACM, New York, NY, USA, 203–216. https://doi.org/10.1145/168619.168635

[71] Dan Williams, Hani Jamjoom, and Hakim Weatherspoon. 2012. The Xen-Blanket: Virtualize Once, Run Everywhere. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. ACM, New York, NY, USA, 113–126. https://doi.org/10.1145/2168836.2168849