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

Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan
Christina Delimitrou, Robbert Van Renesse, 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. Due to the resulting change in the threat model, process isolation within the container becomes redundant in most single-concerned containers, and inter-container isolation becomes increasingly important. In this paper, we propose a new exokernel-inspired architecture called X-Containers that improves both the security and the performance of cloud-native containers. We show that, through relatively minor modifications, the Xen hypervisor can serve as an exokernel, and Linux can be turned into a LibOS. Doing so results in a highly secure and efficient LibOS platform that, unlike other available LibOSes, supports binary compatibility and multicore processing. X-Containers have up to $27\times$ higher raw system call throughput compared to Docker containers, while also significantly outperforming recent container platforms such as Google's gVisor, Intel's Clear Containers, as well as Library OSes like Unikernel and Graphene on web benchmarks.

1. Introduction

An important recent trend in cloud computing is the rise of "cloud-native" container platforms, such as Kubernetes [30], which have become an integral part of production environments. Such platforms support applications designed specifically for cloud infrastructures that consist of loosely-coupled microservices [51] running in containers, with distributed management and orchestration. Cloud-native platforms offer support for automated elastic scaling and agile *DevOps* practices [26], which allow companies to bring new ideas to market faster, respond quickly to customer demands, and handle failures more transparently.

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 [31]. 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" [39], and is recommended by Docker [3].

From a security perspective, single-concerned containers change the threat model of the container architecture, presenting both a challenge and an opportunity. Process isolation, which separates different applications in traditional operating systems, becomes redundant in most single-concerned containers. On the other hand, inter-container isolation becomes increasingly important, and raises many concerns because containers share a monolithic OS kernel with a large Trusted Computing Base (TCB), as well as a large attack surface in terms of the number of kernel interfaces.

There have been several proposals to address the issue of container isolation. Virtualization-based solutions, such as Clear Containers [8], Kata Containers [9], and Hyper Containers [6], wrap containers with a dedicated OS kernel running in a virtual machine (VM). These platforms require native hardware virtualization support to reduce the overhead of adding another layer of indirection. However, most public and private clouds, including Amazon EC2, do not support nested hardware virtualization. Even in clouds such as Google Compute Engine where nested hardware virtualization is enabled, its performance overhead is high (see Section 5 and [15]). LightVM [49] wraps a container in a paravirtualized Xen instance without hardware virtualization support. Unfortunately, it introduces a significant performance penalty in x86-64 platforms (see Sections 4.1 and 5). Finally, Google gVisor [5] is a user-space kernel written in Go that supports container runtime sandboxing, but it only offers limited system call compatibility [13] and incurs significant performance overheads (see Section 5).

The trend of running a single application in its own VM for enhanced security has led to a renewed interest in Library Operating Systems (LibOSes), as suggested by the Unikernel [46] 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 [34] model can improve both container isolation and performance. However, existing LibOSes, such as MirageOS [46], Graphene [58], and OS^v [41], lack features, such as full binary compatibility or multicore support. This makes porting containerized applications very challenging.

In this paper, we propose a new LibOS platform called *X-Containers* that improves both the security and performance of containers without requiring hardware virtualization support. We demonstrate that Xen's paravirtualization architecture [25] can be modified to serve as a highly secure and efficient LibOS platform that supports both binary compatibility and multicore processing. 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 model processes are used for concurrency, while X-Containers provide isolation between containers.

Without hardware virtualization support, system calls are expensive, as they are first handled by 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. As a result, X-Containers have up to 27× higher raw system call throughput compared to native Docker containers running in the cloud, and are competitive to or even outperform native containers for other benchmarks.

The X-Container platform also outperforms other LibOS architectures for specialized services, such as serverless compute, which are gaining traction for short-running, user-driven online services with intermittent behavior. We compare X-Containers, Unikernel, and Graphene, using NGINX [12], a stateless front-end webserver driven by the `wrk` workload generator. We show that X-Containers have comparable performance to Unikernel, and twice the throughput compared to Graphene. Moreover, when running PHP and MySQL, X-Container achieves approximately 3× the performance of Unikernel.

This paper includes the following contributions:

- 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 multicore processing.
- We present X-Containers, a new exokernel-based container architecture that is designed specifically for cloud-native applications. X-Containers are compatible with Linux containers, and to the best of our knowledge, they are the first architecture to support secure isolation of containers in the cloud, without sacrificing compatibility or performance.
- 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 and isolation.

2. Background and Motivation

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 cloud-native platforms [24, 23]. 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 [16], and a "task" in Amazon ECS [2]. It is important to note that single-concerned containers are not necessarily single-process. Some applications might spawn multiple worker processes for concurrency [3], such as NGINX, or Apache webserver. The key is that all processes within a single-concerned container belong to the same service, thus they are tightly coupled and mutually trusting.

2.2. Application Isolation

In traditional operating systems, processes are used for both multiprocessing and security isolation between applications. 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. In single-concerned cloud-native containers, process isolation within the container is unnecessary.

Indeed, an informal survey of popular containers shows that multi-client applications rarely rely on processes for isolating mutually-untrusting clients by dedicating a process to each client—many do not even use multiple processes at all. All the top 10 most popular containerized applications [1] (NGINX, Redis, ElasticSearch, Registry, Postgres, MySQL, etcd, Fluentd, MongoDB, and RabbitMQ) use either a single-threaded event-driven model or multi-threading instead of multiple processes. NGINX and Fluentd can be configured to use a process pool to improving concurrency, similar to Apache webserver—each process has multiple threads that can be used to serve 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.

On the other hand, inter-container isolation remains important, and depends on enforcement by the OS kernel. Unfortunately, modern monolithic OS kernels, such as Linux, have become a large code base with complicated services, device drivers, and extensive system call interfaces. New vulnerabilities are continually being discovered. For example, the recently-disclosed Meltdown attack [45] breaks isolation between containers sharing the same kernel. In addition, there are still more than 500 security vulnerabilities in the Linux kernel [11]. Due to such security concerns, containers are typically run in separate VMs, which sacrifices scalability, performance, and resource efficiency in favor of improved security isolation between containers.

2.3. Challenges of Running Containers with LibOSes

The trend of running a single application in its own VM for enhanced security has led to a renewed interest in the LibOS paradigm, as exemplified by the Unikernel model. 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

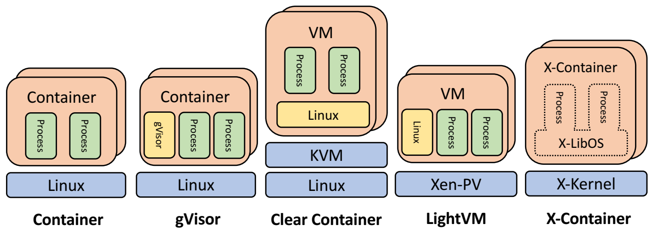


Figure 1: Comparison of different container architectures.

compatibility can make the porting of many containers infeasible. Even for containers that have the source code of all dependencies, any code change or re-compilation can potentially introduce security or compatibility issues that are not acceptable in production environments.

- **Multicore Processing**: While binary compatibility ensures support for spawning multiple processes, multicore processing refers to the capability of running multiple processes concurrently. As an example of the distinction, user-space kernels, such as Google gVisor [5] and User Mode Linux (UML) [33], support spawning multiple processes, but they can only run a single process at a time even when multiple CPU cores are available. Without multicore processing, the performance of many applications would be dramatically impacted due to the reduced concurrency.

To the best of our knowledge, no existing LibOS, except X-Containers, provides both these features. Unikernel [46] and related projects, such as Dune [27, 28], EbbRT [55], OS^v [41], and ClickOS [50], only support single-process applications, and involve substantial source code and compilation changes. Graphene [58] supports concurrent multiprocessing, but provides only one third of the Linux system calls.

3. X-Container Design

The X-Containers architecture is different from existing container architectures, as shown in Figure 1. gVisor [5] has a user-space kernel isolated in its own address space. Clear Container [8] and LightVM [49] run each container in its own virtual machine (using KVM and Xen-PV resp.). In the X-Container architecture, each (single-concerned) container runs with its own LibOS called X-LibOS. Inter-container isolation is guarded by by the *X-Kernel*, a virtual machine monitor acting as an exokernel. The X-Kernel ensures both a small kernel attack surface (*i.e.*, a small number of well-documented system calls) and a small TCB.

3.1. Compatibility vs. Efficiency

In the design of X-Containers, we make a specific trade-off between *compatibility* and *efficiency*. Here compatibility includes not only the Application Binary Interface (ABI) support for running existing applications without change, but also the ability to work with existing software development, profiling, debugging, and deploying tools. Most LibOSes require substantial changes to the application's source code, or even require re-implementing the application from scratch. Their development and debugging support is also far behind the corresponding toolsets available for ordinary systems. As a result, although these LibOSes can be highly optimized for efficiency by design, they sacrifice compatibility, and hence the opportunity to leverage existing, mature infrastructure that has been optimized and tested for years.

3.2. Why Use Linux as the LibOS?

We believe that the best way to develop a 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 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 [40]. 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 of the popular event-driven applications, disabling multi-core and Symmetric Multi-Processing (SMP) support can eliminate unnecessary locking and TLB shoot-downs, which greatly improves performance. Depending on the workload, applications can set different policies in the Linux scheduler. 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 Exokernel?

There have been previous attempts to turn an existing feature-rich monolithic OS kernel into a LibOS [53, 54]. However, these projects also use a monolithic OS kernel to serve as the host kernel. For example, the Linux Kernel Library (LKL) project [54] 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 mapping and scheduling. The same design choice was made by Drawbridge [53], which turns the Windows 7 OS kernel into a LibOS running on Windows and only supports single-process applications.

Graphene [58] 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 [25] (see Section 4.1). There are four reasons that make Xen ideal for implementing a LibOS with binary compatibility and multicore processing.

- *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 multicore processing when turning the guest kernel into a LibOS.
- *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. Threat Model

We focus on single-concerned containers, which correspond to either single-process applications, or applications that 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 hypervisor calls that lead to a smaller number of vulnerabilities in practice. This allows X-Containers to provide better protection to external threats than standard containers isolated by a Linux kernel.

Running an application over a LibOS removes security isolation between a process and the 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 [38], Software-based Fault Isolation [59], Nooks [57], and SPIN [29]. Programming bugs, inadvertent design flaws, denial of service attacks, and side-channel attacks are outside our threat model since their mitigation involves solutions orthogonal to our design, which can additionally be integrated in X-Containers.

4. Implementation

We have implemented a prototype of the X-Containers platform based on Xen and Linux. We leveraged Xen-Blanket [60] drivers to run the platform efficiently in public clouds. We focused on applications running in x86-64 long mode. The modifications to the kernel are in the architecture-dependent layer and transparent to other layers in the kernel. 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 or Domain-Us) 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 guest kernels and user processes 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 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, an `iret` instruction is typically used to reset code and stack segments, flags, and the stack and instruction pointers. Interrupts can also be enabled atomically. To guarantee atomicity and security when switching privilege levels, Xen provides a hypercall for implementing `iret`. In the X-Container architecture, this hypercall is not necessary, and we implement `iret` completely in user mode by pushing registers temporally into the kernel stack and resuming the context with the ordinary `ret` instruction. Similar to `iret`, the `sysret` instruction, which is used for returning from a system call handler, is optimized without trapping in the kernel.

4.3. Multicore Processing Support

X-Containers inherit support for multicore processing 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 does not require a full TLB flush, which greatly improves the performance of address translation. Context switches between different X-Containers do trigger a full TLB flush.

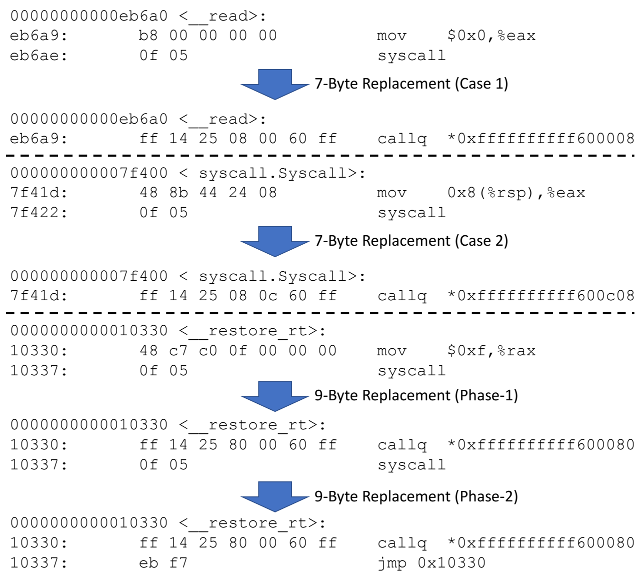
Because the kernel code is no longer protected, kernel routines would not need a dedicated stack if the X-LibOS only supported a single process. However, since the X-LibOS supports multiple processes, we still need dedicated kernel stacks in the kernel context, and when performing a system call, a switch 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 the write-protection bit in the `CR-0` register, so that code running in kernel mode can change any memory page even if it is mapped read-only in the page table. ABOM then performs the binary patch with atomic `cmpxchg` instructions. 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


7-Byte Replacement (Case 1)
7-Byte Replacement (Case 2)
9-Byte Replacement (Phase-1)
9-Byte Replacement (Phase-2)

```
00000000000eb6a0 <__read>:
eb6a9:       b8 00 00 00 00          mov    $0x0,%eax
eb6ae:       0f 05                   syscall

00000000000eb6a0 <__read>:
eb6a9:       ff 14 25 08 00 60 ff    callq  *0xffffffffff600008
------------------------------------------------------------------
000000000007f400 < syscall.Syscall>:
7f41d:       48 8b 44 24 08          mov    0x8(%rsp),%eax
7f422:       0f 05                   syscall

000000000007f400 < syscall.Syscall>:
7f41d:       ff 14 25 08 0c 60 ff    callq  *0xffffffffff600c08
------------------------------------------------------------------
0000000000010330 <__restore_rt>:
10330:       48 c7 c0 0f 00 00 00    mov    $0xf,%rax
10337:       0f 05                   syscall

0000000000010330 <__restore_rt>:
10330:       ff 14 25 80 00 60 ff    callq  *0xffffffffff600080
10337:       0f 05                   syscall

0000000000010330 <__restore_rt>:
10330:       ff 14 25 80 00 60 ff    callq  *0xffffffffff600080
10337:       eb f7                   jmp 0x10330
```

Figure 2: Examples of binary replacement.

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. 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. 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 fix the trap, by moving the instruction pointer backward to the beginning of the call instruction. We have only seen this triggered during the boot time of some operating systems.

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. Current Limitations

In this paper, we address the key challenges of designing a lightweight and performant container technology, by turning the Xen PV architecture into a LibOS platform for supporting unmodified application containers. There are some remaining challenges that are not the focus of this paper, and for which previously-proposed solutions can be applied. We discuss these limitations below.

Memory management: In our prototype, each X-Container is configured with a static memory size. Linux containers are able to adjust memory allocation dynamically. Dynamic memory allocation and over-subscription of Xen VMs have been studied in literature [37, 52], leveraging mechanisms such as ballooning. In addition, Xen provides native Transcendent Memory (tmem) support [47], which can be leveraged by Linux kernels in different VMs for efficiently sharing the page cache and RAM-based swap space [48].

Spawning speed of new instances: An important benefit of containers is that they can be spawned much faster than an ordinary VM. The current implementation of X-Containers places a penalty on the startup cost of a container, but we are taking measures to reduce this startup time. For example, to support Docker containers, we implemented a *Docker Wrapper*. To bootstrap an X-Container, the Docker Wrapper loads an X-LibOS with a Docker image and a special bootloader. The bootloader spawns the processes of the container directly without running any unnecessary services, significantly reducing the start-up cost compared to starting an ordinary VM. In spite of this, still some instantiation overhead remains. For example, we can boot an X-LibOS with a single `bash` process in 180ms, but the overhead of Xen's "xl" toolstack brings the total instantiation time up to 3 seconds. LightVM has proposed a solution to reduce the overhead of the toolstack to 4ms [49], which can be also applied to X-Containers. The delay in bootstrapping a Linux kernel can be reduced by leveraging VM cloning [42, 43].

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?

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)

Table 1: Evaluation of the Automatic Binary Optimization Module (ABOM)

- How does the scalability of X-Containers compare to Docker Containers and VMs?
- How can kernel customization benefit performance?

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 and gVisor as baselines for our evaluation. In Google GCE, we enabled nested hardware virtualization and installed Clear Containers in Ubuntu-16 with KVM. We also implemented `Xen-Containers`, a platform similar to LightVM [49] 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.

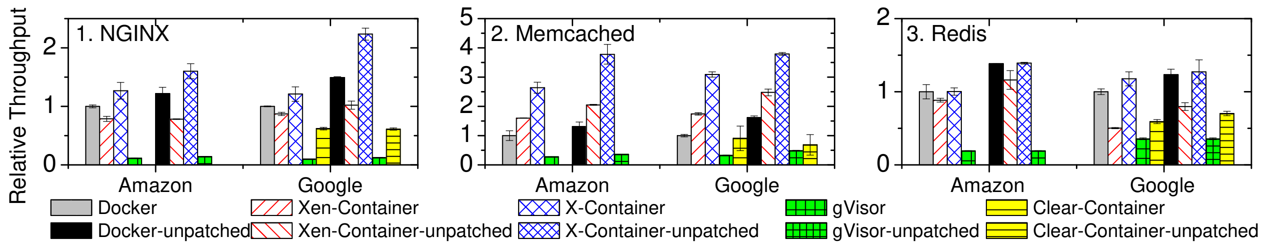
The VMs running native Docker, gVisor, and Clear Containers had Ubuntu 16.04-LTS installed with Docker engine 17.03.0-ce and Linux kernel 4.4. We used Linux kernel 4.14 as the guest kernel for Clear Containers since its current tool stack is no longer compatible with Linux 4.4. The VMs running Xen-Containers 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 use the same Docker image for all configurations. When running network benchmarks, we use separate VMs for the client and server. Unless otherwise noted we report the average and standard deviation of five runs for each experiment.

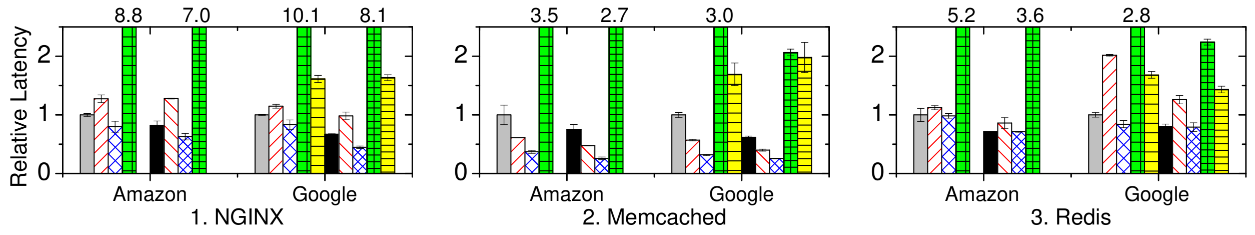
5.2. Automatic Binary Optimization

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 [1], and are written in a variety of programming languages. For each application, we used open-source workload generators as the clients.

Table 1 shows the applications we tested and the reduction in system call invocations that ABOM achieved. For all but one application we tested, ABOM turned 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%.



(a) Relative Throughput (higher is better)



(b) Relative Latency (lower is better)

Figure 3: Relative performance of macrobenchmarks.

5.3. Macrobenchmarks

We evaluated the performance of X-Containers with three macrobenchmarks: NGINX, Memcached, and Redis. The corresponding Docker images we used were `nginx:1.13`, `memcached:1.5.7`, and `redis:3.2.11`, 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 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 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 suffers a significant performance penalty for using nested hardware virtualization (also measured by Google [15]). X-Containers outperformed Docker, gVisor, Xen-Containers, and Clear Containers. Notably, X-Containers improved throughput of Memcached from 134% to 208% compared to native Docker. For NGINX, X-Containers achieved 21% to 50% throughput improvement over Docker. For Redis, the performance of X-Containers was comparable to Docker, but note that this was achieved with stronger inter-container isolation. 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 effect of changing system calls into function calls, we also evaluated performance with a set of microbenchmarks. We started with an Ubuntu-16

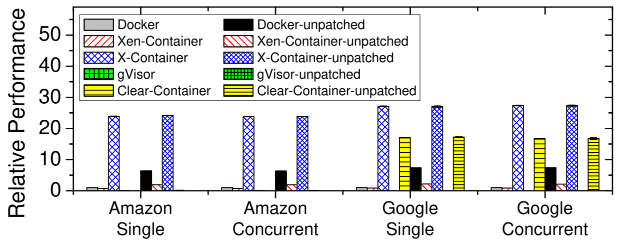


Figure 4: Relative system call throughput (higher is better).

Docker image, and ran `UnixBench` and `iperf` on it. 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 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 see 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`), because system calls are converted to function calls. The throughput of gVisor is only 7 to 9% of Docker due to the high overhead of `ptrace`, so can be barely seen in the figure. Clear Containers achieved much better system call throughput than Docker because the guest kernel is highly optimized by disabling most security features within a Clear container. Also, note that the Meltdown patch does not affect performance of X-Containers

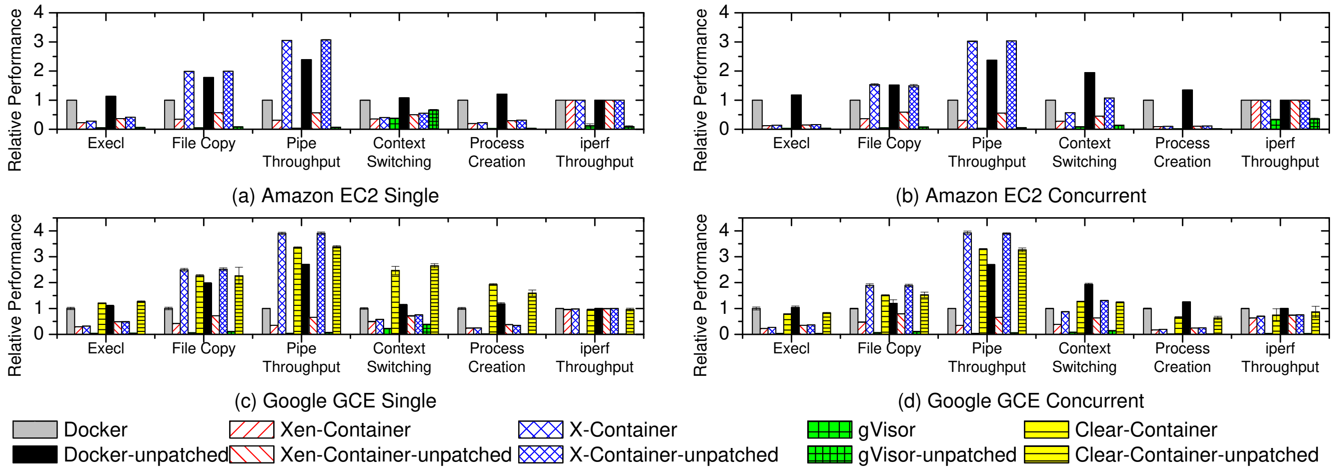


Figure 5: Relative performance of microbenchmarks (higher is better).

and Clear Containers because for X-Containers the system calls did not trap into kernel mode, and for Clear Containers the guest kernel was always unpatched.

Figure 5 shows the relative performance for other microbenchmarks, also normalized to native patched `Docker`. Similar to the system call throughput benchmark, the Meltdown patch did not affect X-Containers and Clear Containers. In contrast, patched Docker containers and Xen-Containers suffer significant performance penalties. X-Containers has noticeable overheads compared to Docker in process creation and context switching. This is because process creation and context switches involves page table operations, which 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* [17] because it can run the benchmarks with minor patches (running with MirageOS [46] requires rewriting the application in OCaml).

Figure 6a compares throughput of the NGINX webserver serving static webpages with a single worker process. As there is only one NGINX server process running, we dedicated a single CPU core for X-Containers and Unikernel. X-Containers achieved 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 outperformed 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

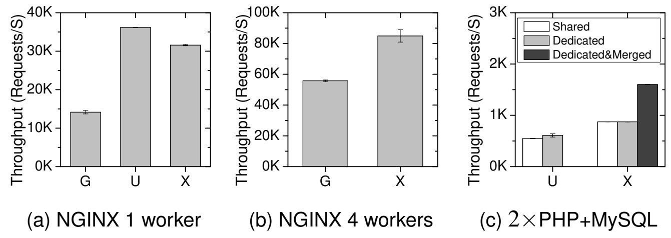


Figure 6: Throughput comparison for Unikernel (U), Graphene (G), and X-Container (X).

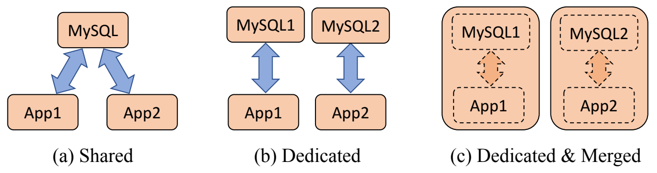


Figure 7: Alternate configurations of two applications that use MySQL.

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 either share the database or have dedicated databases, so there are three possible configurations for this setup depending on the threat model and security requirements. Figure 6c shows the total throughput of two PHP servers with different configurations. All VMs are running a single process with one CPU core. With `Shared` and `Dedicated` configurations, X-Containers outperformed Unikernel by over 40%. We believe that this is because the Linux kernel outperforms the Rumprun kernel for this benchmark. Furthermore, X-Container supports running PHP and MySQL in a single container (the `Dedicated&Merged` configuration), which is not possible for Unikernel (which only supports a single process). Using this setup, X-Container throughput was about three times that of the Unikernel `Dedicated` configuration.

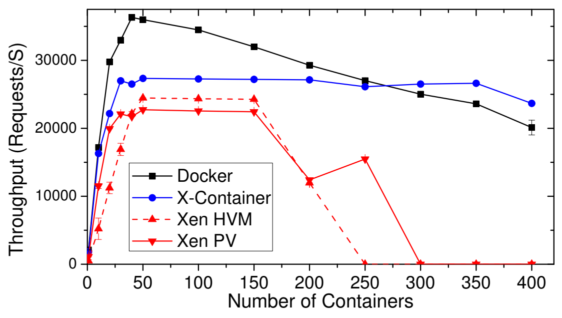


Figure 8: Throughput scalability as the number of containers increases.

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.[1] We also evaluated `Xen HVM` and `Xen PV` configurations that ran Docker containers in regular Xen HV and PV instances respectively. Each Xen VM was assigned 1 vCPU and 512MB memory (512MB is the recommended minimum size for Ubuntu-16). However, because the physical machine only had 96GB memory, when starting more than 200 VMs, we changed the per-VM memory size to 256MB. We found that the VMs could still boot but the network started dropping packets. We were not able to boot more than 250 PV instances, or more than 200 HV instances on Xen.

Figure 8 shows the aggregated throughput of all bare-metal configurations. We can see that `Docker` containers achieved higher throughput for small numbers of containers. This is because context switching between Docker containers is cheaper than between X-Containers and between Xen VMs. However, as the number of containers increased, the performance of Docker containers dropped faster. This is because each NGINX+PHP container ran 4 processes: with N containers, the Linux kernel running Docker containers was scheduling $4N$ processes, while X-Kernel was scheduling N virtual CPUs, each running 4 processes. This hierarchical scheduling turned out to be a more scalable way of co-scheduling many containers, and, with $N = 400$, `X-Containers` outperformed `Docker` by 18%.

5.7. Benefits of Kernel Customization

The X-Containers platform enables applications that require customized kernel modules to run in containers. For example, X-Containers can run software RDMA (both Soft-iwarp and

[1]X-Containers also work with 64MB memory, but for this experiment 128MB is sufficiently small to boot 400 X-Containers.

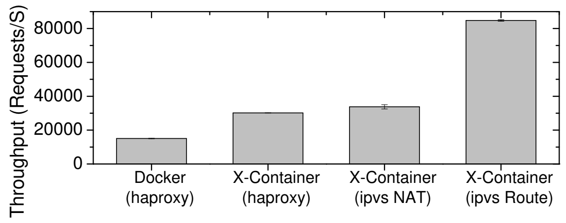


Figure 9: Kernel-level load balancing.

Soft-ROCE) applications. In Docker environments, such modules require root privilege and expose the host network to the container directly, raising security concerns.

In this section, we present a case study of kernel customization in X-Containers that illustrates a performance benefit not easily achievable on Docker platforms. We tested a scenario with three NGINX webservers and a load balancer. The NGINX webservers are each configured to use one worker process. Docker platforms typically use a user-level load balancer, such as *HAProxy*. HAProxy is a single-threaded, event-driven proxy server widely deployed in production systems. X-Containers supports HAProxy, but can also use kernel-level load balancing solutions, such as IPVS (IP Virtual Server). IPVS requires inserting new kernel modules and changing iptable and ARP table rules, which is not possible in Docker without root privilege and access to the host network.

In this experiment, we used the `HAProxy:1.7.5` Docker image. The load balancer and NGINX servers were running on the same physical machine. We configured each X-Container with a single vCPU. We used the `wrk` workload generator and measured total throughput.

Figure 9 compares various configurations. X-Containers with HAProxy achieved twice the throughput of Docker containers. With IPVS kernel level load balancing using NAT mode, X-Containers further improve throughput by 12%. In this case the load balancer was the bottleneck because it served as both the web front-end and NAT server. IPVS supports another load balancing mode called "direct routing." With direct routing, the load balancer only needs to forward requests to backend servers while responses from backend servers are routed directly to clients. This requires changing iptable rules and inserting kernel modules both in the load balancer and NGINX servers. With direct routing mode, the bottleneck shifted to the NGINX servers, and total throughput improved by another factor of 2.5.

6. Related Work

The X-Container architecture is a LibOS platform designed specifically for cloud-native systems. Below we discuss work on containers and LibOSes related to X-Containers.

6.1. Application Containers

OS-level virtualization [56] provides a lightweight mechanism of running multiple OS instances. Docker [4], LXC [10], OpenVZ [14], and Solaris Zones [18] 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 extremely difficult to define a policy for arbitrary, previously unknown applications [5].

Various runtimes have been proposed to address the problem of security isolation in containers. Clear Containers [8], Kata Containers [9], Hyper Containers [6], VMWare vSphere Integrated Containers [19], and Hyper-V containers [7] 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. Google gVisor [5] is a user-space kernel written in Go that supports container runtime sandboxing, but it provides limited compatibility [13] and incurs significant performance overhead.

LightVM with TinyX [49] 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 [60] 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 [22] 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 provide full binary compatibility to existing containers, and cannot run multiple processes within a container.

6.2. Library OS

The insight of a Library OS [34, 53, 21, 32, 44] is to keep the kernel small and link applications to a LibOS containing functions that are traditionally performed in the kernel. Most Library OSes [34, 20, 36, 53, 54] 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 [58] 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 [46] and related projects, such as EbbRT [55], OS^v [41], ClickOS [50], and Dune [27, 28], 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 [35] is an idea similar to X-Containers that runs parts of the Linux kernel in userspace in VM environments. 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.

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 security and isolation. 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

Exokernels are an ideal abstraction for single-concerned cloud-native containers: minimal kernels can securely isolate mutually untrusting containers, and Library OSes allow for customization and can run containers efficiently. However, current Exokernels and Library OSes do not support both binary-level compatibility and multicore processing. The X-Containers platform uses the Xen hypervisor as an exokernel and a modified Linux as an excellent LibOS that can be deployed in clouds without specific support from underlying cloud providers. The paper shows that X-Containers can significantly outperform other container and Library OS platforms.

8. Acknowledgements

This work was partially 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] 8 surprising facts about real docker adoption. https://www.datadoghq.com/docker-adoption/.
[2] Amazon ecs task definitions. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task_definitions.html.
[3] Best practices for writing dockerfiles. https://docs.docker.com/develop/develop-images/dockerfile_best-practices/.
[4] Docker. https://www.docker.com/.
[5] gVisor: Container Runtime Snadbox. https://github.com/google/gvisor.
[6] Hyper containers. https://hypercontainer.io.
[7] Hyper-V Containers. https://docs.microsoft.com/en-us/virtualization/windowscontainers/manage-containers/hyperv-container.
[8] Intel clear containers. https://clearlinux.org/containers.
[9] Kata containers. https://katacontainers.io.
[10] Linux LXC. https://linuxcontainers.org/.
[11] 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.
[12] NGINX official website. http://nginx.org.
[13] Open-sourcing gVisor, a sandboxed container runtime. https://cloudplatform.googleblog.com/2018/05/Open-sourcing-gVisor-a-sandboxed-container-runtime.html.

[14] OpenVZ Containers. https://openvz.org/Main_Page.

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

[16] Pods in kubernetes. https://kubernetes.io/docs/concepts/workloads/pods/pod/.

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

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

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

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

[21] T. E. Anderson. The case for application-specific operating systems. In *[1992] Proceedings Third Workshop on Workstation Operating Systems*, pages 92–94, Apr 1992.

[22] 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. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pages 689–703, GA, November 2016. USENIX Association.

[23] A. Balalaie, A. Heydarnoori, and P. Jamshidi. Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. *IEEE Software*, 33(3):42–52, May 2016.

[24] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Migrating to cloud-native architectures using microservices: An experience report. In *Advances in Service-Oriented and Cloud Computing*, pages 201–215, Cham, 2016. Springer International Publishing.

[25] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

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

[27] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 335–348, Berkeley, CA, USA, 2012. USENIX Association.

[28] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 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, pages 49–65, Berkeley, CA, USA, 2014. USENIX Association.

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

[30] Eric A. Brewer. Kubernetes and the Path to Cloud Native. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 167–167, New York, NY, USA, 2015. ACM.

[31] Brendan Burns and David Oppenheimer. Design Patterns for Container-based Distributed Systems. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, 2016. USENIX Association.

[32] David R. Cheriton and Kenneth J. Duda. A Caching Model of Operating System Kernel Functionality. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association.

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

[34] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[35] Sharath George. Usermode kernel: running the kernel in userspace in VM environments. Master's thesis, University of British Columbia, Dec 2008.

[36] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Jean Wolter, and Sebastian Schönberg. The performance of μ-kernel-based systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 66–77, New York, NY, USA, 1997. ACM.

[37] Jin Heo, X. Zhu, P. Padala, and Z. Wang. Memory overbooking and dynamic control of Xen virtual machines in consolidated environments. In *2009 IFIP/IEEE International Symposium on Integrated Network Management*, pages 630–637, June 2009.

[38] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, April 2007.

[39] Bilgin Ibryam. Principles of Container-Based Application Design. *Redhat Consulting Whitepaper*, 2017.

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

[41] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. OSv: Optimizing the Operating System for Virtual Machines. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 61–72, Berkeley, CA, USA, 2014. USENIX Association.

[42] Ivan Krsul, Arijit Ganguly, Jian Zhang, Jose A. B. Fortes, and Renato J. Figueiredo. VMPlants: Providing and Managing Virtual Machine Execution Environments for Grid Computing. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC '04, pages 7–, Washington, DC, USA, 2004. IEEE Computer Society.

[43] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and Mahadev Satyanarayanan. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 1–12, New York, NY, USA, 2009. ACM.

[44] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE J.Sel. A. Commun.*, 14(7):1280–1297, September 2006.

[45] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *ArXiv e-prints*, January 2018.

[46] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 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, pages 461–472, New York, NY, USA, 2013. ACM.

[47] Dan Magenheimer et al. Transcendent memory on xen. *Xen Summit*, pages 1–3, 2009.

[48] Dan Magenheimer, Chris Mason, Dave McCracken, and Kurt Hackel. Transcendent memory and linux. In *Proceedings of the Linux Symposium*, pages 191–200. Citeseer, 2009.

[49] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) Than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 218–233, New York, NY, USA, 2017. ACM.

[50] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. ClickOS and the Art of Network Function Virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 459–473, Berkeley, CA, USA, 2014. USENIX Association.

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

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

[53] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. 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, pages 291–304, New York, NY, USA, 2011. ACM.

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

[55] Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, and Jonathan Appavoo. EbbRT: A framework for building per-application library operating systems. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 671–688, GA, 2016. USENIX Association.

[56] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. 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, pages 275–287, New York, NY, USA, 2007. ACM.

[57] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 207–222, New York, NY, USA, 2003. ACM.

[58] 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. Cooperation and security isolation of Library OSes for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 9:1–9:14, New York, NY, USA, 2014. ACM.

[59] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 203–216, New York, NY, USA, 1993. ACM.

[60] Dan Williams, Hani Jamjoom, and Hakim Weatherspoon. The Xen-Blanket: Virtualize once, run everywhere. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 113–126, New York, NY, USA, 2012. ACM.