How to Design a Library OS for Practical Containers?

Hajime Tazaki
IIJ Research Laboratory
Japan

Akira Moroo
Ricerca Security, Inc.
Japan

Yohei Kuga
The University of Tokyo
Japan

Ryo Nakamura
The University of Tokyo
Japan

Abstract

Container engines with operating-system virtualization have been widely used and now offer extensions to replace core functionalities that are derived from the host kernel. Because such extensions with an alternate kernel, which is often implemented in a library operating system (libOS), can be designed to have free choice, developers are tempted to take a clean-slate approach, i.e., implement the kernels from scratch. However, this design decision makes it difficult to cover broad features of the original Linux kernel, and some application programs may not work on such kernels. Precise emulation of the huge codebase and rich feature set of the Linux kernel is not easily possible. In this paper, we have tried to improve the level of compatibility in a libOS by using the source code of the Linux kernel as the container kernel. We present μKontainer, an alternate container kernel based on a libOS by extending the existing open-source software, Linux Kernel Library, while preserving the lightweight property of conventional containers. We have studied the level of compatibility with the conformance tests of network protocol implementation of nine different libOSs, and μKontainer performs identically like the Linux kernel. The network-related benchmark shows mostly comparable results with a conventional container and a native Linux host; in the best case, the goodput of the short-sized packet is up to 84% faster than that of a native Linux host. This paper sheds light on the design space of the libOS when we introduced the extended container kernel.

CCS Concepts: • **Software and its engineering → Operating systems**; *Virtual machines.*

Keywords: library OS, unikernels, anykernel

ACM Reference Format:
Hajime Tazaki, Akira Moroo, Yohei Kuga, and Ryo Nakamura. 2021. How to Design a Library OS for Practical Containers?. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '21), April 16, 2021, Virtual, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3453933.3454011

VEE '21, April 16, 2021, Virtual, USA

ACM ISBN 978-1-4503-8394-3/21/04.
https://doi.org/10.1145/3453933.3454011

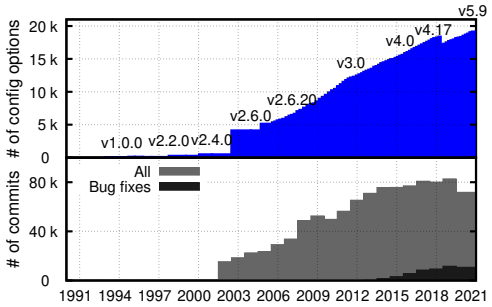


Figure 1. Evolution of the Linux kernel: with the number of configuration options, and the number of commits with the number of bug fixes. *Linux was not built in a day.*

1 Introduction

> *"Rome was not buylt in one day"*
>
> — (Erasmus's Proverbs)

Container engines and their ecosystems play important roles in the various fields where computation is required. With the advantage of lightweight execution compared with typical virtualization based on hardware-assisted partitioning, containers have been applied in numerous use cases: quick instantiation in a serverless computing context [32, 39, 58], runtime environment for foreign [13, 21, 35, 52] or new platforms (e.g., trusted execution environment [2, 57]), sandboxing of container instances [16, 52], experimental testbeds where a large number of OS instances are executed in parallel [19], and desktop application packaging [26]. To cover broader use cases, various studies have attempted to enhance the capability of the container execution and its runtime environments.

When the aforementioned use cases require extensions to involve different kernels (e.g., a container kernel) than a host kernel, where the conventional containers share the same host kernel among instances, one important property to verify such an extension is how the behavior of the new kernel differs from the original kernel, which is a Linux kernel in most cases. Because programs running on extended containers are often identical to the conventional containers, the functional degradation caused by incompatibility is not acceptable if we consider a large number of use cases. The lack of this level of compatibility (we call this **the kernel-level**

compatibility) might be a reason why software in production must reconsider its core design after deployment (e.g., in the case of Windows Subsystem for Linux (WSL1) [35]).

However, prior work on alternate container kernels has struggled to achieve the required compatibility. Early implementations of unikernels equipped with a library operating system (libOS) [6, 29, 60] require application programs to be rebuilt to generate a statically linked binary file, resulting in source code-level compatibility with the programs. Thus, proprietary programs where the source code is unavailable do not benefit from the unikernels. Binary compatible libOSs [31, 49, 55] allow us to use Linux application binaries without any modifications; however, *the kernel-level compatibility* is not always complete because the destination of the translated system calls is not always Linux, for example, in the case of WSL1 [21] running on Windows kernel. Lightweight virtual machine monitors (VMMs) [1, 52, 59] reduce the severity of the presented issues by running a complete Linux guest OS on top of slim VMMs; however, this might be controversial because the conventional container was invented because of the heavy execution of the hardware virtualization, but it now uses hardware virtualization to fix problems that the conventional container cannot solve.

The difficulties to preserve the kernel-level compatibility are attributed to the continuous growth of the Linux kernel, as illustrated in Figure 1. The number of configuration options of the kernel, which is related to the number of implemented features, is continuously increasing except around version 4.17 when several old CPU architectures were removed from the tree [3]. Such enormous options, which have exceeded 10,000 since being established 10 years ago, are maintained by continuous and significant efforts from the community, which are represented by several tens of thousands of commits per year. Furthermore, the number of fixed bugs (the bottom half of Figure 1) is also increasing to make the software mature. Such long-term evolution by both enriching features and fixing issues is not easily reproducible while considering the precise compatibility of the behavior.

This paper presents μKontainer, a container runtime extension that implements unikernels over userspace processes, and pursues a further alternative approach to offer a Linux-compatible libOS for the container kernel. We aim at preserving the bug-for-bug compatibility; instead of reimplementing a container kernel from scratch, or translating system calls for the compatibility, μKontainer utilizes the source code of the Linux kernel as a libOS within a container kernel. We have extended Linux Kernel Library (LKL) [44] to be transparently executable via container runtime to retain mature, decades-long source code of the Linux kernel. Our goals are to offer **Linux kernel-level compatibility** (§ 5.1) and **portability** (§ 5.2), which are identical to the goals of lightweight VMMs to provide a container runtime for running practical applications, while minimizing the overhead introduced by the libOS (§ 5.3).

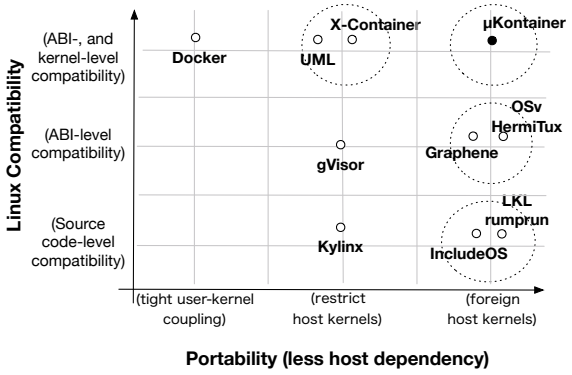


Figure 2. Design space of a libOS (container kernel) for a Linux container.

The contributions of this paper are summarized as follows.

- Review the design space of a library OS for the container kernel and identify the fundamental issues in terms of compatibility and platform portability.
- Implement an extended LKL to enable using the library as an alternate container kernel while offering Linux kernel-level compatibility.
- Evaluate concerns about performance under userspace execution and perform an extensive conformance study on the network stack implementations under network-related scenarios, which are primary use cases of containers.

2 Design Patterns of Library OSs

There are multiple ways to extend conventional containers (e.g., Docker) with an alternate container kernel (and libOS), but the achieved level of compatibility with the original Linux systems varies. In this section, we review these different levels of Linux compatibility from previous studies and classify them into the following three levels.

- **Source code-level compatibility**: application programs are runnable if source code is available and rebuilt for libOSs, but the binary is different from the original form, and the kernel behavior of the libOSs does not need to be identical to the original kernel.
- **ABI-level compatibility**: application programs (identical binary executable code) are runnable on libOSs without modifications, but the kernel behavior of the libOSs does not need to be identical to the original kernel.
- **Kernel-level compatibility**: application programs are runnable on libOSs with identical behavior of the original kernel.

We also summarize these levels in Figure 2 with the view of platform portability.

Source code-level compatibility: Unikernels are compact, single-address-space virtual machines that contribute to the lightweight execution. In the early days of unikernels,

the applications running on them were limited to being written in particular languages [36]; however, they have since evolved to apply to existing application programs as static binary executables that run as unikernels. Examples of this category are rumprun unikernel [30], LKL [44] without extensions described in this paper, and IncludeOS [6]. Because those unikernels must be built with a special standard library (i.e., `libc`), source code-level compatibility is offered. Thus, if users do not have access to the source code of programs, we cannot benefit from unikernels.

One exceptional design was considered by the Nabla container [60]: instead of using hardware virtualization, it takes processes as isolation primitives to host different unikernel instances. Because processes are universally available abstractions across various OSs, they contribute to platform portability. However, the Linux compatibility is inherited from its ancestor, rumprun unikernels, which can only offer POSIX API-level compatibility based on its NetBSD kernel.

ABI-level compatibility: System call translation is a technique used to run application programs on foreign OSs without recompiling source codes. Graphene [55], a library operating system offering application binary interface (ABI) compatibility with Linux userspace application, is designed based on its ancestors, Drawbridge [43], and Bascule [5]. The core libOS component attempts to translate system calls from userspace applications to one of the host kernels via the platform adaptation layer (PAL), which covers Linux, BSD, macOS, Windows host kernels, as well as the Intel Secure Guard Extension (SGX) with remote procedure calls forwarding to the host kernel [54]. Noah [48] and WSL [21] are also families using this technique. Because Linux is known to have a stable kernel ABI for userspace programs, the translation of system calls works mostly fine—even as the Linux kernel evolves.

However, the ABI-level compatibility is not sufficient because even if an application can be loaded without any visible errors on a libOS, it may behave differently than expected because of the missing or incomplete features of the kernel. For instance, in the design of Graphene, some system calls are simply forwarded (via shim layer and PAL) to the system call of the host kernel, in which case we can expect identical behavior if the host kernel is Linux (`setsockopt(2)` for instance). However, some other system calls are implemented inside the libOS, managing libOS-local states while interacting with the host kernel (e.g., `epoll_wait(2)`). The latter case is troublesome and is the reason for the lack of kernel-level compatibility. We found that the number of issues reported about epoll on the Graphene GitHub page was 65 (compared with 12 with setsockopt, at the time of writing), which is not negligible. We have confirmed this observation regarding its kernel-level compatibility. Such a reproduction of detailed behaviors, defined as **kernel-level compatibility**, is difficult to achieve if the underlying host kernel is not Linux.

System call translation can also be applied to alternate guest kernels that offer Linux ABI compatibility. Binary compatible unikernels (OSv [31], HermiTux [40]) and userspace kernels (gVisor [16]) are categorized in this family. Such alternate kernels can be designed with network stack libraries (e.g., lwIP [14], Seastar [9], netstack [17]), or porting kernel code for specific purposes (e.g., FreeBSD filesystem and network stack implementation in the case of OSv). The characteristics of the compatibility are almost identical to the system call translation; the translator works acceptably at the binary interface level, but some detailed Linux behavior may not be emulated.

Full virtualization: Lightweight VMMs take a different approach. Instead of creating lightweight kernels, rather than using heavy virtual machines, VMMs try to improve virtual machines by optimizing their design and VMM implementation while preserving the platform portability. Eliminating host dependency is a key technique to improve boot-time performance. Firecracker [1] and Kata container [52] are categorized in this technique. Docker Desktop [13] uses this approach to run a Linux container on foreign OSs. Although there is no compatibility concern because it can host the typical Linux guest OS, the resource footprint is identical to the general Linux VM, which might be a concern in resource-scarce environments.

Para-virtualization: X-container [50], a Xen-based container kernel, alleviates the issue of source code- and ABI-level compatibility approaches by using a standard Linux kernel as the libOS in a single application. These approaches also showed good system call throughput, but the applicability of the platform is limited to where Xen para-virtualization is available, which is not always universal.

User-mode Linux (UML) [11] is yet another Linux kernel running as a userspace program with the support of host OSs. Because its design is described as an architecture port of Linux kernels, all Linux kernel functionalities are available; thus, there is no compatibility concern either. Because it takes processes as an execution context, UML can become available in various environments. However, the current implementation requires a redesign to support running on non-Linux OSs or non-Intel CPUs, resulting in lower platform portability.

Discussion: As depicted in Figure 2, the design space of a libOS for Linux containers is broad, and developers of the extensions to container kernels have various choices. This can be because the kernel-space ABI of Linux is stable: the design and implementation of every libOS can concentrate on its additional values (e.g., small resource footprint, fault isolation from host kernels, and execution on non-Linux hosts) only if it guarantees ABI compatibility. However, to our knowledge, previous libOS studies do not achieve both Linux kernel-level compatibility and platform portability simultaneously.

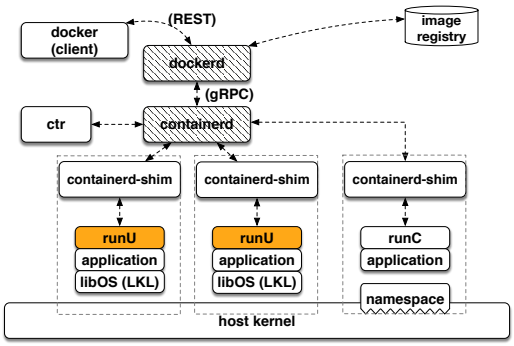


Figure 3. μKontainer overview: an OCI-compliant runtime environment (`runu`) handles the instance management of libOS processes. Shadow components are ported to macOS because there is no support on macOS.

Tsai et al. introduced a metric to measure the level of compatibility, called API importance [56], that presents the implementation coverage of popular system calls that applications use. While the API importance metric is useful in evaluating the completeness of interfaces that alternate container kernels expose, the metric does not capture the internal behavior of the kernels (i.e., bug-for-bug compatibility). Our compatibility goal in this paper is to explore an alternative design for the kernel-level compatibility to retain the mature and rich feature set of the Linux kernel.

3 Design and Implementation

The design of our system, μKontainer, follows the motivation described in the previous section to address the shortcomings of container runtime extensions. Our high-level goals are to 1) provide not only source code- and ABI-level compatibility of Linux applications but also kernel-level compatibility, 2) add a minimum amount of code to be able to run Linux kernels in different environments, and 3) introduce a minimal level of overhead. In this section, we explain the design goal and the expected benefits of μKontainer, introduce how an instance works with the detailed internals, how resources are isolated, and how to interact with the underlying container framework.

3.1 Overview

To address the shortcomings of existing alternative container kernels that replace the host Linux kernel, the key idea of μKontainer is to utilize the host processes as isolation primitive and insert a Linux-compatible container kernel running in userspace. This design choice is because of the availability of the process abstraction among various host OSs and platforms that contribute to the portability of μKontainer.

Regarding the alternate container kernel to offer a kernel interface to programs running on container instances, we take advantage of LKL [44] and extend it for use over container instances as an alternate container kernel, eliminating the kernel-level compatibility concerns of container kernels. Programs are automatically linked with the LKL by our modified cross-compilation toolchain `frankenlibc` [10] and `musl libc` [15] as a replacement for the standard library (`libc`) that most applications use. In addition, when the library is used for dynamic linkers and loaders of executable and linkable format (ELF) binaries, unmodified ELF binaries can use the LKL system call that preserves ABI-level compatibility. As a result, the code path of system calls goes to the LKL instead of host system calls.

A μKontainer instance runs as a single host process that includes all parts for running an application, i.e., an application binary, libraries, and configurations, as conventional containers. Unlike conventional containers, μKontainer contains the libOS as the container kernel inside its process. The libOS runs on top of the system call interface of underlying host OSs; therefore, the container kernels are decoupled from the host kernel. This decoupling offers extensibility and platform portability of the container kernel because we can expand and modify the container kernel independently from the host kernel.

Furthermore, the design of μKontainer is centered around our dedicated runtime environment, `runu`, which is an implementation of the open container initiative (OCI) runtime specification [41] that invokes μKontainer as a process. As an addition to existing runtime extensions used by Kata container [52] or gVisor [16], the new runtime environment redefines the behavior of container instances instead of the default runtime `runc`. The OCI compliance allows us to use the whole container infrastructure (i.e., `dockerd`, `containerd`, and image registry); therefore, users can take advantage of μKontainer via a drop-in runtime environment. Figure 3 is an overview of μKontainer using the `runu` process as an isolation primitive, whereas conventional containers using `runc` cooperate with the host kernel to instantiate a container.

The design of μKontainer results in four key components: a libOS (§ 3.2), an isolation primitive (§ 3.3), a platform abstraction layer (§ 3.4), and container engine integration (§ 3.5), which are described in the following sections.

3.2 LibOS: Linux Kernel Library

The core part of μKontainer relies on a libOS, i.e., LKL [44], which is an implementation of the anykernel architecture originally introduced by the NetBSD rump kernel [29, 30]. Unlike other userspace ports of the Linux kernel such as UML [11], LKL aims to be a reusable library in a variety of environments so that programs can link to the components of OS features implemented in the Linux kernel as libOSs. As illustrated in Figure 4, LKL introduces a hardware-independent architecture in the Linux kernel tree by decoupling the LKL host environment (machine/environment-dependent code) from the Linux kernel, largely contributing to the platform portability of this library. LKL is implemented as a patch set to the Linux kernel tree, and our version is currently based on version 4.19.0 of the kernel.

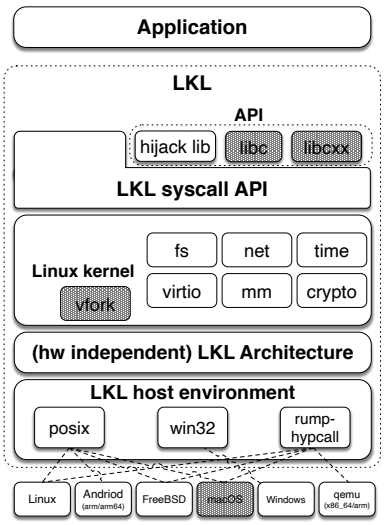


Figure 4. Structure of LKL as a portable and reusable library of the Linux kernel. The gray parts are extended components of this paper.

The patch set comprises 2.4 thousand lines of code (KLoC) for the environment-independent element and 11.8 KLoC for the environment-dependent element, potentially increasing in size if new underlying environments are added. Our extensions to LKL are indicated as shadow boxes in Figure 4, and we detail those extensions in § 4.

LKL uses Linux code as much as possible; thus, application interfaces implemented in a standard library and implemented system call interfaces to the kernel ideally have no compatibility issues with the identical behavior of the Linux kernel. Several existing libOS projects also claim Linux compatibility, such as IncludeOS [6], Graphene [55], EbbRT [49], and OSv [31]. gVisor [16] also pursues Linux compatibility. Their approaches vary, including binding the Linux standard library (glibc, musl, newlib) to their system call interfaces and exposing Linux-compatible ABI to applications. However, the source code- and the ABI-level compatibilities are not always perfect [56]. By using the codebase of the Linux kernel, our approach should offer better compatibility.

Using the Linux-kernel codebase with hardware-independent architecture has other benefits because there is almost no porting effort required even if there is a version update to the original kernel, while a porting effort is required by other librarifying approaches to the Linux kernel [51]. Furthermore, an application running with the LKL has rich features, such as the latest network protocols or algorithms, inherited from the Linux kernel.

3.3 Process: An Isolation Primitive

Isolation in conventional containers is performed via software-based partitioning implemented in Linux by introducing namespaces to various resources (network, mount, process, and user identifier [34]). The namespace-based isolation is reasonably efficient if users of container instances wish to share the host kernel while also wishing to have different instances from others.

Table 1. List of calls interacting with the host kernel, totaling 29 host calls in the Linux used.

	syscalls
process-related	arch_prctl,exit,kill,ucontext(3)
I/O related	poll,ppoll,pread,preadv,pwrite,pwritev,read, readv,write,writev,llseek,close,fcntl,fstat, fsync,lseek
clock info	clock_getres,clock_gettime,clock_nanosleep
memory related	mmap,mmap2,mprotect,munmap
others	getrandom,ioctl

However, this characteristic of depending on the host kernel restricts the choice of arbitrary features of containers because the container kernel is identical to the host kernel. As a result, the container can use only the features that the host kernel supports because the container and host kernels are not separated.

To relax this restriction, μKontainer uses the process as an isolation primitive for container instances. Instead of having a partition with a namespace in a host kernel, the kernel residing in the libOS offers a different kernel instance from the others that are isolated by the different virtual memory spaces provided to host processes. This design allows kernel extensions to be introduced to the container kernel because of the OS personality. Moreover, a container has the freedom of options to optimize the shape of a kernel in a selfish manner. For example, users can omit unnecessary features of the container kernel to shorten the code path.

The execution of μKontainer begins with an instantiation of a primary process, an OCI runtime process (`runu`), triggered by `containerd`. The primary process initializes the block and network devices used by the container instance via virtio [47] as communication channels. The container kernel is then initialized within the same primary process, continuing with a typical set of Linux kernel start-up routines. Lastly, the `main()` function of the application is called (also within the same process). This application can access the kernel space of libOS (LKL) via LKL system calls, but the system calls invoked are the function calls inside the userspace.

3.4 Platform Portability

Another benefit of using the userspace process as an isolation primitive is less effort to port μKontainer to new platforms. The clear interface from μKontainer to the host kernel is implemented via host calls (i.e., system calls and functions calls interacting with the host kernel), and we use those necessary calls by I/O operations from containers (block, network, and plan9 filesystem devices [27]) and access several outsourced resources including clock, memory, and program

scheduling. Therefore, the number of host calls used is relatively small; thus, developers are easily able to implement those host calls in various OSs. To validate the portability of μKontainer, we choose macOS as a new platform and examine the required effort to the port. Our implementation based on `frankenlibc` uses 29 (on Linux, shown in Table 1) and 23 (on macOS) host calls. The 23 host calls on macOS consist of about 1.2 KLoC of C and assembly code, which only requires a small amount of effort to develop. The coverage of the calls is slightly slimmer than that of Drawbridge [43]/Bascule [5] because our libOS (LKL) implements some of the host calls inside a libOS.

3.5 Docker Integration

To enrich the container environment via Docker, we have implemented several components for μKontainer over Docker infrastructure.

OCI runtime environment: The implementation of the OCI runtime environment (`runu`) has two aims: 1) to invoke a μKontainer process that includes a libOS, and 2) to bridge the interface of the OCI runtime specification. The implementation allows us to replace the default runtime environment (`runc`) with our custom one via a command-line option for the container invocation to preserve the portability of the container usage model.

Because of the cross-platform support of the Go language, the implementation of `runu` is quite straightforward on both Linux (x86_64, arm32, and aarch64 architecture) and macOS, which are the currently tested platforms. We have also integrated 9pfs server functionality [12] to share a filesystem layout from a container image to a container instance.

Docker image compatibility: μKontainer can optionally run with public container images under certain conditions to satisfy the image compatibility that is difficult to achieve for some container runtimes such as the Nabla container [60]. The shared library including the entire libOS code can be dynamically loaded by our custom program loader when `runu` runs with public container images. Our current implementation is compatible with musl libc-based Linux distributions (e.g., Alpine Linux).

`runu` replaces the system loader at the beginning of the execution and simply invokes the unmodified user-specified programs as the general process of libc replacement. On non-ELF host OSs, such as macOS, our (unmodified) ELF loader implemented in Linux kernel (binfmt_elf_fdpic.c) works together with the dynamic linker of musl libc, and our ELF loader performs the task instead of the host system's loader because the host cannot recognize ELF files and only loads Mach-O binary. As a result, an Alpine Linux image, which consists of ELF binary files, can run on macOS.

3.6 Limitations

Because of the nature of the method used to interpose system calls by a shared library, the current implementation of μKontainer cannot incorporate unmodified, statically linked executables. While the shared-library approach is beneficial because the libc replacement does not introduce an additional delay (as discussed later in § 5.3), even if the workload has heavy system call usage, this can present issues in terms of compatibility with existing container images; however, several approaches taken by existing studies, such as UML [11]/gVisor [16] (ptrace-based interposition) or HermiTux [40] (binary translation), can be applied to μKontainer to solve this limitation, and this will be considered in future work.

Another limitation of the current implementation concerns the fork implementation based on the `vfork` system call (details in § 4.1). Because vfork shares the address space with a child, a parent process must block its execution before the child process calls exec or exit.

4 Extensions to Linux Kernel Library

The development of the LKL, which is the heart of μKontainer, was started around 2007, and its source code was refactored in 2015 to fit into the latest Linux kernel. Since then, various attempts have been made to facilitate new use cases, including incorporating hardware-offload features of network devices [8] and placing new protocol implementation into mobile devices [42]. We have extended LKL upon those enhancements to design μKontainer because the vanilla version of LKL only provides the basic features of a container kernel. The vanilla version of LKL has neither existing application support nor ABI-level compatibility; thus, the kernel-level compatibility is not freely available without the extension. In this section, we highlight our contributions to LKL to implement μKontainer.

4.1 fork(2)/execve(2) Support

Although the nature of unikernels [36] starts with a single process execution and running multiple instances of unikernel was suggested for a workaround, discouraging multiple-process support degrades the level of compatibility for existing applications.

We have implemented vfork and execve system calls to improve the Linux kernel-level compatibility. Unlike the dependency on host facilities to implement the fork system call, as seen in Graphene and rump kernel, our implementation reuses the Linux implementation of the system calls with architecture-specific hooks for each system call implementation. As a result, multiple processes in LKL applications are viewed in a single host process; however, inside the host process, those are recognized as different LKL processes. This is similar to the design of X-Containers [50], as multiple processes inside a single container have no isolation.

For the fork implementation, we implemented vfork, as LKL architecture currently runs without memory management units (MMUs). Similarly, the ELF binary loader used in the execve system call supports dynamic-linked binary

consisting of position-independent code (FDPIC), which is a standard method for non-MMU architectures in Linux.

Note that executing ELF binary code on a non-ELF host needs special consideration of the differences in the binary interface. For instance, the usages of segment registers, with thread-local information access in particular, differ between Linux and macOS in x86_64 processors. We applied a workaround to convert binary upon accessing such registers to use an appropriate register[1].

4.2 C/C++ Standard Library Bindings

With original LKL implementation, applications benefit from LKL if 1) developers manually rewrite system call usages of applications for LKL system call API, or 2) replace C symbols of system calls by dynamically rewriting them during the start-up process of applications (e.g., LD_PRELOAD feature of dynamic linker). The former requires a considerable amount of porting effort as the size of code increases, while the latter alleviates this time-consuming task but has a limitation: the dynamic rewriting only works for visible symbols, whereas there are some hidden symbols where the rewriting does not work.

To have dozens of existing compatible Linux applications published as container images over μKontainer, we have implemented the standard library bindings for applications that use LKL for the underlying kernel. We have ported musl libc for C applications and libcxx from LLVM for C++ applications. In addition to a build toolchain of `frankenlibc`, which was originally a cross-compilation environment for NetBSD rump kernel, the source code of existing applications can be built as statically linked executables that automatically use LKL-syscall API for their system call invocations. Optionally, the build toolchain can generate a shared library (i.e., `libc.so`). This library is used to install an existing Linux distribution to provide binary compatibility with applications without modifying their source code.

4.3 macOS Platform Port

Our extended LKL and `frankenlibc` contain macOS platform support, as outlined in § 3.4. This port is straightforward regarding the clear layering of LKL and `frankenlibc` (i.e., layering across applications, libc, kernel, (rump) hypercall, and the machine/underlying OS-dependent layer). However, one specific challenge to porting a libOS to a different host OS, particularly for the LKL used in our case, is when there is a different binary format. A study during LKL development indicated that the portable executable (PE) binary used by Windows does not require many changes to the original ELF-based implementation of the Linux kernel, whereas the Mach-O binary used by macOS requires a variety of translations in various places. Our LKL port to macOS includes a manual relocation of symbols at the link stage where the Linux kernel is used heavily in its initialization processes, whereas the Apple toolchain (i.e., LLVM) does not have a linker-script feature for the relocation. The differences between the Apple and Linux toolchains, including the assembly language syntax, required us to implement a substantial amount of glue code, which could be eliminated if we can modify the Apple toolchain.

[1] https://git.io/fjHHM

5 Evaluation

Our three-part evaluation aimed to answer the following questions.

- *What degree of kernel-level compatibility do container kernels achieve in terms of network protocol implementation?* (Linux compatibility, § 5.1).
- *How much benefit is a container on a non-Linux (foreign) OS to running an application?* (platform portability, § 5.2).
- *What level of overhead does the new design introduce as a result of gaining the new features?* (§ 5.3).

All our evaluations focus on how the proposed container architecture benefits or detracts from our extended libOS (i.e., LKL) in various use cases.

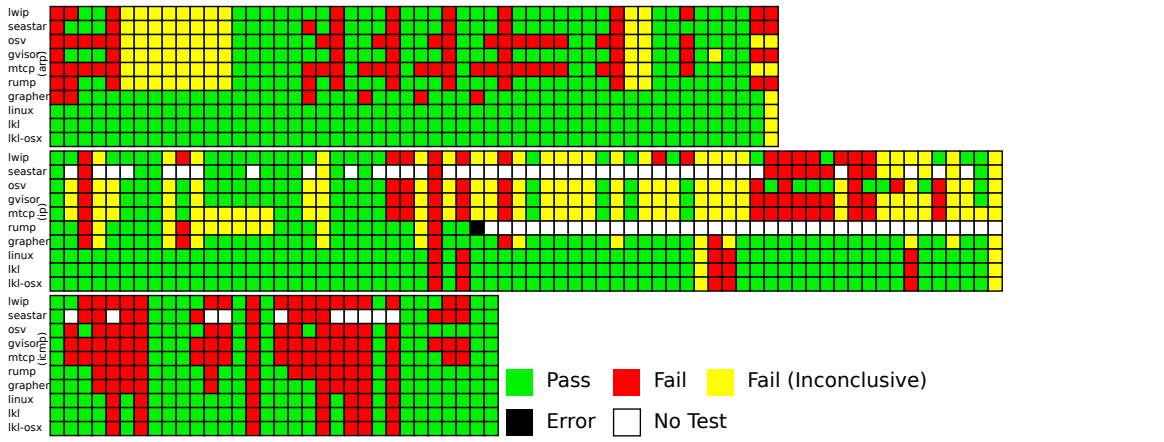
Note that our evaluations are centered around the behavior of network stacks to highlight the impact of the design choice for the libOS of interests. With whatever designs taken for library OSs (with system call translation, userspace kernel, para-virtualization), this incompatible behavior makes applications running on the alternate container kernel impossible to work as expected in the worst case. This is particularly critical with the network-related features for two reasons: 1) the incompatible behaviors may disrupt connections with other nodes that behave correctly, and 2) network stacks usually have a large number of parameters (e.g., `sysctl -a |grep net`) and configuration interfaces (e.g., netlink, /proc files, socket options, control messages (cmsg), flags (MSG_*)) that are not easily reproducible.

5.1 Linux Compatibility

If the container kernel cannot be extended to allow for the inclusion of new features such as network protocol extensions, one option for container users is to implement the required extension inside an application (in userspace). Various libraries are available for this particular purpose, such as lwIP [14], Seastar [9], mTCP [25], rump kernel [29] (used by Nabla container), and netstack [17] (used by gVisor), that implement network stack functionality in userspace by bypassing the network stack of the (host) kernel. However, reimplementing a network stack from scratch requires a large amount of effort because the implementations vary widely, and a level of maturity and compatibility with the original one (Linux kernel in this case) is not easily achievable. Furthermore, network stack implementations in conventional monolithic kernels usually have a broader configuration interface than other subsystems such as filesystem (`ifconfig`, `route`, and additional commands v.s., `mount()`

	stack	Quagga	Quagga-kernel	ARP	IPv4	ICMPv4
lwIP [14]	original	–	–	31/52	27/68	14/32
Seastar [9]	original	–	–	32/52	12/27	10/22
OSv [31]	FreeBSD-based	–	–	20/52	28/68	17/32
gVisor [16, 17]	original	running	incomplete (no /proc, AF_NETLINK)	31/52	21/68	11/32
mTCP [25]	original	–	–	16/52	15/68	12/32
rump [29]	NetBSD	–	–	31/52	17/31	19/32
Graphene [55]	Linux	running	incomplete (no /proc, AF_NETLINK)	45/52	51/68	21/32
Linux	Linux	running	good	51/52	61/68	25/32
LKL	Linux	running	good	51/52	61/68	25/32
LKL (macOS)	Linux	running	good	51/52	61/68	25/32

(a) Test configurations and scores (PASS/Total).



(b) Detailed test results: ARP, IPv4, and ICMPv4 tests (top to bottom). X-axis: test numbers; Y-axis: network stacks. The failures which do not complete a test are indicated as *Fail (inconclusive)*

Figure 5. Conformance test results (IxANVL) for network protocol based on RFC specifications.

system call [29]); thus, this measurement should well capture the kernel-level compatibility in depth.

We measured the level of maturity and kernel-level compatibility of Linux by testing the conformance of the network stack implementation. We used Ixia IxANVL (automated network validation library) [24], a software utility to validate network protocol compliance and interoperability. By running a set of test suites to verify the behavior of the network stack, based on standard specifications of IETF RFCs, the tool reports the number of successful tests for each network stack implementation. We use the reported number as the level of maturity of network stacks.

Regarding the network stack under tests, we used lwIP (git 7b7bc349 revision), Seastar (git c19219ed revision), OSv version v0.24, gVisor (git faa34a0 revision), mTCP (git 611cc05d revision), Graphene (git 200452cd revision), rump kernel (git f10683c revision of buildrump.sh), LKL (git 5221c547af3d revision based on Linux 4.16.0 version), and native Linux kernel (4.15.0-34 version). We also used LKL over a macOS host to demonstrate the platform portability on a foreign OS. We used IxANVL version 9.19.9.32 and ran the following test suites for the conformance tests: ARP, IPv4, and ICMPv4.

Figure 5a reports the number of tests passed (succeeded) with all the tested network stacks, and Figure 5 breaks down each test in a matrix view. Note that some of the tests rely on additional configurations to the network stack during tests. Because of the higher kernel-level compatibility, we can use the `zebra` daemon of Quagga [45] software with Linux kernel and LKL (on both Linux and macOS) without any porting effort to dynamically configure the state of the network stack. In contrast, there are no such interfaces in other network stacks other than the command-line parameters at the beginning of the process. This gives better results in LKL and Linux. gVisor and Graphene can also run Quagga as they offer the ABI-level compatibility, but because of the lack of implemented system calls and kernel interfaces (i.e., proc filesystems and netlink sockets), Quagga does not help to improve the result of tests. Figure 5a additionally includes the conditions of each network stack. Among all network stacks, Seastar cannot create multiple network interfaces; thus, some of the tests (e.g., IPv4 and ICMPv4 tests) were not conducted with IxANVL. Rump kernel, which is based on a similar approach with LKL by reusing NetBSD network stack code, scored higher success rates on most tests but had fatal

errors in the IPv4 tests, resulting in program halts during the tests (thus, the number of total tests is smaller than others). LKL and Linux stacks present higher numbers of successful tests, while others report more failures in corner cases. An example of failure is the test that should trigger IP forwarding to the received interface where a network stack usually sends back an ICMP redirect message. LKL and Linux correctly behave as specified in RFC1812 [4], but the others do not, resulting in failures. Furthermore, the results of LKL and Linux kernel are identical, even testing over macOS, and are thus compatible in their behavior regarding network stacks. This fact is obvious because we used the same source code between Linux and LKL, but no other implementations can preserve this level of compatibility.

The ABI-level compatibility, which gVisor and Graphene offer, scores worse results than for kernel-level compatibility (i.e., LKL). Even if there is no visible compatibility issue to invoke applications with gVisor and Graphene, the internal behaviors of system calls and access kernel interfaces (proc filesystem) are not compatible. This is a clear example showing that the ABI-level compatibility is not sufficient.

The results in this experiment only reveal a part of the Linux compatibility and feature richness of network stack implementations. Although implementing additional tests, such as those with transport protocols and other network protocols (e.g., IPv6), is usually more complicated, they provide a more precise understanding of the compatibility. We will conduct more tests in the future; however, the collected information of this experiment also confirms that reimplementing the network stack from scratch would reduce the maturity of the implementation because of the lack of the kernel-level compatibility with the Linux kernel, which would yield troublesome interoperability in real-world deployment and application portability.

5.2 Platform Portability

The next demonstration is to use μKontainer with Linux-specific features to improve the network performance of a foreign OS (a macOS case). Only μKontainer can use Linux-specific features on macOS without any porting effort because of the higher level of portability. We established a simple network topology connecting three machines (a server, client, and middlebox) via 1-Gbps links and injected TCP traffic via `netperf`. The objective of the experiment is to present the strength of the congestion-control algorithm, Bottleneck Bandwidth, and Round-trip propagation time (BBR) [7], which is not available in the current macOS. Our experiment assumes the following scenario: suppose a mobile terminal uploads a large amount of data to a server, but the link between the client and the server is unstable and even has packet loss. One possible solution in this situation is to try a different congestion-control algorithm of a transport protocol (i.e., TCP). If the host OS does not have sufficient algorithms, then users might ask for help from foreign OSs using virtualization to provide different algorithms to the TCP stack so the data can be quickly uploaded. μKontainer applies to such a scenario because it does not use a full guest OS instance but instead uses only an application-embedded network stack in a container that solves a particular problem of the host OS.

Table 2. Goodput with BBR or Cubic congestion-control algorithms on macOS.

Algorithm	Goodput (Mbps)	Stddev
docker-cubic	4.59	0.56
ukontainer-bbr	97.15	5.85
ukontainer-cubic	7.20	4.54
native-cubic	5.38	3.77
noah-cubic	5.13	2.37

We used `netperf` with the synthetic packet losses and delay (1% loss and 100 ms delay) at the middlebox, and the client used a MacBook running High Sierra (10.13.6), with 4-core 2.5 GHz Intel Core i7, and 16 GB memory, connected to the Linux server via a 1-Gbps Ethernet cable. Using this setup, we compared μKontainer, Noah [48] (a system call translator of Linux binary on macOS) version from git 2e570f7 revision, Docker for Mac [13] version 18.03.1-ce, and the host macOS (native). μKontainer uses two congestion-control algorithms, BBR and Cubic [18], while others only use Cubic. Noah and the native host use the default algorithm of macOS, while Docker for Mac uses algorithms in guest Linux (i.e., LinuxKit [38]), which does not have BBR installed out of the box; thus, this case uses the default algorithm (i.e., Cubic).

Table 2 shows the goodput result of the netperf benchmark of TCP_STREAM from five iterations. Because the Cubic algorithm uses packet loss as a congestion signal (unlike BBR), the goodput is lower than for BBR because of its smaller window sizes during transmission. However, by using carefully measured round-trip times with packet pacing using the Linux `qdisc` subsystem in LKL on macOS, BBR can efficiently utilize the bandwidth even under poor network conditions.

Running a foreign platform's kernel on userspace (the process in μKontainer) rather than emulating a userspace program of a foreign platform (as Noah does) is also possible by hardware virtualization with the guest OS instance, as introduced by Docker, and may achieve similar results if the Linuxkit of Docker supports BBR. However, the approach presented in this demonstration offers application-specific installation without involving the host system to work around the issue we faced in our scenario (i.e., unstable network condition).

5.3 Benchmarks

This section presents the result of benchmarks where we try to identify how much overhead μKontainer introduced. Because all (original) Linux kernel functionalities are executed

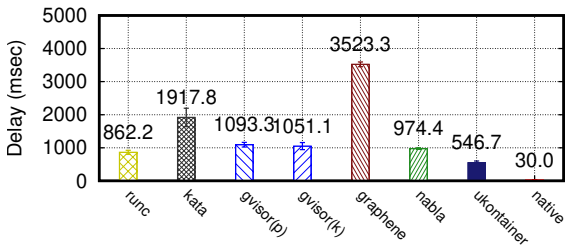


Figure 6. Duration of Python script execution from 30 measurement iterations (with the mean values).

in userspace processes, understanding the basic performance of μKontainer helps when presenting the concerns regarding the lower performance on userspace execution.

Setup: Unless stated otherwise, our experiments were conducted on two machines, Dell PowerEdge R330 with a 4-core 3.8 GHz Xeon E3-1200 and 64 GB memory, interconnected via an Intel X540 10-Gbps link. The machines ran the Linux 4.18.5 kernel of Fedora 28 and used Docker 18.06.1-ce for the container framework. To compare μKontainer with other approaches, we used the Kata container version 1.8.0 with the 4.19.28-48.1 Linux kernel, gVisor from the git e9ea7230 revision, Graphene with the git 200452cd revision (we used Graphene shielded containers (GSC) to build a container image for tests), Nabla containers [22] from the git 2cecc88 revision, a native Linux application on the host kernel without containers, and μKontainer with the `runu` runtime environment.

5.3.1 Fast Cold Start. Prior studies have revealed that containers are often used as platforms for serverless computing, such as AWS Lambda, Azure Functions, or Google Cloud Functions, but have the shortcomings associated with program agility and function instantiation [32, 39, 58]. In this experiment, we try to understand how μKontainer performs in such an environment comparing with different container runtimes.

We measured the duration of a simple Python program execution until the program is ready to listen to a socket. Although the result involves other preparatory elements, such as filesystem and network interface creation and several message interactions between the container engine and runtime to manage the container lifecycle, we used the `docker run` command to invoke a container instance because it reflects the typical use cases of serverless deployments.

Figure 6 plots the result of this measurement with the standard deviation from 30 repetitions. The duration of the (Linux) native Python program is about 30 ms, and this can be used as a baseline for typical process instantiation in the host system. The standard Docker runtime environment (`runc`) takes 862 ms, requiring configurations such as signal handler installations, system call filtering to the host kernel, followed by multiple process invocations. gVisor (`gvisor(p)`, gVisor with ptrace system call trap, and `gvisor(k)` with a trap via KVM) do not utilize the namespace facility, but they require more time (1051 ms and 1093 ms) because of the overhead of the context switches across multiple system calls. Nabla container (`nabla`) shows a longer duration than `runc` and `runu`, which has a time of 974 ms, and this is also slower than that presented by Williams et al. [60] because our measurement involves interactions with `containerd` and OCI runtime while their paper may not include these considerations. The Kata container (`kata`) takes 1918 ms, and this is shorter than a typical virtual-machine instantiation but slightly longer than that of the other approaches and represents the cost of transparent hardware virtualization. Graphene shows the longest duration because GSC, a tool for creating a container image for Graphene, generates a substantial amount of initialization code before the program execution, resulting in a start-up time of 3523 ms, although this might be a trade-off for preparing a sandbox runtime in advance. μKontainer with the `runu` runtime environment takes 547 ms, and this is the fastest among all OCI runtimes including the result of `runc`, although it also involves an additional filesystem mount to load Python library files. In its best case, the μKontainer with the `runu` is faster than the Kata container by a factor of 3.5. Although we did not investigate this, the boot process of the Linux kernel with LKL can be shortened if we strip out unused parts of the kernel configuration (e.g., `make tinyconfig`). This kind of container-centric optimization is one of the strengths of both μKontainer and the Kata container and is obtained by taking advantage of their agility.

Table 3. Memory footprint (Mbytes).

	hello	nginx	python
runc	0.004	3.932	4.94
kata	169.560	168.868	171.660
gvisor	15.100	17.408	17.896
graphene	1.384	8.676	15.156
ukontainer	5.94	7.136	14.776

5.3.2 Memory Footprint. The next evaluation is the measurement of the memory footprint with three simple programs: a C-based hello-world style program (only print a line to console and sleep), nginx web server after its start-up, and a python hello-world style program (same behavior as the C-based one).

We compared the following five runtimes: the default Docker runtime (`runc`), Kata container (`kata`), gVisor (`gvisor(p)`), `Graphene`, and μKontainer (`runu`). Because we tried to determine the difference in memory consumption between container kernels, we only measured the resident set size of the main programs to eliminate the other consumption by control processes (e.g., `containerd-shim`, OCI runtimes, and helper programs such as `runsc-gofer` in `gvisor`).

Table 3 shows the results of the measurement. In the experiment, we investigated the memory information of the

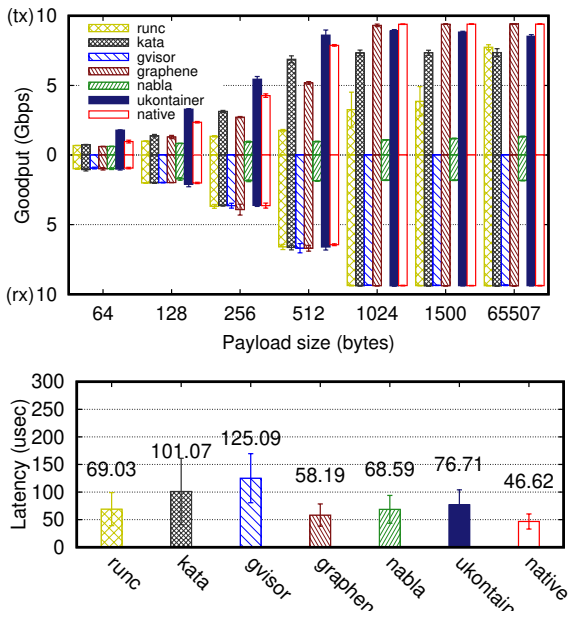


Figure 7. Results for netperf Tx and Rx goodput measurement (top) and latency measured (bottom), with standard deviations.

processes (hello, nginx, python) for `runc` and μKontainer because those appear as host processes, while we measured qemu-lite-system-x86_64 for `kata`, pal-Linux for `Graphene`, and runsc-sandbox for `gvisor(p)` because processes on those runtimes do not appear in the host kernel.

Of all of the runtimes, `runc` consumes the smallest amount of memory because the other four runtimes contain not only the code of target programs but also the container kernel inside those processes. The result of `gvisor(p)` and μKontainer is comparable, while `kata` clearly shows the larger footprint. Graphene shows a modest resource consumption, as its kernel part is not included in the program, but in the host kernel, resulting in a small addition to the runc results. `gvisor(p)` and `kata` may be configured to use smaller memory, but we cannot determine the configuration method; consequently, we used the default one.

5.3.3 Network-I/O Workload. To confirm how much overhead we could observe from the implementation, we conducted a simple benchmark test with a network-I/O-intensive workload on both the transmit (Tx) and receive (Rx) paths using `netperf`. We used the Dell machines with the seven variants of client software (Docker, Kata container, gVisor, Graphene, Nabla container, μKontainer, and native Linux). The server side (netserver) always used the native Linux program. We enabled network interface card (NIC) offload features (i.e., TCP segmentation offload (TSO) and checksum calculation) if the container runtime environments supported them. Thus, we did not use them with gVisor and Nabla because they seem not to be currently supported. In all the experiments, we used the `docker0` bridge interface attached via virtual NIC. All approaches except μKontainer used a veth device, whereas μKontainer used a tap device connected to the `docker0` bridge.

Figure 7 (top) shows the results of the TCP_STREAM (Tx side) and TCP_MAERTS (Rx side) goodput for 10 seconds of transactions with various payload sizes. The overall Tx results showed similar trends except for gVisor and Nabla. The results of gVisor are almost invisible in the figure (less than 1 Mbps) and this is as expected (and as advertised) because the system call interposition based on ptrace(2), which involves expensive processing, is slow under a heavy system call workload [16]. We also tested the KVM implementation of gVisor but did not see any improvement in this experiment. The result of Nabla is slower in most conditions mainly because of the lack of TSO. The Tx experiments gave similar results for the Kata container (`kata`), μKontainer (`runu`), Graphene, and the native Linux program (native). For these approaches, the similarity is caused by efficient packet handling with the TSO mechanism. However, μKontainer outperformed the native approach for small packet sizes (84% faster for 64-byte packets), whereas the improvements are no longer seen in larger packet sizes. The Kata container uses the virtio network driver in the same manner as μKontainer and is expected to present similar results; however, the Kata container uses additional veth devices in the current design [53], resulting in a slightly worse performance overall than the performance of μKontainer. Graphene shows a similar performance with the native Linux as it uses a host Linux network stack as-is. The default Docker runtime environment (`runc`) also works with TSO but could not fill the 10-Gbps pipe, being slower than μKontainer by a factor of two for a 1024-byte payload. The observation with the packet capture confirms that the aggregation of the transmitted packets was not always performed when the payload size was under 1500 bytes, resulting in a poor goodput with `runc`.

Figure 7 (bottom) shows the latency measured by Ping-Pong packets with 1-byte payload size (conducted by TCP_RR test of `netperf` for a duration of 20 seconds). Overall, the results of μKontainer are comparable with the results of `runc`: in the worst case, μKontainer has 65% longer latency to native execution.

6 Discussion

The design choice of using processes as an isolation primitive raises a question: *What about security properties?* Although isolation by software partitioning is often considered weaker protection than hardware-based isolation, μKontainer's pure process-based container can be considered as offering much weaker isolation. Moreover, if a user intentionally bypasses the container kernel and communicates with the host kernel, the isolation is even worse. This could

be alleviated by applying the existing approaches to protection used by conventional containers (e.g., the seccomp filter in Linux or sandbox_init in macOS) to restrict the operation from container instances. In contrast, using processes as an isolation primitive yields other benefits. Williams et al. [60] presented the advantage of performance as well as a smaller attack surface when running unikernels as processes, in addition to the classical argument of robustness that avoids a whole system crash in userspace executions.

Trade-off between performance and compatibility: Compatibility is often compromised because of the design decision of the target system. While network and storage stack-bypass technologies often require applications to be rewritten (or ported) to obtain performance benefit [9, 14, 17, 23, 25, 37, 61], the bypass technique itself does not require incompatibility, as demonstrated in § 5.1: μKontainer bypasses network and storage stack but preserves Linux compatibility. Users can choose the options based on their own target system requirements.

7 Related Work

There are several areas of technology related to the work in this paper, and we outline the relationships of previous work to μKontainer in this section.

System call translation (WSL [21], Noah [48], Linuxlator [20]) is a method for executing a userspace program on a foreign OS. Translations in the function body include system call numbers, conversions of different structures (e.g., `struct sockaddr` between Linux and BSD), and nonexistent system calls (e.g., `epoll_create(2)`). Above this component, a Linux container image can run on a foreign OS: e.g., Docker on FreeBSD and WSL work on this mechanism without having to recompile the binary image for different target OSs. Although the emulation is usually minimized to only the different parts of two components (i.e., ABI and host system calls), the translation is often not trivial. This result is because of the large number of system calls (e.g., 333 system calls in Linux 4.15 on x86 architecture) and the unstable interface of the underlying host kernel, often making kernel-level compatibility difficult for system call translation.

The hypervisor-based container (Kata container [52], Docker for Mac [13], and WSL2 [35]) is the most transparent way to host foreign userspace programs on various OSs in terms of application compatibility. Although both the Kata container and Docker for Mac have improved overheads in their resource footprints, they still require relatively higher memory even if users only wish to invoke a single application, as shown in § 5.3.2. This overhead can be neglected if users instantiate a second application, but not during the cold-start phases, as discussed in § 5.3.1.

Unikernel [36] in the context of container technology has been envisioned as a possible future direction based on the core idea of library OS, which was first expressed in exokernel [28] in the late 1990s. The basic idea is to deploy a small guest OS implementation above hardware virtualization (OSv [31]) or an isolated, secured, userspace application environment via a processor extension (SCONE [2]). Nabla container [60] is proposed to shrink the host kernel surface by using a similar idea with μKontainer, *unikernels as userspace processes*. Lupine Linux [33] takes a different approach to offer unikernel properties in a standard VM form. A recent contribution to Unikernel Linux (UKL) [46] shares the motivation of ours, utilizing fully compatible, rich features of the Linux kernel in a different shape. Our proposed approach is effectively a unikernel adaptation in a userspace execution environment, and with carefully preserving kernel-level compatibility and feature richness, which are not easily achieved because these features require significant development effort to the network stack, filesystems, and system call interface, as discussed in § 5.1,

8 Conclusion

We have comprehensively reviewed the software architecture of alternate container kernels with libOSs and have identified issues with the current state of the art. This review examines the core design of our proposed μKontainer system, which uses a Linux kernel in a library mode. We have evaluated our proposed system in terms of use cases that demonstrate the kernel-level compatibility of network stack implementation and platform portability (a Linux kernel feature on macOS application). The benchmarks for network-intensive workloads reveal comparable results with those of native processes, which are achieved at the userspace execution of μKontainer.

We believe that μKontainer is a practical implementation of libOS based on the evaluation with performance, overhead, and implemented features. Our implementation can still be improved, including further minimization of the libOS implementation for an even smaller resource footprint. Such improvements make μKontainer valuable, promoting the continued evolution of the container ecosystem as a virtualized computing environment.

Acknowledgments

The research leading to these results has been supported by the EU-JAPAN initiative by the EC Horizon 2020 Work Programme (2018-2020) Grant Agreement No. 814918 and Ministry of Internal Affairs and Communications "Strategic Information and Communications R& D Promotion Programme (SCOPE)" Grant no. JPJ000595, "Federating IoT and cloud infrastructures to provide scalable and interoperable Smart Cities applications, by introducing novel IoT virtualization technologies (Fed4IoT)." The authors thank Keysight Technologies Japan K.K. for the support of the IxANVL tests. We also wish to thank Aniketh Girish for his contributions to container-engine integration, and Sho Yuhara for his supports of experiments.

References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association.

[2] 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 *OSDI*, volume 16, pages 689–703, 2016.

[3] Arnd Bergmann. [PATCH 00/16] remove eight obsolete architectures. http://lkml.iu.edu/hypermail/linux/kernel/1803.1/06845.html. (Accessed January 14th 2020).

[4] F. Baker. Requirements for IP Version 4 Routers. RFC 1812 (Proposed Standard), June 1995. Updated by RFCs 2644, 6633.

[5] Andrew Baumann, Dongyoon Lee, Pedro Fonseca, Lisa Glendenning, Jacob R. Lorch, Barry Bond, Reuben Olinsky, and Galen C. Hunt. Composing OS Extensions Safely and Efficiently with Bascule. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 239–252, New York, NY, USA, 2013. ACM.

[6] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E Engelstad, and Kyrre Begnum. IncludeOS: A minimal, resource efficient unikernel for cloud services. In *Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on*, pages 250–257. IEEE, 2015.

[7] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-based Congestion Control. *Commun. ACM*, 60(2):58–66, January 2017.

[8] HK Jerry Chu and Yuan Liu. User Space TCP-Getting LKL Ready for the Prime Time. *Linux Netdev 1.2*, October 2016.

[9] Cloudius Systems. Seastar. http://www.seastar-project.org/. (Accessed Jan 26th 2017).

[10] Justin Cormack. frankenlibc. https://github.com/justincormack/frankenlibc. (Accessed Jan 26th 2017).

[11] J. Dike. User Mode Linux. In *Proceedings of the 5th Anual Linux Showcase and Conference*, ALS'01, pages 3–14. USENIX Association, 2001.

[12] Docker Inc. A modern, performant 9P library for Go. https://github.com/docker/go-p9p.

[13] Docker Inc. Docker for Mac. https://www.docker.com/docker-mac. (Accessed Apr 18th 2018).

[14] Adam Dunkels. Design and Implementation of the lwIP TCP/IP Stack. *Swedish Institute of Computer Science*, 2:77, 2001.

[15] Rich Felker. musl libc. https://www.musl-libc.org/. (Accessed Jan 26th 2017).

[16] Google Inc. gVisor: Container Runtime Sandbox. https://github.com/google/gvisor. (Accessed May 8th 2018).

[17] Google Inc. IPv4 and IPv6 userland network stack. https://github.com/google/netstack. (Accessed Sep 14th 2018).

[18] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[19] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible Network Experiments Using Container-based Emulation. In *Proceedings of ACM CoNEXT 2012*, pages 253–264. ACM, 2012.

[20] Brian N. Handy, Rich Murphey, and Jim Mock. Linux Binary Compatibility. https://www.freebsd.org/doc/handbook/linuxemu-lbc-install.html. (Accessed Apr 18th 2018).

[21] Mike Harsh. Run Bash on Ubuntu on Windows. https://blogs.windows.com/buildingapps/2016/03/30/run-bash-on-ubuntu-on-windows/. (Accessed Apr 18th 2018).

[22] IBM. Nabla Containers. https://github.com/nabla-containers/runnc. (Accessed July 3rd 2019).

[23] Solarflare Communications Inc. OpenOnload. http://www.openonload.org/. (Accessed 14th January 2015).

[24] Ixia. IxANVL. https://www.keysight.com/us/en/products/network-security/ixanvl.html. (Accessed March 8th 2021).

[25] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, Seattle, WA, April 2014. USENIX Association.

[26] Jessie Frazelle. Docker Containers on the Desktop. https://blog.jessfraz.com/post/docker-containers-on-the-desktop/, 2015. (Accessed Aug 15th 2018).

[27] Venkateswararao Jujjuri, Eric Van Hensbergen, Anthony Liguori, and Badari Pulavarty. VirtFS–a virtualization aware file system pass-through. In *Ottawa Linux Symposium (OLS)*, pages 109–120, 2010.

[28] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, SOSP '97, pages 52–65, New York, NY, USA, 1997. ACM.

[29] Antti Kantee. *The Design and Implementation of the Anykernel and Rump Kernels, 2nd Edition*. http://book.rumpkernel.org, 2016.

[30] Antti Kantee and Justin Cormack. Rump Kernels: No OS? No Problem! *USENIX ;login:*, 39(5):11–17, 2014.

[31] 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 *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 61–72, Philadelphia, PA, June 2014. USENIX Association.

[32] Ricardo Koller and Dan Williams. Will Serverless End the Dominance of Linux in the Cloud? In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 169–173, New York, NY, USA, 2017. ACM.

[33] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. A linux in unikernel clothing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[34] Linux Programmer's Manual. namespaces - overview of Linux namespaces. http://man7.org/linux/man-pages/man7/namespaces.7.html. (Accessed Sep 14th 2018).

[35] Craig Loewen. Announcing WSL 2. https://devblogs.microsoft.com/commandline/announcing-wsl-2/. (Accessed July 24th 2019).

[36] 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.

[37] Ilias Marinos, Robert N.M. Watson, and Mark Handley. Network Stack Specialization for Performance. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 175–186, New York, NY, USA, 2014. ACM.

[38] Moby Project. LinuxKit. https://github.com/linuxkit/linuxkit. (Accessed Sep 14th 2018).

[39] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'18)*, 2018.

[40] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. A Binary-compatible Unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution*

Environments, VEE 2019, pages 59–73, New York, NY, USA, 2019. ACM.

[41] Open Container Initiative. OCI Runtime Specification. https://github.com/opencontainers/runtime-spec. (Accessed Sep 14th 2018).

[42] Cristina Opriceana and Hajime Tazaki. Network stack personality in Android phone. In *Linux netdev 2.2*, THE Technical Conference on Linux Networking netdev 2.2, 2017.

[43] 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.

[44] Octavian Purdila, Lucian Adrian Grijincu, and Nicolae Tapus. LKL: The Linux kernel library. In *Roedunet International Conference (RoEduNet), 2010 9th*, pages 328–333, 2010.

[45] Quagga. Quagga Routing Suite. https://www.quagga.net/. (Accessed Sep 14th 2018).

[46] Ali Raza, Parul Sohal, James Cadden, Jonathan Appavoo, Ulrich Drepper, Richard Jones, Orran Krieger, Renato Mancuso, and Larry Woodman. Unikernels: The Next Stage of Linux's Dominance. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, pages 7–13, New York, NY, USA, 2019. ACM.

[47] Rusty Russell. Virtio: Towards a De-facto Standard for Virtual I/O Devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, July 2008.

[48] Takaya Saeki, Yuichi Nishiwaki, Takahiro Shinagawa, and Shinichi Honiden. Bash on Ubuntu on macOS. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, APSys '17, pages 17:1–17:8, New York, NY, USA, 2017. ACM.

[49] 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, November 2016. USENIX Association.

[50] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. X-Containers: Breaking Down Barriers to Improve Performance and Isolation of Cloud-Native Containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 121–135, New York, NY, USA, 2019. ACM.

[51] Hajime Tazaki, Ryo Nakamura, and Yuji Sekiya. Library operating system with mainline Linux kernel. In *Linux netdev 0.1*, THE Technical Conference on Linux Networking netdev 0.1, 2015.

[52] The OpenStack Foundation. Kata Containers. https://katacontainers.io/. (Accessed Aug 15th 2018).

[53] The OpenStack Foundation. Kata Containers Architecture. https://github.com/kata-containers/documentation/blob/master/design/architecture.md. (Accessed January 15th 2020).

[54] Chia-Che Tsai. Library OS is the New Container. https://sched.co/FxXc. (Accessed November 21st 2019).

[55] 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.

[56] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. A Study of Modern Linux API Usage and Compatibility: What to Support when You'Re Supporting. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 16:1–16:16, New York, NY, USA, 2016. ACM.

[57] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC '17)*, pages 645–658, 2017.

[58] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146. USENIX Association, 2018.

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

[60] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. Unikernels As Processes. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 199–211, New York, NY, USA, 2018. ACM.

[61] Irene Zhang, Jing Liu, Amanda Austin, Michael Lowell Roberts, and Anirudh Badam. I'm not dead yet!: The role of the operating system in a kernel-bypass era. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, pages 73–80, New York, NY, USA, 2019. ACM.