KylinX: Simplified Virtualization Architecture for Specialized Virtual Appliances with Strong Isolation

YIMING ZHANG, CHENGFEI ZHANG, YAOZHENG WANG, and KAI YU,
NiceX Lab, National University of Defense Technology
GUANGTAO XUE, Shanghai Jiao Tong University
JON CROWCROFT, University of Cambridge

Unikernel specializes a minimalistic LibOS and a target application into a standalone single-purpose virtual machine (VM) running on a hypervisor, which is referred to as (virtual) *appliance*. Compared to traditional VMs, Unikernel appliances have smaller memory footprint and lower overhead while guaranteeing the same level of isolation. On the downside, Unikernel strips off the *process* abstraction from its monolithic appliance and thus sacrifices flexibility, efficiency, and applicability.

In this article, we examine whether there is a balance embracing the best of both Unikernel appliances (strong isolation) and processes (high flexibility/efficiency). We present KylinX, a dynamic library operating system for simplified and efficient cloud virtualization by providing the pVM (process-like VM) abstraction. A pVM takes the hypervisor as an OS and the Unikernel appliance as a process allowing both page-level and library-level dynamic mapping. At the page level, KylinX supports pVM fork plus a set of API for inter-pVM communication (IpC, which is compatible with conventional UNIX IPC). At the library level, KylinX supports shared libraries to be linked to a Unikernel appliance at runtime. KylinX enforces mapping restrictions against potential threats. We implement a prototype of KylinX by modifying MiniOS and Xen tools. Extensive experimental results show that KylinX achieves similar performance both in micro benchmarks (fork, IpC, library update, etc.) and in applications (Redis, web server, and DNS server) compared to conventional processes, while retaining the strong isolation benefit of VMs/Unikernels.

CCS Concepts: • **Software and its engineering → Multiprocessing / multiprogramming / multitasking**; • **Security and privacy →** *Virtualization and security*;

Additional Key Words and Phrases: Virtualization architecture, virtual appliances, Unikernel, process-like VM (pVM)

ACM Reference format:
Yiming Zhang, Chengfei Zhang, Yaozheng Wang, Kai Yu, Guangtao Xue, and Jon Crowcroft. 2020. KylinX: Simplified Virtualization Architecture for Specialized Virtual Appliances with Strong Isolation. *ACM Trans. Comput. Syst.* 37, 1-4, Article 2 (February 2021), 27 pages.
https://doi.org/10.1145/3436512

This work is supported by the National Key Research and Development Program of China (2018YFB2101102), the National Natural Science Foundation of China (61772541, 61872376, 61932001, U1736207, 62072306), and the Program of Shanghai Academic Research Leader (20XD1402100). Yiming Zhang designed KylinX when he was visiting University of Cambridge.
Authors' addresses: Y. Zhang (corresponding author), C. Zhang, Y. Wang, and K. Yu, NiceX Lab, National University of Defense Technology, Changsha, China; emails: {zhangyiming, zhangchengfei, wangyaozheng, yukai}@nicexlab.com; G. Xue, Shanghai Jiao Tong University, Shanghai, China; email: xue-gt@cs.sjtu.edu.cn; J. Crowcroft, Computer Laboratory, University of Cambridge, Cambridge, United Kingdom; email: Jon.Crowcroft@cl.cam.ac.uk.

0734-2071/2020/02-ART2 $15.00
https://doi.org/10.1145/3436512

1 INTRODUCTION

Commodity clouds (like EC2 [4]) provide a public platform where tenants rent virtual machines (VMs) to run their applications. These cloud-based VMs are usually dedicated to specific online applications such as big data analysis [32] and game servers [28] and are referred to as (virtual) appliances [71, 81]. The highly specialized, single-purpose appliances need only a very small portion of traditional operating system (OS) support to run their accommodated applications, while the current general-purpose OSs contain extensive libraries and features for multi-user, multi-application scenarios. The mismatch between the single-purpose usage of appliances and the general-purpose design of traditional OSs induces performance and security penalty, making appliance-based services cumbersome to deploy and schedule [67, 79], inefficient to run [71], and vulnerable to bugs of unnecessary libraries [36].

This problem has recently motivated the design of Unikernel [71], a library operating system (LibOS) architecture that is targeted for efficient and secure appliances in the clouds. Unikernel refactors a traditional operating system into libraries and seals the application binary and requisite libraries into a specialized appliance image that could run directly on a hypervisor such as Xen [39] and KVM [30]. Compared to other isolation techniques (such as Tiny Core Linux [25], OS^v [63], FireCracker [8], gVisor [10], and Kata Containers [12]), Unikernel appliances eliminate unused code and achieve smaller memory footprint, shorter boot times, and lower overhead while guaranteeing the same level of isolation and even higher security [71]. The hypervisor's steady interface avoids hardware compatibility problems encountered by early LibOSs [51].

On the downside, however, Unikernel strips off the *process* abstraction from its statically sealed monolithic appliances and thus sacrifices flexibility. For example, Unikernel cannot support dynamic fork, a basis for commonly used multi-process abstraction of conventional UNIX applications; and the compile-time determined immutability precludes runtime management such as online library update and address space randomization. This inability has largely reduced the applicability of Unikernel.

In this article, we examine whether there is a balance embracing the best of both Unikernel appliances (strong isolation) and processes (high flexibility/efficiency). We draw an analogy between appliances on a hypervisor and processes on a traditional OS and take one step forward from static Unikernels to present KylinX, a dynamic library operating system for simplified and efficient cloud virtualization by providing the pVM (process-like VM) abstraction. We take the hypervisor as an OS and the appliance as a process allowing both page-level and library-level dynamic mapping for pVM.

At the page level, KylinX supports pVM fork plus a set of API for inter-pVM communication (IpC), which is compatible with conventional UNIX inter-process communication (IPC). The security of IpC is guaranteed by only allowing IpC between a *family* of mutually trusted pVMs forked from the same root pVM.

At the library level, KylinX supports shared libraries to be dynamically linked to a Unikernel appliance, enabling pVMs to perform (i) online library update, which replaces old libraries with new ones at runtime; and (ii) recycling, which reuses in-memory domains for fast booting. We analyze potential threats induced by dynamic mapping and enforce corresponding security restrictions.

We have implemented a prototype of KylinX based on Xen [39] (a type-1 hypervisor) by modifying MiniOS [19] (a Unikernel LibOS written in C) and Xen's toolstack. KylinX can fork a pVM in about 1.3 ms and link a library to a running pVM in a few ms, both comparable to process fork on Linux (about 1 ms). Latencies of KylinX IpCs are also comparable to that of UNIX IPCs. Evaluation on real-world applications (including a Redis server [18] and a web server [16]) shows that KylinX achieves higher applicability and performance than static Unikernels while retaining the isolation guarantees.

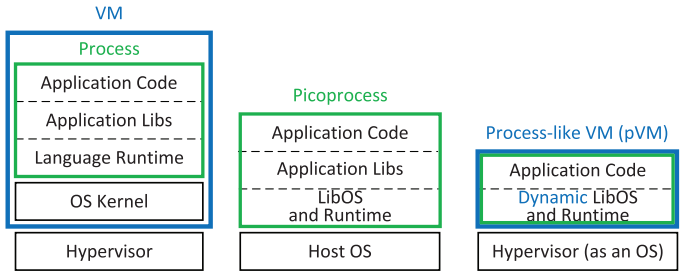

Fig. 1. Alternative virtualization architectures.

The rest of this article is organized as follows: Section 2 introduces the background and design options. Section 3 presents the simplified virtualization architecture of KylinX LibOS with security restrictions. Section 4 reports the evaluation results of the KylinX prototype. Section 5 introduces related work. Section 6 concludes the article and discusses future work.

2 PRELIMINARIES

This section first reviews the concept of VMs, picoprocess [50], and Unikernel [71], and then discusses the motivation and design choices of KylinX.

2.1 VMs, Containers & Picoprocesses

There are several conventional models in the literature of virtualization and isolation, mainly including processes, Jails, and VMs.

- OS processes. The process model is targeted for a conventional (partially trusted) OS environment and provides rich ABI (application binary interface) and interactivity that make it not suitable for truly adversarial tenants.
- FreeBSD Jails [60]. The jail model provides a lightweight mechanism to separate applications and their associated policies. It runs a process on a conventional OS, but restricts several of the syscall interfaces to reduce vulnerability.
- VMs. The VM model builds an isolation boundary matching hardware. It provides legacy compatibility for guests to run a complete OS, but it is costly due to duplicated and vestigial OS components.

VMs (shown in Figure 1 (*left*)) have been widely used in multi-tenant clouds, since they guarantee strong (type-1-hypervisor) isolation [70]. However, the current virtualization architecture of VMs is heavy with layers mainly including hypervisor, VM, OS kernel, process, language runtime (such as glibc [21] and JVM [29]), libraries, and application, which are complex and could no longer satisfy the efficiency requirements (in, e.g., resource footprint and boot times) of commercial clouds.

Containers (such as LXC [14] and Docker [20]) leverage kernel features to package processes. They are recently in great demand [5, 6, 33], because they are lightweight compared to VMs. However, containers offer little isolation, and thus they often run with user-space kernels (provided by, e.g., gVisor [10]) to achieve proper security guarantees [74].

To address containers' inability of isolation, picoprocesses [50] are proposed as enhanced containers (as shown in Figure 1 (*center*)) with stronger isolation and lighter-weight host obligations. They use a small interface between the host OSs and the guests to implement a LibOS, which realizes the host ABI and maps the high-level guest API onto the small interface. Picoprocesses are particularly suitable for client software delivery, because client software needs to run on various

combinations of host hardware platforms and OSs [50]. They could also run on top of hypervisors [41, 79].

Recent studies [41, 69, 85] on picoprocesses relax the original static isolation model by allowing dynamics. For example, Drawbridge [79] refactors Windows 7 to create a self-contained LibOS running in a picoprocess supporting desktop applications. Graphene [85] supports picoprocess fork and multi-picoprocess API. Bascule [41] allows OS-independent extensions to be attached safely and efficiently to a picoprocess at runtime. Tardigrade [69] constructs fault-tolerant services in Bascule picoprocesses. Although these relaxations dilute the strict isolation model, they effectively extend the applicability of picoprocesses to a much broader range of applications.

2.2 Unikernel Appliances

Process-based virtualization and isolation techniques (like containers and picoprocesses) face challenges from the broad kernel syscall API that is used to interact with the host OS for, e.g., process/thread management, IPC, networking. The number of Linux syscalls has reached almost 400 [3] and is continuously increasing, and the syscall API is much more difficult to secure than the ABI of VMs (which could leverage hardware memory isolation and CPU rings) [74].

Recently, researchers and cloud vendors propose to reduce VMs, instead of augmenting processes, to achieve secure and efficient cloud virtualization [8, 12, 25, 63, 71]. Unikernel [71] is focused on single-application VM appliances [35] and adapts the Exokernel [51] style LibOS to VM guests to enjoy performance benefits while preserving the strong isolation guarantees of a type-1 hypervisor. As shown in Figure 1 (*left*), Unikernel breaks the traditional general-purpose virtualization architecture and implements the conventional OS features (e.g., device drivers and networking) as libraries. Compared to other hypervisor-based reduced VMs (like Tiny Core Linux [25] and OS^{v} [63]), Unikernel seals only the application and its requisite libraries into the image.

Since the hypervisor already provides a number of management features (such as isolation and scheduling) of traditional OSs, Unikernel adopts the *minimalism* philosophy [45], which minimizes the VMs by not only removing unnecessary libraries in compilation but also stripping off the duplicated management features from its LibOS. For example, Mirage [72] follows the multikernel model [40] of running one VM per vCPU and leverages the hypervisor for multicore scheduling, so the single-threaded runtime could have fast sequential performance; MiniOS [19] relies on the hypervisor (instead of an in-LibOS linker) to load/link the appliance at boot time; and LightVM [74] achieves fast VM boot by redesigning Xen's control plane.

2.3 Motivation & Design Choices

Conceptually, Unikernel appliances and conventional UNIX processes are analogous: They both abstract the unit of isolation, privileges, and execution states and provide management functionalities such as memory mapping, execution cooperation, and scheduling. To achieve low memory footprint and small trusted computing base (TCB), Unikernel strips off the process abstraction from its monolithic appliance and links a minimalistic LibOS against its target application, demonstrating the benefit of relying on the hypervisor to eliminate duplicated features. But on the downside, the lack of process abstraction and compile-time determined monolithicity largely reduce the flexibility, efficiency, and applicability of Unikernel.

To address this problem, in this article, we propose KylinX, which provides the process-like VM (`pVM`) abstraction by explicitly taking the hypervisor as an OS and the Unikernel appliance as a process, as shown in Figure 1 (*right*). Compared to picoprocesses (Figure 1 (*center*)), which enhance processes/containers to achieve VM-like isolation, KylinX aims to improve VMs/Unikernels to achieve process-like flexibility and efficiency. Most effort of picoprocesses is to realize isolation

Table 1. Inspired by Dynamic Picoprocesses, KylinX Explores New Design Space and Extends the Applicability of Unikernel

	Static	Dynamic
Picoprocess	Embassies [56], Xax [50], etc.	Graphene [85], Bascule [41], etc.
Unikernel	Mirage [72], MiniOS [19], etc.	**KylinX**

(by, e.g., exploiting the MMU virtualization support [85]), which is easily inherited by pVMs from the hypervisor-backed virtual machines. KylinX allows both page-level and library-level dynamic mapping, so pVMs could embrace the best of both Unikernel appliances and UNIX processes. As shown in Table 1, KylinX could be viewed as an extension (providing the pVM abstraction) to Unikernel, similar to the extension of Graphene [85] (providing conventional multi-process compatibility) and Bascule [41] (providing runtime extensibility) to picoprocess.

We implement KylinX's dynamic mapping extension in the hypervisor instead of the guest LibOS for the following reasons: First, an extension outside the guest LibOS allows the hypervisor to enforce mapping restrictions and thus improves security. Second, the hypervisor is more flexible to realize dynamic management for, e.g., restoring live states during pVM's online library update. And third, it is natural for KylinX to follow Unikernel's minimalism philosophy (discussed in Section 2.2) of leveraging the hypervisor to eliminate duplicated guest LibOS features.

The state-of-the-art Unikernels seal monolithic and minimum appliances against modification, stripping away any unused functionality at compile-time [36, 71, 78]. In contrast, KylinX slightly relaxes Unikernels' strict requirements to support flexible management, with proper restrictions on dynamic mapping of pages (Section 3.2.3) and libraries (Section 3.3.4). KylinX's dynamic page/library mapping mechanism is necessary for flexible pVM management, and any unused libraries will not be (statically or dynamically) linked to pVMs. Considering the sizes of normal Linux VMs (on the order of hundreds of MBs to several GBs), the slightly increased appliance image sizes (at the scale of a few hundred KBs) compared to static Unikernels are negligible. Therefore, it is an appropriate tradeoff between monolithicity/minimality and flexibility/functionality for most applications running inside Unikernel appliances.

Backward compatibility is another tradeoff. The original Mirage Unikernel [71] takes an extreme position where existing applications and libraries have to be completely rewritten in OCaml [15] for type safety, which requires a great deal of engineering effort and may introduce new vulnerabilities and bugs. In contrast, KylinX aims to support source code (mainly C) compatibility, so a large variety of legacy applications could run on KylinX with moderate effort for adaptation.

Threat model. KylinX assumes a traditional threat model [63, 71], the same context as Unikernel [71] where VMs/pVMs run on the hypervisor and are expected to provide network-facing services in a public multi-tenant cloud. We assume the adversary can run untrusted code in the VMs/pVMs, and applications running in the VMs/pVMs are under potential threats both from other tenants in the same cloud and from malicious hosts connected via Internet. KylinX treats both the hypervisor (with its toolstacks) and the control domain (dom0) as part of the TCB and leverages the hypervisor for isolation against attacks from other tenants. The use of secure protocols such as SSL and SSH helps KylinX pVMs trust external entities.

Recent advance in hardware like Intel Software Guard eXtensions (SGX) [17] demonstrates the feasibility of shielded execution in *enclaves* to protect VMs/pVMs from threats of the privileged hypervisor and dom0 [37, 42, 58], which will be studied in our future work. We also assume hardware devices are not compromised, although in rare cases hardware threats have been identified [43].

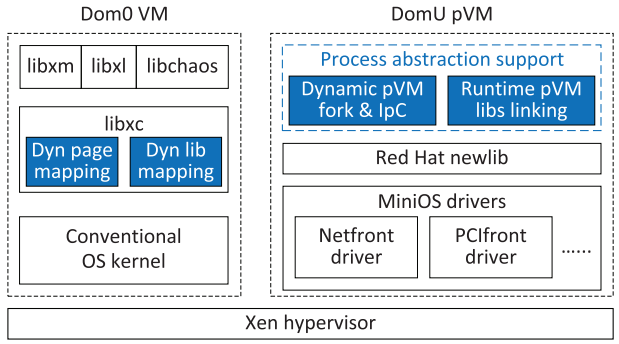

Fig. 2. KylinX components. Xen comprises a privileged control VM (domain) called dom0 and a set of user (or guest) domains called domU. Here, the DomU pVM is essentially a Unikernel appliance.

3 KYLINX DESIGN

3.1 Overview

KylinX extends Unikernel to realize desirable features and optimizations that are previously applicable only to processes. Instead of designing a new LibOS from scratch, we base KylinX on MiniOS [36], a C-style Unikernel LibOS for user VM domains (domU) running on the Xen hypervisor. MiniOS uses its front-end drivers to access hardware, which connect to the corresponding back-end drivers in the privileged dom0 or a dedicated driver domain. MiniOS has a single address space without kernel and user space separation, as well as a simple scheduler without preemption. MiniOS is tiny but fits the bill allowing a neat and efficient LibOS design on Xen. For example, Erlang on Xen [1], LuaJIT [2], ClickOS [75], and LightVM [74] leverage MiniOS to provide Erlang, Lua, Click, and fast boot environments, respectively.

As shown in Figure 2, the MiniOS-based KylinX design consists of (i) the (restricted) dynamic page/library mapping extensions of Xen's toolstack in Dom0 and (ii) the process abstraction support (including dynamic pVM fork/IpC and runtime pVM library linking) in DomU.

3.2 Dynamic Page Mapping

KylinX supports process-style appliance fork and communication by leveraging Xen's shared memory and grant tables to perform cross-domain page mapping.

3.2.1 pVM *Fork.* The `fork` API is the basis for realizing traditional multi-process abstractions for pVMs. KylinX treats each user domain (pVM) as a process, and when the application invokes `fork()` a new pVM will be generated. We leverage the highly efficient memory sharing mechanism of Xen to implement the `fork()` operation, which creates a *child* pVM by using the `xc_dom_image` structure to duplicate the memory allocation of the parent pVM and invoking Xen's `unpause()` API to "resume" the execution of the replica of the calling parent pVM.

The control domain (dom0) is responsible for forking and starting the child pVM. We modify *libxc* to keep the `xc_dom_image` structure (which consists of shared page tables, physical-to-machine mapping, device tree, `start_info` page, etc.) in memory when the parent pVM was created. When `fork()` is invoked in the parent pVM, as shown in Figure 3, the parent pVM uses `setjmp()` to get and store the current states of CPU registers. Then, Dom0 copies the `xc_dom_image` structure for the child pVM and configures new grant tables for the child pVM to share the sections, the stack, as well as the stored CPU register states between the parent and child pVMs. Writable pages are granted in a copy-on-write (CoW) manner.

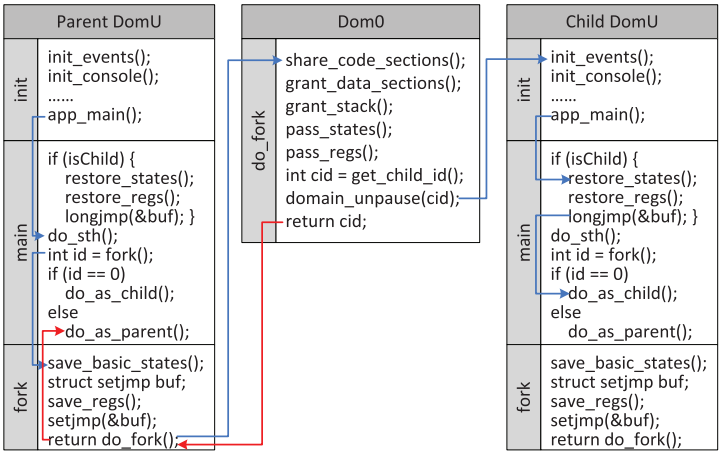

Fig. 3. pVM fork. After *fork* is invoked, KylinX creates a child pVM by sharing the parent (caller) pVM's pages.

After preparing the memory layout and CPU register states (inherited from the parent), the child pVM is started by calling `domain_unpause()` of *libxc*, which resumes the execution of a domain via the Xen hypercall `domctl` (with op code `XEN_DOMCTL_unpausedomain`). The child pVM (i) accepts the shared pages from its parent, (ii) restores the CPU registers and jumps to the next instruction after fork (via `longjmp()`), and (iii) begins to run as a child. After `fork()` is completed, KylinX asynchronously initializes an event channel and shares dedicated pages between the parent and child pVMs to enable their IpC, as introduced in the next subsection.

File descriptors. KylinX preserves the semantics of open file descriptors in UNIX, where a forked pVM can be viewed as a forked process. If the parent pVM has open file descriptors, it will share the descriptors with the child pVM using shared memory. The file seek cursors are also copied to the child, but (for simplicity) currently the parent and child pVMs do not share the movements of seek cursors, which will be studied in our future work. Similar to conventional UNIX processes, a child pVM can read/write open files (such as sockets and disk files) with appropriate permissions granted by the parent pVM.

Eliminating XenStore. As discussed in Reference [74], XenStore is highly inefficient in maintaining metadata information for Xen. Most recently, LightVM [74] replaces XenStore and Xen's *xl*/*libxl* with the lightweight `noxs` (No-XenStore) and *chaos*/*libchaos*. Although significantly improving Xen's metadata maintenance and achieving fast booting, the downside of LightVM is that it introduces a complete redesign of Xen's control plane as well as a re-implementation of the toolstack, making it hard to be accepted by the Xen project. Since the metadata maintenance is orthogonal to our design, KylinX could support not only Xen's original XenStore but also LightVM's `noxs`. As illustrated in Section 4.3, by adopting `noxs`, KylinX pVMs achieve millisecond-level boot times even for a large number of pVMs.

*3.2.2 Inter-*pVM *Communication (IpC).* KylinX provides a multi-process (multi-pVM) application with the view that all of its processes (pVMs) are collaboratively running on the OS (hypervisor). Currently KylinX follows the strict isolation model [85] where only mutually trusted pVMs can communicate with each other, which will be discussed in more detail in Section 3.2.3.

KylinX leverages the event channel and shared pages of Xen to realize inter-pVM communication. If two mutually trusted pVMs have not yet initialized an event channel when they communicate for the first time because they have no parent-child relationship via `fork()` (Section 3.2.1), then

Table 2. Inter-pVM Communication API

Type	API	Description
Pipe	pipe	Create a pipe and return the *fds*.
	write	Write *value* to a pipe.
	read	Read *value* from a pipe.
Signal	kill	Send signal to a domain.
	signal	set handler of a signal.
	exit	Child sends SIGCHLD to parent.
	wait	Parent waits for child's signal.
Message Queue	ftok	Return the key for a given *path*.
	msgget	Create a message queue for *key*.
	msgsnd	Write *msg* to message queue.
	msgrcv	Read *msg* from message queue.
Shared Memory	shmget	Create & share a memory region.
	shmat	Attach shared memory (of *shmid*).
	shmdt	Detach shared memory.

KylinX will (i) verify their mutual trustworthiness (Section 3.2.3), (ii) initialize an event channel, and (iii) share dedicated pages between them.

Similar to GNU Hurd [27] and Graphene [86], which use a helper thread for communication between parent and child processes/picoprocesses, KylinX also runs a per-pVM specific helper thread that handles all IpC for its pVM and is transparent to the accommodated application and directly interacts with the underlying Xen substrate by leveraging event channels to notify events and using shared pages to realize the communication. Any kinds of communications can be abstracted as transferring bytes and controls between pVMs. The helper thread of one pVM writes the IpC code as well as the arguments to the shared pages and notifies its counterpart via their event channel. Then the helper thread of the other pVM will receive the event, read the code and arguments from the shared pages, and call the corresponding processing function.

As listed in Table 2, currently KylinX has already realized the following four types of inter-pVM communication APIs:

- `pipe(fd)` creates a pipe and returns two file descriptors (`fd[0]` and `fd[1]`), one for `write` and the other for `read`.
- `kill(domid, SIG)` sends a signal (`SIG`) to another pVM (`domid`) by writing `SIG` to the shared page and notifying the target pVM (`domid`) to read the signal from that page; `signal(SIG, handler)` sets the disposition of the signal `SIG` to `handler`, which will be called after reading the signal (when notified via Xen's event channel); `exit` and `wait` are implemented using `kill`.
- `ftok(path, projid)` translates the `path` and `projid` to an IpC key, which will be used by `msgget(key, msgflg)` to create a message queue with the flag (`msgflg`) and return the queue ID (`msgid`); `msgsend(msgid, msg, len)` and `msgrcv(msgid, msg, len)` write/read the queue (`msgid`) to/from the `msgbuf` structure (`msg`) with length `len`, respectively.
- `shmget(key, size, shmflg)` creates and shares a memory region with the key (`key`), memory size (`size`), and flag (`shmflg`), and returns the shared memory region ID (`shmid`), which could be attached and detached by `shmat(shmid, shmaddr, shmflg)` and `shmdt(shmaddr)`.

ALGORITHM 1: Dynamic Library Mapping in *libxc* (`xc_dom_map_dyn()`)

```
Input: dynlib_header: the program header of the dynamic library to be mapped
Output: 0: success, -1: failure
Read the information of all dynamic libraries from dynlib_header;
for each dynamic library do
    if signature check fails or version check fails or loader check fails then
        return -1;
    end
    Retrieve info of dynamic sections;
    Load dynamic sections;
    Relocate unresolved symbols (lazily);
end
Jump to the program entry;
return 0;
```

3.2.3 Dynamic Page Mapping Restrictions. This subsection discusses security issues of KylinX's dynamic page mapping.

When performing dynamic pVM fork, the parent pVM shares its pages with an empty child pVM, the procedure of which introduces no new threats.

When performing IpC, KylinX guarantees the security by the abstraction of a *family* of mutually trusted pVMs, which are forked from the same root pVM. For example, if a pVM *A* forks a pVM *B*, which further forks another pVM *C*, then the three pVMs *A*, *B*, and *C* belong to the same family.

For simplicity, currently KylinX follows the all-all-nothing isolation model: Only the pVMs belonging to the same family are considered to be trusted and are allowed to communicate with each other. KylinX rejects communication requests between untrusted pVMs.

3.3 Dynamic Library Mapping

3.3.1 pVM *Library Linking.* Inherited from MiniOS, KylinX has a single flat virtual memory address space where application binary and libraries, system libraries (for bootstrap, memory allocation, etc.), and data structures co-locate to run. KylinX adds a *dynamic segment* into the original memory layout of MiniOS to accommodate dynamic libraries after they are loaded.

As depicted in Figure 2, we implement the dynamic library mapping mechanism in the Xen control library (*libxc*), which is used by the upper-layer toolstacks such as *xm*/*xl*/*chaos*. A pVM is actually a para-virtualized domU, which in turn (i) creates a domain, (ii) parses the kernel image file, (iii) initializes the boot memory, (iv) builds the image in the memory, and (v) boots up the image for domU. In the above 4th step, we add a function, `xc_dom_map_dyn()` (Algorithm 1), to map the shared libraries into the *dynamic segment* by extending the static linking procedure of *libxc* as follows:

- First, KylinX reads the addresses, offsets, file sizes, and memory sizes of the shared libraries from the program header table of the appliance image.
- Second, it verifies whether the restrictions (Section 3.3.4) are satisfied. If not, the procedure terminates.
- Third, for each dynamic library, KylinX retrieves the information of its dynamic sections, including the dynamic string table, symbol table, and so on.
- Fourth, KylinX maps all the requisite libraries throughout the dependency tree into the dynamic segment of the pVM, which will *lazily* relocate an unresolved symbol to the proper virtual address when it is actually accessed.
- Finally, it jumps to the pVM's entry point.

KylinX will not load/link the shared libraries until they are actually used, which is similar to lazy binding [23] for conventional processes. Therefore, the boot times of KylinX pVMs are lower than that of previous Unikernel VMs. Further, compared to previous Unikernels that support only static libraries, another advantage of KylinX using shared libraries is that it effectively reduces the memory footprint in high-density deployment (e.g., 8K VMs per machine in LightVM [74] and 80K containers per machine in Flurries [89]), which is the single biggest factor [74] limiting both scalability and performance.

Next, we will discuss two simple applications of dynamic library mapping of KylinX pVMs.

3.3.2 Online pVM *Library Update.* It is important to keep the system/application libraries up to date to fix bugs and vulnerabilities. Static Unikernel [71] has to recompile and reboot the entire appliance image to apply updates for each of its libraries, which may result in significant deployment burdens when the appliance has many third-party libraries.

Online library update is more attractive than rolling reboots mainly in *keeping connections* to the clients. First, when the server has many long-lived connections, rebooting will result in high reconnection overhead. Second, it is uncertain whether a third-party client will re-establish the connections or not, which imposes complicated design logic for reconnection after rebooting. Third, frequent rebooting and reconnection may severely degrade the performance of critical applications such as high-frequency trading.

Dynamic mapping makes it possible for KylinX to realize online library update. However, libraries may have their own states for, e.g., compression or cryptography, therefore simply replacing stateless functions cannot satisfy KylinX's requirement.

Like most library update mechanisms (including DYMOS [65], Ksplice [38], Ginseng [77], PoLUS [48], Katana [80], Kitsune [54], etc.), KylinX requests the new and old libraries to be binary-compatible: It is allowed to add new functions and variables to the library, but it is not allowed to change the interface of functions, remove functions and variables, or change fields of structures. For library states, we expect that all the states are stored as variables (or dynamically allocated structures) that could be saved and restored during the update procedure.

KylinX provides the `update(domid, new_lib, old_lib)` API to dynamically replace `old_lib` with `new_lib` for a domU pVM (ID = `domid`) with necessary update of library states. We also provide an *update* command "`update domid, new_lib, old_lib`" for parsing parameters and calling the `update()` API, by which `old_lib` could be updated with `new_lib`.

The difficulty of dynamic pVM update lies in manipulating symbol tables in a sealed VM appliance. We leverage dom0 to address this problem. When the `update` API is called, KylinX will perform the following operations:

- First, map the new library into dom0's virtual address space.
- Second, share the loaded library with domU.
- Third, verify whether the old library is quiescent by asking domU to check the call stack of each kernel thread of domU.
- Fourth, wait until the old library is not in use and pause the execution.
- Fifth, modify the entries of affected symbols to the proper addresses.
- And, finally, release the old library.

Similar to the existing online library update mechanisms for processes [38, 48, 54, 65, 77, 80], DomU checks if the old library is quiescent by examining all call stacks to verify its noninvolvement. If a thread calling functions of the old library is blocked, then KylinX cannot update the library until the call is done.

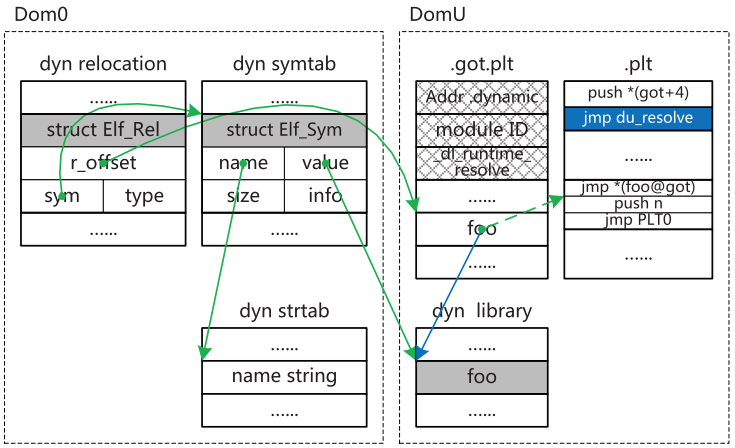

Fig. 4. KylinX dynamic symbol resolution for functions. The green lines represent pointers for normal lazy binding of processes. The blue line represents the result of KylinX's resolution, pointing to the real function instead of the .plt table entry (dashed green line).

In the above 5th step, there are two kinds of symbols (*functions* and *variables*), which will be resolved as discussed below.

Functions. The dynamic resolution procedure for functions is illustrated in Figure 4. We keep the relocation table, symbol table, and string table in dom0, as they are not in the loadable segments. We load the global offset table of functions (.got.plt) and the procedure linkage table (.plt) in dom0 and share them with domU. To resolve symbols across different domains, we modify the second line of assembly in the first entry of the .plt table (as illustrated in the blue region in Figure 4) to point to KylinX's symbol resolve function (`du_resolve`).

After the new library (`new_lib`) is loaded, the entry of each function of `old_lib` in the .got.plt table (e.g., `foo` in Figure 4) is modified to point to the corresponding entry in the .plt table, i.e., the second assembly (`push n`) illustrated by the dashed green line in Figure 4. When a function (`foo`) of the library is called for the first time after `new_lib` is loaded, the resolution function `du_resolve` will be called with two parameters (`n` and `*(got+4)`), where `n` is the offset of the symbol (`foo`) in the .got.plt table, and `*(got+4)` is the ID of the current module.

`du_resolve` then asks dom0 to call its counterpart `d0_resolve`, which finds `foo` in `new_lib` and updates the corresponding entry (located by `n`) in the .got.plt table of the current module (ID = `module_ID`) to the proper address of `foo` (the blue line in Figure 4).

Variables. Dynamic resolution for variables is slightly complex. Currently, we simply assume that `new_lib` expects all its variables to be set to their live states in `old_lib` instead of their initial values. Without this restriction, the compiler will have to rely on extensions to allow developers to specify their intention for each variable, which would significantly complicate the development of applications.

(1) Global variables. If a global variable (`g`) of the library is accessed in the main program, then `g` is stored in the data segment (.bss) of the program and there is an entry in the global offset table (.got) of the library pointing to `g`, so after `new_lib` is loaded KylinX will resolve `g`'s entry in the .got table of `new_lib` to the proper address of `g`. Otherwise, `g` is stored in the data segment of the library and so KylinX is responsible for copying the global variable `g` from `old_lib` to `new_lib`.

(2) Static variables. Since static variables are stored in the data segment of the library and cannot be accessed from outside, after `new_lib` is loaded KylinX will simply copy them one by one from `old_lib` to `new_lib`.

(3) Pointers. If a library pointer (p) points to a dynamically allocated structure, then KylinX preserves the structure and set p in new_lib to it. If p points to a global variable stored in the data segment of the program, then p will be copied from old_lib to new_lib. If p points to a static variable (or a global variable stored in the library), then p will point to the new address.

3.3.3 pVM Recycling. The standard boot (Section 3.3.1) of KylinX pVMs and Unikernel VMs [74] is relatively slow. As evaluated in Section 4.2, it takes 100+ milliseconds to boot up a pVM or a Unikernel VM, most time of which is spent in creating the empty domain. Therefore, we design a pVM recycling mechanism for KylinX pVMs, which leverages dynamic library mapping to bypass domain creation.

The basic idea of recycling is to reuse an in-memory empty domain to dynamically map the application (as a shared library) to that domain. Specifically, an empty recyclable domain is checkpointed before calling the (dummy) entry function (app_entry) of a placeholder library, waiting for running the target application. The application is compiled into a shared library of which the app_entry function contains the real application code. To eliminate the overhead of domain initiation, KylinX can restore the checkpointed domain and follow the online update procedure (Section 3.3.2) to "update" the domain's placeholder library with the shared application library.

3.3.4 Dynamic Library Mapping Restrictions. KylinX should be able to isolate any new vulnerabilities compared to the statically and monolithically sealed Unikernel when performing dynamic library mapping. The main threat is that the adversary may load a malicious library into the pVM's address space, replace a library with a compromised one that has the same name and symbols, or modify the entries in the symbol table of a shared library to the fake symbols/functions.

To address these threats, KylinX enforces restrictions on the identities of libraries as well as the loaders of the libraries. KylinX supports developers to specify the restrictions on the signature, version, and loader of the dynamic library, which are stored in the header of the pVM image and will be verified before linking a library.

Signature and version. The library developer first generates the library's SHA1 digest that will be encrypted by RSA (Rivest-Shamir-Adleman). The result is saved in a *signature* section of the dynamic library. If the appliance requires signature verification of the library, the signature section will be read and verified by KylinX using the public key. Version restrictions are requested and verified similarly.

Loader. The developer may request different levels of restrictions on the loader of the libraries: (i) only allowing the pVM itself to be the loader; (ii) also allowing other pVMs of the same application; or (iii) even allowing pVMs of other applications. With the first two restrictions a malicious library in one compromised application would not affect others. Another case for loader check is to load the application binary as a library and link it against a pVM for fast recycling (Section 3.3.3), where KylinX restricts the loader to be an empty pVM.

With these restrictions, KylinX introduces no new threats compared to the statically sealed Unikernel. For example, runtime library update (Section 3.3.2) of a pVM with restrictions on the signature (to be the trusted developer), version (to be the specific version number), and loader (to be the pVM itself) will have the same level of security guarantees as recompiling and rebooting.

4 EVALUATION

4.1 Implementation

We implement a prototype of KylinX on top of Ubuntu 16.04 and Xen 4.8. Following the default settings of MiniOS [19], we, respectively, use RedHat Newlib and lwIP as the libc/libm libraries

and TCP/IP stack. We also implement a simple in-memory filesystem for KylinX so the evaluation results are not skewed by disk access speeds. We have ported several applications to KylinX, including a multi-process Redis server [18], a multi-thread web server [16], and a BIND DNS server, which will be evaluated in the next section. Due to the limitation of RedHat Newlib, and lwIP, we perform two kinds of adaptations for porting these applications to KylinX. First, KylinX can support only `select` but not the more efficient `epoll`. Second, inter-process communications (IPC) are limited to the API listed in Table 2.

Coroutine-based multithreading. Besides the *multi-process* support, which is one of the main features of KylinX, another widely used abstraction of conventional applications is *multi-thread*. The original MiniOS executes each user thread on one kernel thread, which is too heavyweight for context switching and memory consumption in pVMs. To address this problem, KylinX realizes its own lightweight threading library by following the "coroutine" mechanism [52, 66], where a kernel thread may accommodate many user threads (in the form of coroutines) running *logically* concurrently with ultralight overhead. Internally, a background event-loop coroutine polls for completion events and if a thread blocks only that thread's corresponding coroutine is put to sleep while other coroutines continue to run. Note that KylinX implements the multi-thread support inside the pVM rather than in the hypervisor/Dom0, because the multiple threads of a process share the same address space and mutually trust each other, requiring no hardware-based isolation.

Syscall limitation. Although realizing the conventional multi-process abstraction, currently KylinX is still limited by the relatively small number of syscalls originally supported by MiniOS, which prevents KylinX from being seamlessly compatible with a large range of traditional Linux applications. Linux has a slowly evolving syscall footprint by now containing roughly 330+ syscalls, among which only about 70 syscalls are originally supported by MiniOS. With the newly added multi-process support (Table 2), KylinX now provides only about 1/4 syscalls of Linux. According to the statistics [84], a solid two-thirds of Linux syscalls are indispensable to a normal Ubuntu distribution with commonly used applications. Since the compatibility layer (like User-Mode-Linux [26] and L^{4}Linux [13]) is orthogonal to the core features (dynamic page/library mapping) of KylinX, the complete implementation of syscall compatibility is out of the scope of this article and will be studied in our future work. Note that implementing more functionalities and syscalls in KylinX will inevitably increase the size of KylinX pVMs, which could be viewed as a tradeoff between functionality and minimality.

Our testbed has two machines, each of which has an Intel 6-core Xeon E5-2640 CPU, 128 GB RAM, and one 1 GbE NIC. We assign 1 core to Dom0 and the remaining 5 cores to multiple VMs/pVMs in a round-robin fashion. Our evaluation answers the following questions:

- What is the impact of dynamic mapping on the boot time (Section 4.2)?
- How does KylinX perform in pVM fork with noxs (Section 4.3)?
- What is the recycling performance of KylinX (Section 4.4)?
- How about KylinX's memory footprint and image size (Section 4.5)?
- How well does KylinX perform in supporting IpC (Section 4.6) and runtime library update (Section 4.7)?
- And what is the performance of KylinX applications of a Redis server, a web server, and a DNS server (Section 4.8)?

All experiments in this section except pVM fork with noxs (Section 4.3) use the unmodified Xen toolstack and XenStore. Each result is an average of 20 runs.

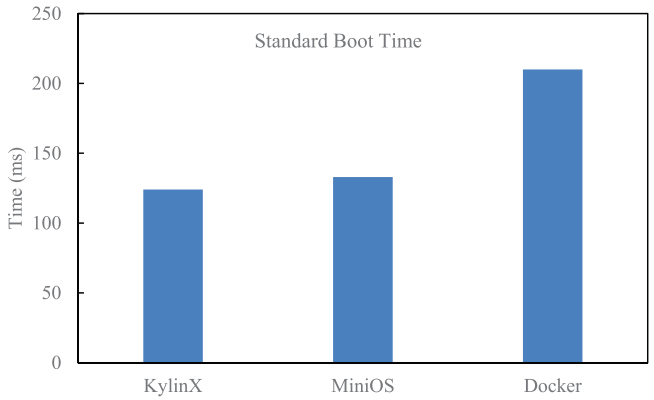

Fig. 5. Boot time of single appliance/container (reduced Redis).

4.2 Standard Boot

We evaluate the time of the standard boot procedure (Section 3.3.1) of KylinX pVMs and compare it with that of MiniOS VMs and Docker containers, all running a Redis server. Redis is an in-memory key-value store that supports fast key-value storage and queries. Each key-value pair consists of a fixed-length key and a variable-length value. It uses a single-threaded process to serve user requests and realizes (periodic) serialization by forking a new backup process.

In this experiment, we have made the following simplifications: First, we disable the logging mechanism in XenStore to eliminate the interference of periodic log file flushes. Second, we compile *libc* (the C library of RedHat Newlib) into a static library and *libm* (the math library of Newlib) into a shared library that will be linked to the KylinX pVM at runtime. This is because RedHat Newlib *libc* has been designed to be static for use in embedded systems and is difficult to be converted into a shared library. The lwIP library is dynamically linked. Third, since MiniOS cannot support `fork`, we (temporarily) remove the corresponding code in this experiment.

The result is shown in Figure 5. It takes about 124 ms to boot up a single KylinX pVM, which could be roughly divided into two stages, namely, creating the domain/image in memory (steps 1–4 in Section 3.3.1), and booting the image (step 5). Dynamic mapping is performed in the first stage, and the dynamic segment is loaded right after the kernel segment. Most of the time (about 121 ms) is spent in the first stage, which invokes hypercalls to interact with the hypervisor. The second stage takes about 3 ms to start the pVM. In contrast, MiniOS takes about 133 ms to boot up a VM, and Docker takes about 210 ms to start a container. KylinX takes less time than MiniOS mainly because its shared libraries are not read/linked during the booting.

We then evaluate the total times of sequentially booting up a large number (up to 1K) of pVMs on one machine. We also evaluate the total boot times of MiniOS VMs and Docker containers for comparison.

The result is depicted in Figure 6. First, KylinX is slightly faster than MiniOS owing to its lazy loading/linking. Second, the boot times of both MiniOS and KylinX increase superlinearly as the number of VMs/pVMs increases while the boot time of Docker containers increases only linearly, mainly because XenStore is highly inefficient when serving a large number of VMs/pVMs [74].

4.3 Fork with Noxs

Compared to containers, KylinX's standard booting cannot scale well for a large number (about 500) of pVMs due to XenStore's inefficiency in metadata maintenance. LightVM [74] redesigns Xen's control plane to achieve ms-level booting times for a large number of VMs. We replace XenStore

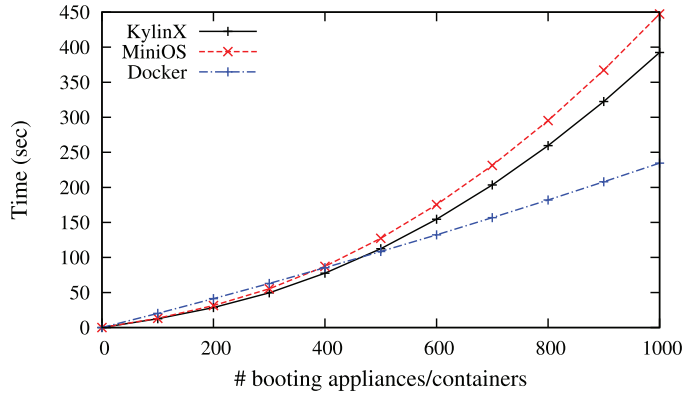

Fig. 6. Total time of standard booting (reduced Redis).

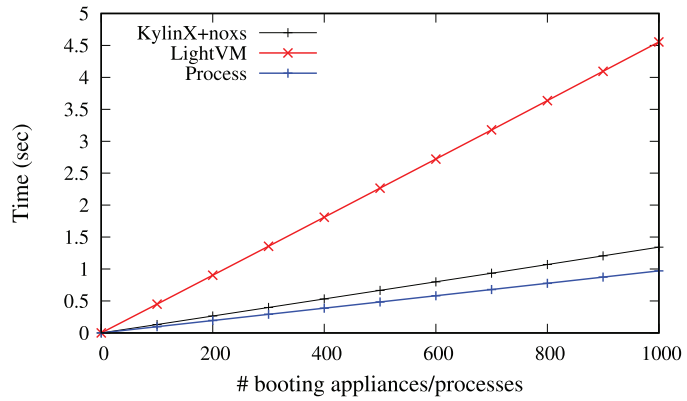

Fig. 7. Time of pVM fork (with noxs), LightVM boot, and process fork.

with LightVM's noxs (along with its toolstack) and test the pVM fork mechanism (Section 3.2.1) running *unmodified* Redis emulating conventional process fork. We also evaluate the times of process fork and LightVM booting for comparison. The fork of a single pVM takes about 1.3 ms, several times faster than LightVM's original boot procedure (about 4.5 ms). KylinX pVM fork is slightly slower than a process fork (about 1 ms) on Ubuntu, because several operations including page sharing and parameter passing are time-consuming. Note that the initialization for the event channel and shared pages of parent/child pVMs is asynchronously performed and thus does not count for the latency of pVM fork.

We also evaluate the total times of sequentially forking and booting a large number (up to 1K) of pVMs, processes, and VMs, respectively, on KylinX, Ubuntu, and LightVM. The result is depicted in Figure 7, where pVM fork is always comparable to process fork and several times faster than LightVM boot. The result proves that noxs enables the boot times of KylinX pVMs to increase linearly even for a large number of pVMs and that KylinX supports micro-reboots [47, 49].

4.4 Recycling

This subsection evaluates the recycling mechanism (Section 3.3.3) of KylinX. We compile a daytime server implementing the daytime protocol [7] into a dynamic library (.so) and checkpoint a recyclable empty pVM with a (placeholder) shared library, which will be replaced with the daytime

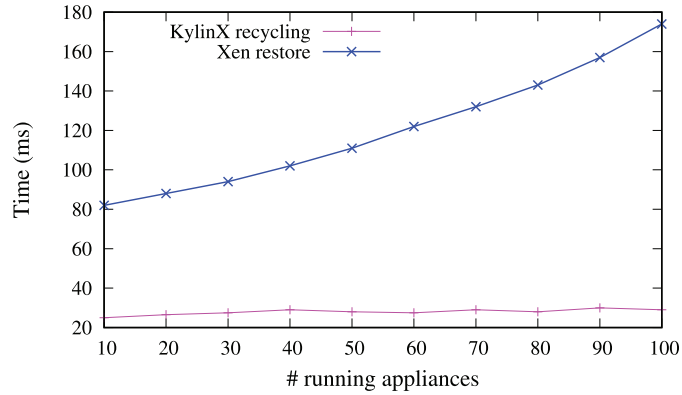

Fig. 8. Time of KylinX recycling.

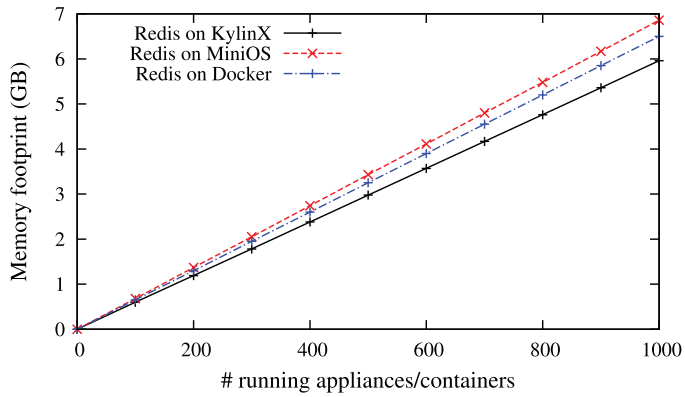

Fig. 9. Memory usage (reduced Redis).

library in the experiment. We use the recycling mechanism to fast boot 10 daytime server pVMs when there are already x (= 10 ~ 100) pVMs running on the machine to test how quick KylinX's recycling is under various workloads. The result is shown in Figure 8, where KylinX can restore and resume the 10 pVMs in less than 30 milliseconds, regardless of the number of running pVMs on the machine. For comparison, we also evaluate the checkpointing/restoring mechanism of Xen for daytime server VMs, the time of which increases as the number of running VMs increases and is significantly higher than that of the recycling mechanism of KylinX.

4.5 Memory Footprint & Image Size

We measure the memory footprint of KylinX, MiniOS, and Docker (Running Redis) for different numbers of pVMs/VMs/containers on one machine. The result (depicted in Figure 9) proves that KylinX pVMs have smaller memory footprint compared to statically sealed MiniOS and Docker containers. This is because KylinX allows the libraries (except *libc*) to be shared by all appliances of the same application (Section 3.3), and thus the shared libraries need to be loaded at most once. The memory footprint advantage facilitates ballooning [55], which could be used to dynamically share physical memory between VM appliances, and enables KylinX to achieve comparable memory efficiency with memory deduplication [53] (discussed in Section 5.1) while introducing much less complexity.

Table 3. Image Sizes (MB)

	KylinX	MiniOS
Bare appliance	1.26	0.94
Redis server (reduced)	1.41	1.07
Web server	1.31	0.98
DNS server	1.49	1.16

Table 4. IpC vs. IPC in Latency (μs)

	pipe	msgqueue	kill	exit/wait	shm
KylinX[1]	55	43	41	43	39
KylinX[2]	240	256	236	247	232
Ubuntu	54	97	68	95	53

KylinX [1]: a pair of lineal pVMs that already have an event channel and shared pages.
KylinX [2]: a pair of non-lineal pVMs.

Small memory footprint of the shared libraries also improves the instruction cache hit ratio, compared to the statically linked libraries of containers/Unikernels, which tend to cause instruction cache bottlenecks and low core utilization when running large numbers of applications on physical machines in warehouse-scale data centers [61].

We also evaluate the image sizes of different KylinX pVMs. As discussed in Section 4.2, KylinX statically links *libc* but dynamically links *libm*. Table 3 lists the image sizes of a bare pVM, a Redis server pVM, and a web server pVM, both for KylinX and for MiniOS. The original Redis server is a multi-process application (Section 4.8.1) that is not supported by MiniOS, so in this experiment, we adapt it to MiniOS by removing the calls to the multi-process API. Table 3 shows that (i) the image sizes of both MiniOS and KylinX are around one MB, much smaller than that of most Linux appliances (on the order of hundreds of MBs to several GBs), and (ii) the image sizes of KylinX pVMs are slightly larger than that of MiniOS appliances, mainly because of the dynamic page/library mapping support (Figure 2) KylinX adds to pVM. Although for statically sealed Unikernels the time needed to load the image grows linearly with its size, it is not the case for dynamically linked pVMs. For instance, KylinX takes less booting time than MiniOS, because its shared libraries are not read/linked during the booting (Section 4.2). Besides, we also expect KylinX to have smaller image sizes than MiniOS when the accommodated applications require more shared libraries.

4.6 Inter-pVM Communication

We evaluate the performance of inter-pVM communication (IpC) of KylinX by forking a parent pVM and measuring the parent/child communication latencies. We refer to a pair of parent/child pVMs as *lineal* pVMs. As introduced in Section 3.2.1, two lineal pVMs already have an event channel and shared pages and thus they could communicate with each other directly. In contrast, non-lineal pVM pairs have to initialize the event channel and shared pages before their first communication. The initialization is verified by a monitor in dom0, which would prohibit the communication if they do not belong to the same family.

The result is listed in Table 4, and we compare it with that of the corresponding IPCs on Ubuntu. KylinX IpC latencies between two lineal pVMs are comparable to the corresponding IPC latencies on Ubuntu, owing to the high-performance event channel and shared memory mechanism of Xen. Note that the latency of `pipe` includes not only creating a pipe but also writing and reading a value

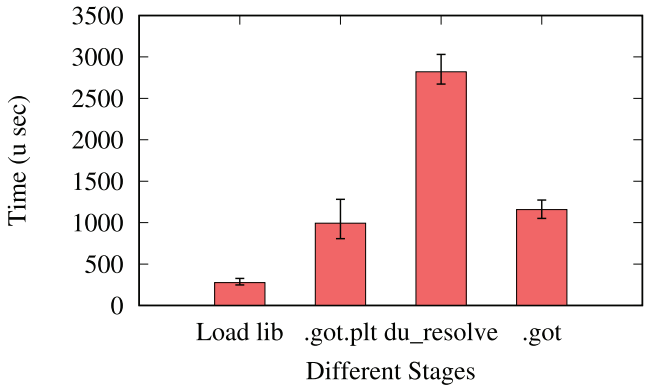

Fig. 10. Runtime library update.

through the pipe. The first-time communication latencies between non-lineal pVMs are several times higher due to the initialization cost.

4.7 Runtime Library Update

We evaluate the runtime library update mechanism of KylinX by dynamically replacing the default *libm* (of RedHat Newlib 1.16) with a newer version (of RedHat Newlib 1.18). *libm* is a math library used by MiniOS/KylinX and contains a collection of 110 basic math functions.

To test KylinX's update procedure for global variables, we also add 111 pseudo global variables as well as one `read_global` function (reading out all the global variables) to both the old and the new *libm* libraries. The `main` function first sets the global variables to random values and then periodically verifies these variables by calling the `read_global` function. Consequently, there are totally 111 functions as well as 111 variables that need to be updated in our test. The update procedure could be roughly divided into four stages and we measure the time of each stage's execution.

First, KylinX loads `new_lib` into the memory of dom0 and shares it with domU. Second, KylinX modifies the relevant entries of the functions in the .got.plt table to point to the corresponding entries in the .plt table. Third, KylinX calls `du_resolve` for each of the functions, which asks dom0 to resolve the given function and returns its address in `new_lib` and then updates the corresponding entries to the returned addresses. Finally, KylinX resolves the corresponding entries of the global variables in the .got table of `new_lib` to the proper addresses. In this experiment, we modify the third stage in our evaluation to update all the 111 functions in *libm* at once, instead of lazily linking a function when it is actually being called (Section 3.3.2), present an overview of the entire runtime update overhead of *libm*.

The result is depicted in Figure 10, where the total overhead for updating all the functions and variables is about 5 milliseconds. The overhead of the third stage (resolving functions) is higher than others including the fourth stage (resolving variables), which is caused by several time-consuming operations in the third stage including resolving symbols, cross-domain invoking `d0_resolve`, returning the real function addresses, and updating the corresponding entries.

4.8 Applications

Besides the process-like flexibility and efficiency of pVM scheduling and management, KylinX also provides high performance for its accommodated applications comparable to that of their

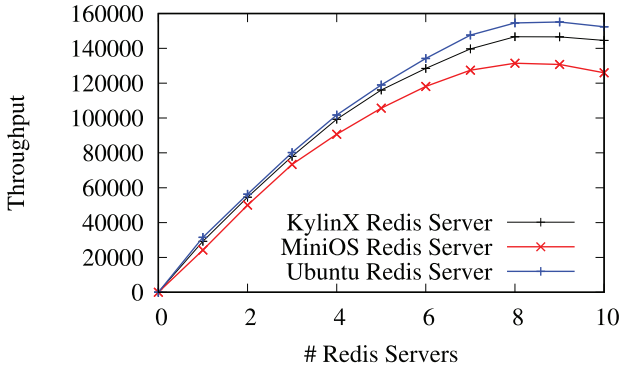

Fig. 11. Redis server application.

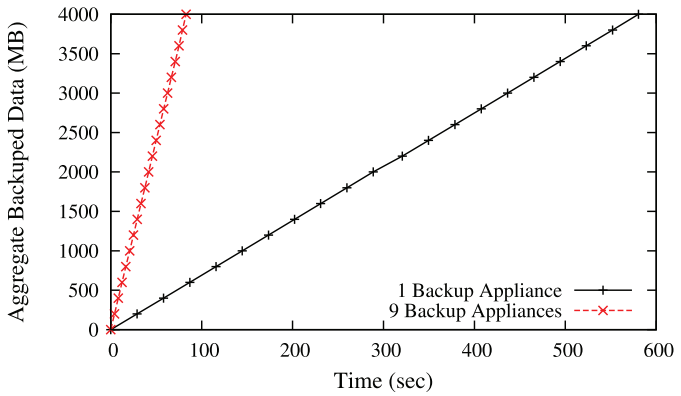

Fig. 12. Serialization by the child pVM after `fork()`.

counterparts on Ubuntu, as evaluated in this subsection. For fairness, Ubuntu also adopts lwIP as the network stack.

4.8.1 Redis Server Application. We evaluate the performance of Redis server in a KylinX pVM and compare it with that in MiniOS/Ubuntu. Again, since MiniOS cannot support `fork()`, we temporarily remove the code for serialization in this experiment. Besides, The Redis server uses `select` instead of `epoll` to realize asynchronous I/O.

We use the Redis benchmark [18] to evaluate the performance, which uses a configurable number of busy loops asynchronously writing KVs. We run different numbers of pVMs/VMs/processes (each for 1 server) servicing write requests from clients. We measure the write throughput (the number of write operations handled per second) as a function of the number of servers. The result is depicted in Figure 11, where the three kinds of Redis servers have similar write throughput (due to the limitation of `select`), increasing almost linearly with the numbers of concurrent servers (scaling being linear up to eight instances before the lwIP stack becomes the bottleneck).

We then switch on the serialization for Redis servers and test the serialization performance of the forked child pVMs. A client first fills the KylinX Redis server with 4 GB of data and then sends a `bgsave` command to the server. The pVM forks multiple children to save the 4GB data to the disk. We test the serialization procedure for different numbers of backup pVMs. The result is shown in Figure 12, where the aggregate backup throughput is about 6.9 MB/sec and 48.8 MB/sec for 1 and 9 pVMs, respectively, demonstrating the forked pVMs are scalable for serialization tasks.

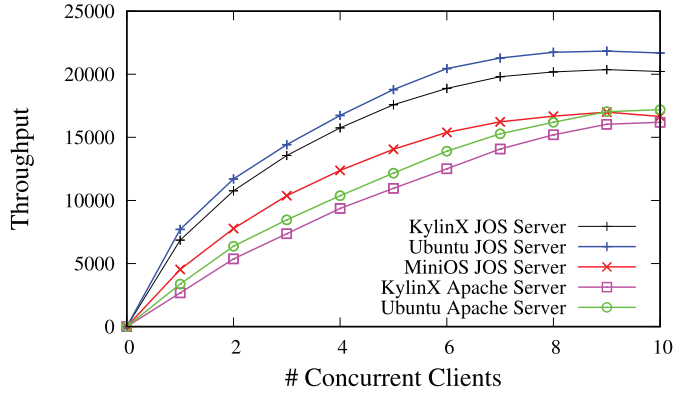

Fig. 13. Web server application.

4.8.2 Web Server Application. We evaluate the JOS web server [16] in KylinX, which adopts multithreading for multiple connections. After the main thread accepts an incoming connection, the web server creates a worker thread to parse the header, reads the file, and sends the contents back to the client. We use the Weighttp benchmark that supports a small fraction of the HTTP protocol (but enough for our web server) to measure the web server performance. Similar to the evaluation of Redis server, we test the web server by running multiple Weighttp [9] clients on one machine, each continuously sending `GET` requests to the web server. We evaluate the throughput (the number of `GET` requests handled per second) as a function of the number of concurrent clients and compare it with the web servers running on MiniOS and Ubuntu, respectively. The result is depicted in Figure 13, where the KylinX web server achieves higher throughput than the MiniOS web server, since its coroutine-based cooperative threading provides higher sequential performance. Both KylinX and MiniOS web servers are slower than the Ubuntu web server, because their connections are inefficiently scheduled with the drivers provided by MiniOS [36].

We also evaluate (a reduced version of) the Apache web server [11] on KylinX and Ubuntu. We set the Apache web server to use the *prefork* MPM (multi-processing module), which pre-forks multiple child processes (with one thread each) in a process pool each handling one connection from a client. Since the rich-featured Apache web server invokes a set of syscalls (like `set_robust_list`, `getdents`, `setgroups`, etc.) that are orthogonal to multi-process and not yet supported by KylinX, we have to (i) largely reduce the code to prevent Apache web server from invoking most of the (dispensable) unsupported syscalls and (ii) partially emulate a few unreducible syscalls (e.g., we use `signal` to partially emulate `rt_sigaction`). The result is also shown in Figure 13, where the throughput of Apache web server on KylinX is slightly lower than that on Ubuntu, because Ubuntu has more efficient library support from its fully optimized glibc. The JOS web servers outperform the Apache web servers by about $1/4 \sim 1/3$, mainly because of the Apache web server's relatively complex security and scheduling mechanisms.

4.8.3 DNS Server Application. BIND [22] implements the DNS (Domain Name System) protocol and resolves DNS queries from clients. We test the BIND DNS server (v9.9) in the KylinX appliance and compare the performance with that on MiniOS and Ubuntu. We use the *queryperf* tool of BIND to evaluate the throughput (number of resolved DNS requests per second) for various zone sizes of 100, 1,000, and 10,000.

The result is shown in Figure 14, where we store the zone in the standard Bind9 format in an in-memory filesystem (to avoid disk I/O induced skewness) with different zone sizes (numbers

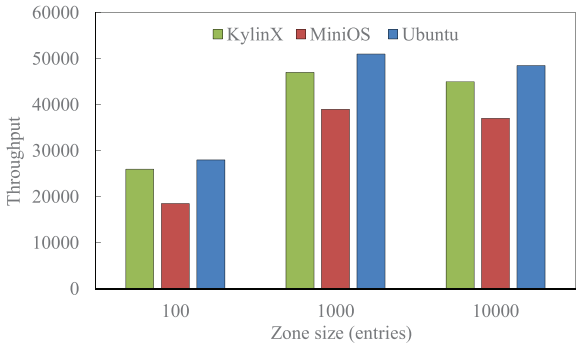

Fig. 14. DNS server application.

of entries). The Ubuntu DNS server achieves about 50,000 queries per second for normal zone sizes (1,000 and 10,000), less than 10% higher than the KylinX DNS server. The KylinX DNS server further outperforms the MiniOS DNS server by more than 20%, mainly owing to KylinX's coroutine-based lightweight threading mechanism. The advantage of the Ubuntu DNS server over the KylinX/MiniOS DNS server is mainly because the C library (Newlib) used by KylinX/MiniOS has lower performance than its counterpart (glibc) used by Ubuntu. Besides, since both MiniOS and KylinX are based on Xen, the interaction between the scheduling of lwIP sockets and the netfront driver also affects the DNS server performance.

5 RELATED WORK

KylinX is related to static Unikernel appliances [36, 71], reduced VMs [25, 62, 63], containers [14, 20, 83], and picoprocess [41, 42, 50, 69, 79, 85].

5.1 Unikernel & Reduced VMs

KylinX is an extension of Unikernel [71] and is implemented on top of MiniOS [36]. Unikernel OSs include Mirage [71], Jitsu [70], Unikraft [24], and so on.

Mirage [71] is targeted for the commodity clouds [90] and compiles the applications and libraries into a single bootable VM image that runs directly on a VM hypervisor (e.g., Xen). Jitsu [70] leverages Mirage [71] to design a power-efficient and responsive platform for hosting cloud services in the edge networks. LightVM [74] leverages Unikernel on Xen to achieve fast booting. MiniOS [36] designs and implements a C-style Unikernel LibOS that runs as a para-virtualized guest OS within a Xen domain. It packages the development tool chain, standard libraries (newlib), and inter-domain communications (IDC) to support preparation of appliance image files running on Xen domains. MiniOS has better backward compatibility than Mirage and supports single-process applications written in C. However, the original MiniOS statically seals an appliance and suffers from similar problems with other static Unikernels.

The difference between KylinX and static Unikernels (such as Mirage [71], MiniOS [36], and EbbRT [82]) lies in the `pVM` abstraction, which explicitly takes the hypervisor as an OS and supports process-style operations such as `pVM` fork/IpC and dynamic library mapping. Mapping restrictions (Section 3.3.4) make KylinX introduce as little vulnerability as possible and have no larger TCB than Mirage/MiniOS [70, 71]. KylinX supports source code (`C`) compatibility instead of using a type-safe language to rewrite the entire software stack [71].

Recent research [25, 62, 63] tries to improve the hypervisor-based type-1 VMs to achieve smaller memory footprint, shorter boot times, and higher execution performance. Tiny Core Linux [25]

trims an existing Linux distribution down as much as possible to reduce the overhead of the guest. OS^v [63] implements a new guest OS for running a single application on a VM, resolving libc function calls to its kernel that adopts optimization techniques such as the spinlock-free mutex [88] for addressing the lock-holder preemption problem and the net-channel networking stack [59] for reducing the number of locks. RumpKernel [62] reduces the VMs by implementing an optimized guest OS.

Different from KylinX, these general-purpose LibOS designs consist of unnecessary features for a target application leading to larger attack surface. They cannot support the multi-process abstraction. Besides, KylinX's `pVM` fork is much faster than replication-based `VM_fork` in SnowFlock [64] and booting in LightVM [74].

Memory deduplication has been widely studied at the hypervisor layer. For example, VMWare ESXi server [34] computes the hash of each physical memory page to identify duplication. To avoid false positives, ESXi has to perform byte-by-byte checking before sharing hash-identical pages. Disco [46] records every page change to track identical pages, which is more efficient than ESXi but requires non-trivial modifications to the guest OS. Glimpse [73] computes fingerprints over fixed-size blocks to find similar files. DifferenceEngine [53] exploits memory redundancy with delta encoding based on similarity comparison, which adapts well to read-intensive scenarios but performs inefficiently in write-intensive systems due to frequent recomputation and induces high overhead in computing the intersection of many pages.

5.2 Containers

Containers are not intended for isolation but for packaging. They use OS-level virtualization techniques [83] and leverage kernel features to package processes, instead of relying on the hypervisors or a dedicated kernel. In return, they do not need to trap system calls or emulate the hardware and could run as normal OS processes. For example, Linux Containers (LXC) [14] and Docker [20] create containers by using a number of Linux kernel features (such as *namespaces* and *cgroups*) to package various resources and run container-based processes.

Containers require to use the same host OS API [63] and thus expose hundreds of system calls and enlarge the attack surface of the host. Therefore, although LXC and Docker containers are usually more efficient than traditional VMs, their host OSs expose the entire host kernel ABI (application binary interface), and thus they provide less security guarantees, since attackers may compromise processes running inside containers.

5.3 Picoprocess

A picoprocess is essentially a container that implements a LibOS between the host OS and the guest mapping high-level guest API onto a small interface and providing lightweight isolation. The original picoprocess design (such as Xax [50] and Embassies [56]) only permits a tiny syscall API, which can be small enough to be convincingly (even verifiably) isolated. Howell et al. show how to support a small subset of single-process applications on top of a minimal picoprocess interface [57] by providing a POSIX emulation layer and binding existing programs.

Recent studies relax the static and rigid picoprocess isolation model and greatly extend the applicability of picoprocess. For example, Drawbridge [79] is a Windows translation of the Xax [50] picoprocess and creates a picoprocess LibOS that supports rich desktop applications. Graphene [85] broadens the LibOS paradigm by supporting multi-process API in a family (sandbox) of picoprocesses (using message passing). Bascule [41] allows OS-independent extensions to be attached safely and efficiently at runtime. Tardigrade [69] uses picoprocesses to easily construct fault-tolerant services. cKernel [91] leverages picoprocess to provide the virtual execution environment

abstraction. The success of these relaxations on picoprocess inspires our dynamic KylinX extension to Unikernel.

Containers and picoprocesses tend to have a large TCB, since the LibOSs often contain unused features. In contrast, KylinX and other Unikernels leverage the virtual hardware abstraction (e.g., event channels and shared memory) safely exposed by the hypervisor to simplify their implementation and follow the *minimalism* philosophy [45] to link an application only against requisite libraries to improve not only efficiency but also security.

Dune [43] leverages Intel VT-x [87] to provide a process (rather than a machine) abstraction to isolate processes and access privileged hardware features. It augments UNIX processes with isolation and enhances the performance by allowing them to access privileged CPU features. Going one step further, IX [44] incorporates virtual devices into the Dune process model and achieves high throughput and low latency for networked systems. lwCs [68] provides independent units of protection, privilege, and execution states within a process. These designs have realized some similar properties of KylinX; for example, access to (virtual) hardware features while providing the process abstraction. Dune and IX achieve this in a way different from Unikernel LibOSs by augmenting processes, and they have essentially implemented a hypervisor. Compared to these techniques, KylinX runs directly on Xen (a type-1 hypervisor), which naturally provides strong isolation and allows KylinX to focus on the flexibility and efficiency issues. Further, leveraging hardware support (e.g., MMU virtualization required by Dune) for process isolation in a libOS not only introduces a duplicate scheduler and memory management, but also complicates conventional management functionalities (like live migration) across platforms, because hardware vendors (Intel and AMD) have incompatible MMU virtualization support [85].

Haven [42], SCONE [37], Ryoan [58], and Graphene-SGX [86] leverage hardware protection of Intel SGX [17] to achieve shielded execution in processes against attacks from malicious operating systems and hypervisors. These studies are orthogonal to KylinX and thus could be incorporated into KylinX. For instance, we may put a `pVM` into the trusted *enclave* for strong confidentiality.

6 CONCLUSION

The tension between strong isolation and rich features has been long lived in the literature of cloud virtualization. This article exploits the new design space and proposes the `pVM` abstraction by adding two new features (dynamic page and library mapping) to the highly specialized static Unikernel. The simplified virtualization architecture (KylinX) takes the hypervisor as an OS and safely supports flexible process-style operations such as `pVM` fork and inter-`pVM` communication, runtime update, and fast recycling. In the future, we will extend KylinX syscalls for compatibility to more conventional Linux applications and improve isolation and security through modularization [36], disaggregation [76], and SGX enclaves [37, 42, 58, 86]. We will improve the performance of KylinX by adopting more efficient runtime like MUSL [31] and faster TCP/IP stacks supporting `epoll`, and adapt the KylinX architecture to the MultiLibOS model [82], which allows spanning `pVMs` onto multiple machines in the cloud.

ACKNOWLEDGMENTS

We thank Dr. Ziyang Li, Huiba Li, Qiao Zhou, and the anonymous reviewers for their help in improving this article. This work was performed when the first author was visiting the NetOS group of Computer Lab at University of Cambridge, and we thank Professor Anil Madhavapeddy, Ripduman Sohan, and Hao Liu for the discussion. A preliminary version of this article was presented at USENIX Annual Technical Conference 2018.

REFERENCES

[1] Erlangonxen.org. [n.d.]. Retrieved December 23, 2020 from https://github.com/cloudozer/ling.
[2] The LuaJIT Project. [n.d.]. Retrieved December 23, 2020 from http://luajit.org/.
[3] The Linux man-pages project. [n.d.]. Retrieved December 23, 2020 from https://man7.org/linux/man-pages/man2/syscalls.2.html.
[4] AWS. [n.d.]. Retrieved December 23, 2020 from https://aws.amazon.com/ec2/.
[5] AWS. [n.d.]. Retrieved December 23, 2020 from https://aws.amazon.com/cn/lambda/.
[6] Microsoft. [n.d.]. Retrieved December 23, 2020 from https://azure.microsoft.com/en-us/services/kubernetes-service/.
[7] Wikipedia. [n.d.]. Retrieved December 23, 2020 from https://en.wikipedia.org/wiki/Daytime_Protocol.
[8] Firecracker. [n.d.]. Retrieved December 23, 2020 from https://firecracker-microvm.github.io/.
[9] Thomas Porzelt. [n.d.]. Retrieved December 23, 2020 from https://github.com/lighttpd/weighttp/.
[10] gVisor. [n.d.]. Retrieved December 23, 2020 from https://gvisor.dev/.
[11] The HTTP server project. [n.d.]. Retrieved December 23, 2020 from https://httpd.apache.org/.
[12] Katacontainers. [n.d.]. Retrieved December 23, 2020 from https://katacontainers.io/.
[13] L4Linux. [n.d.]. Retrieved December 23, 2020 from https://l4linux.org/.
[14] Canonical Ltd. [n.d.]. Retrieved December 23, 2020 from https://linuxcontainers.org/.
[15] The OCaml Project. [n.d.]. Retrieved December 23, 2020 from https://ocaml.org/.
[16] MIT PDOS. [n.d.]. Retrieved December 23, 2020 from https://pdos.csail.mit.edu/6.828/2018/labs/lab6/.
[17] Intel. [n.d.]. Retrieved December 23, 2020 from https://qdms.intel.com/dm/i.aspx/5A160770-FC47-47A0-BF8A-062540456F0A/PCN114074-00.pdf.
[18] RedisLabs. [n.d.]. Retrieved December 23, 2020 from https://redis.io/.
[19] The Xen Project. [n.d.]. Retrieved December 23, 2020 from https://wiki.xenproject.org/wiki/Mini-OS.
[20] Docker, Inc. [n.d.]. Retrieved December 23, 2020 from https://www.docker.com/.
[21] The GNU C Library Project. [n.d.]. Retrieved December 23, 2020 from https://www.gnu.org/software/libc/.
[22] Internet Systems Consortium, Inc. [n.d.]. Retrieved December 23, 2020 from https://www.isc.org/downloads/bind/.
[23] Philip Guenther. [n.d.]. Retrieved December 23, 2020 from https://www.openbsd.org/papers/eurobsdcon2014_securelazy/slide003a.html.
[24] Unikraft Team. [n.d.]. Retrieved December 23, 2020 from https://xenproject.org/developers/teams/unikraft/.
[25] The Core Project. [n.d.]. Retrieved December 23, 2020 from http://tinycorelinux.net/.
[26] The UML Project. [n.d.]. Retrieved December 23, 2020 from http://user-mode-linux.sourceforge.net/.
[27] The GNU Hurd Project. [n.d.]. Retrieved December 23, 2020 from https://www.gnu.org/software/hurd/hurd.html.
[28] Game Sparks Technologies Ltd. [n.d.]. Retrieved December 23, 2020 from http://www.gamesparks.com/.
[29] Oracle. [n.d.]. Retrieved December 23, 2020 from http://www.java.com/.
[30] Red Hat, Inc. [n.d.]. Retrieved December 23, 2020 from https://www.linux-kvm.org/.
[31] The Musl Libc Project. [n.d.]. Retrieved December 23, 2020 from http://musl.libc.org/.
[32] SAS Institute, Inc. [n.d.]. Retrieved December 23, 2020 from http://www.sas.com/en_us/home.html.
[33] Jack Clark. [n.d.]. Retrieved December 23, 2020 from https://www.theregister.com/2014/05/23/google_containerization_two_billion/.
[34] Mohammed Raffic. [n.d.]. Retrieved December 23, 2020 from http://www.vmwarearena.com/difference-between-vmware-esx-and-esxi/.
[35] Samer Al-Kiswany, Dinesh Subhraveti, Prasenjit Sarkar, and Matei Ripeanu. 2011. VMFlock: Virtual machine co-migration for the cloud. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*. ACM, 159–170.
[36] Melvin J. Anderson, Micha Moffie, and Chris I. Dalton. 2007. *Towards Trustworthy Virtualisation Environments: Xen Library OS Security Service Infrastructure*. Technical Report. HP Tech Report (2007), 88–111.
[37] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel O'Keeffe, Mark L. Stillwell, et al. 2016. SCONE: Secure Linux containers with Intel SGX. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*.
[38] Jeff Arnold and M. Frans Kaashoek. 2009. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the 4th ACM European Conference on Computer Systems*. ACM, 187–198.
[39] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. *ACM SIGOPS Op. Syst. Rev.* 37, 5 (2003), 164–177.
[40] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. ACM, 29–44.

[41] Andrew Baumann, Dongyoon Lee, Pedro Fonseca, Lisa Glendenning, Jacob R. Lorch, Barry Bond, Reuben Olinsky, and Galen C. Hunt. 2013. Composing OS extensions safely and efficiently with Bascule. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 239–252.

[42] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding applications from an untrusted cloud with haven. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*.

[43] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe user-level access to privileged CPU features. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*. 335–348.

[44] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. 49–65.

[45] R. John Brockmann. 1990. The why, where and how of minimalism. In *ACM SIGDOC Aster. J. Comput. Docum.* 14 (1990). ACM, 111–119.

[46] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. 1997. DISCO: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP'97)*, Michel Banâtre, Henry M. Levy, and William M. Waite (Eds.). ACM, 143–156. DOI: https://doi.org/10.1145/268998.266672

[47] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. 2004. A microrebootable system—Design, implementation, and evaluation. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*.

[48] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. 2007. POLUS: A powerful live updating system. In *Proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society, 271–281.

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

[50] John R. Douceur, Jeremy Elson, Jon Howell, and Jacob R. Lorch. 2008. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, Vol. 8. 339–354.

[51] Dawson R. Engler, M. Frans Kaashoek et al. 1995. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. ACM.

[52] Bryan Ford and Jay Lepreau. 1994. Evolving mach 3.0 to a migrating thread model. In *Proceedings of the USENIX Winter Technical Conference*. USENIX Association, 97–114. Retrieved from https://www.usenix.org/conference/usenix-winter-1994-technical-conference/evolving-mach-30-migrating-thread-model.

[53] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. 2010. Difference engine: Harnessing memory redundancy in virtual machines. *Commun. ACM* 53, 10 (2010), 85–93.

[54] Christopher M. Hayden, Edward K. Smith, Michail Denchev, Michael Hicks, and Jeffrey S. Foster. 2012. Kitsune: Efficient, general-purpose dynamic software updating for C. *ACM SIGPLAN Not.* 47 (2012). ACM, 249–264.

[55] Jin Heo, Xiaoyun Zhu, Pradeep Padala, and Zhikui Wang. 2009. Memory overbooking and dynamic control of Xen virtual machines in consolidated environments. In *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management (IM'09)*. IEEE, 630–637.

[56] Jon Howell, Bryan Parno, and John R. Douceur. 2013. Embassies: Radically refactoring the web. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*. 529–545.

[57] Jon Howell, Bryan Parno, and John R. Douceur. 2013. How to run POSIX apps in a minimal picoprocess. In *Proceedings of the USENIX Annual Technical Conference*. 321–332.

[58] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. 2016. Ryoan: A distributed sandbox for untrusted computation on secret data. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 533–549.

[59] Van Jacobson and Bob Felderman. 2006. Speeding up networking. In *Proceedings of the Ottawa Linux Symposium*.

[60] Poul-Henning Kamp and Robert N. M. Watson. 2000. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference* 43 (2000), 116.

[61] Svilen Kanev, Juan Pablo Darago, Kim M. Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David M. Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd International Symposium on Computer Architecture*, Deborah T. Marr and David H. Albonesi (Eds.). ACM, 158–169. DOI: https://doi.org/10.1145/2749469.2750392

[62] Anti Kantee and Justin Cormack. 2014. Rump kernels: No OS? No problems! *Mag. USENIX & SAGE* 39, 5 (2014), 11–17.

[63] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. 2014. OSv: Optimizing the operating system for virtual machines. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'14)*. 61–72.

[64] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal De Lara, Michael Brudno, and Mahadev Satyanarayanan. 2009. SnowFlock: Rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European Conference on Computer Systems*. ACM, 1–12.

[65] Robert P. Cook and Insup Lee. 2003. Dymos: A dynamic modification system. *ACM SIGSOFT Software Engineering Notes* 18, 8 (2003), 202–202.

[66] Huiba Li, Yiming Zhang, Dongsheng Li, Zhiming Zhang, Shengyun Liu, Peng Huang, Zheng Qin, Kai Chen, and Yongqiang Xiong. 2019. URSA: Hybrid block storage for cloud-scale virtual disks. In *Proceedings of the 14th EuroSys Conference*. 15:1–15:17. DOI: https://doi.org/10.1145/3302424.3303967

[67] Huiba Li, Yiming Zhang, Zhiming Zhang, Shengyun Liu, Dongsheng Li, Xiaohui Liu, and Yuxing Peng. 2017. PARIX: Speculative partial writes in erasure-coded systems. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 581–587.

[68] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-weight contexts: An OS abstraction for safety and performance. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. USENIX, 49–64.

[69] Jacob R. Lorch, Andrew Baumann, Lisa Glendenning, Dutch T. Meyer, and Andrew Warfield. 2015. Tardigrade: Leveraging lightweight virtual machines to easily and efficiently construct fault-tolerant services. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*. 575–588.

[70] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, et al. 2015. Jitsu: Just-in-time summoning of unikernels. In *Proceedings of the 12th USENIX Symposium on Networked System Design and Implementation*.

[71] Anil Madhavapeddy, Richard Mortier, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library operating systems for the cloud. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 461–472.

[72] Anil Madhavapeddy, Richard Mortier, Ripduman Sohan, Thomas Gazagnaire, Steven Hand, Tim Deegan, Derek McAuley, and Jon Crowcroft. 2010. Turning down the LAMP: Software specialisation for the cloud. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. 11–11.

[73] Udi Manber and Sun Wu. 1994. GLIMPSE: A tool to search through entire file systems. In *Proceedings of the USENIX Winter Technical Conference*. 23–32.

[74] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is lighter (and safer) than your container. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*. ACM, 218–233.

[75] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the art of network function virtualization. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*. USENIX Association, 459–473.

[76] Derek Gordon Murray, Grzegorz Milos, and Steven Hand. 2008. Improving Xen security through disaggregation. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM, 151–160.

[77] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. 2006. *Practical dynamic software updating for C. ACM SIGPLAN Not.* 41 (2006). ACM.

[78] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. 2019. A binary-compatible unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, (VEE'19)*, Jennifer B. Sartor, Mayur Naik, and Chris Rossbach (Eds.). ACM, 59–73. DOI: https://doi.org/10.1145/3313808.3313817

[79] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. 2011. Rethinking the library OS from the top down. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*. ACM, 291–304.

[80] Ashwin Ramaswamy, Sergey Bratus, Sean W. Smith, and Michael E. Locasto. 2010. Katana: A hot patching framework for elf executables. In *Proceedings of the International Conference on Availability, Reliability, and Security (ARES'10)*. IEEE, 507–512.

[81] Joanna Rutkowska and Rafal Wojtczuk. 2010. *Qubes OS Architecture*. Technical Report. Invisible Things Lab Tech Rep 54 (2010).

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

[83] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. 2007. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *ACM SIGOPS Op. Syst. Rev.* 41 (2007). ACM, 275–287.

[84] Chia-che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. 2016. A study of modern Linux API usage and compatibility: What to support when you're supporting. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys'16)*, Cristian Cadar, Peter R. Pietzuch, Kimberly Keeton, and Rodrigo Rodrigues (Eds.). ACM, 16:1–16:16. DOI: https://doi.org/10.1145/2901318.2901341

[85] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. 2014. Cooperation and security isolation of library OSes for multi-process applications. In *Proceedings of the 9th European Conference on Computer Systems*. ACM, 9.

[86] Chia-Che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'17)*.

[87] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kägi, Felix H. Leung, and Larry Smith. 2005. Intel virtualization technology. *Computer* 38, 5 (2005), 48–56.

[88] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. 2004. Towards scalable multiprocessor virtual machines. In *Proceedings of the Virtual Machine Research and Technology Symposium*. 43–56.

[89] Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K. K. Ramakrishnan, and Timothy Wood. 2016. Flurries: Countless fine-grained NFs for flexible per-flow customization. In *Proceedings of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT'16)*. ACM, 3–17.

[90] Yiming Zhang, Chuanxiong Guo, Dongsheng Li, Rui Chu, Haitao Wu, and Yongqiang Xiong. 2015. CubicRing: Enabling one-hop failure detection and recovery for distributed in-memory storage systems. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*. 529–542.

[91] Yiming Zhang, Dongsheng Li, Qiao Zhou, Feng Huang, Yingwen Chen, Yang Hu, Ping Zhong, Yongqiang Xiong, and Huaimin Wang. 2018. The fusion of VMs and processes: A system perspective of cKernel. In *Proceedings of the 38th IEEE International Conference on Distributed Computing Systems (ICDCS'18)*. 1404–1409. DOI: https://doi.org/10.1109/ICDCS.2018.00141

Received April 2019; revised July 2020; accepted November 2020