KylinX: A Dynamic Library Operating System for Simplified and Efficient Cloud Virtualization

Yiming Zhang, *NiceX Lab, NUDT;* Jon Crowcroft, *University of Cambridge;* Dongsheng Li and Chengfen Zhang, *NUDT;* Huiba Li, *Alibaba;* Yaozheng Wang and Kai Yu, *NUDT;* Yongqiang Xiong, *Microsoft;* Guihai Chen, *SJTU*

https://www.usenix.org/conference/atc18/presentation/zhang-yiming

This paper is included in the Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-931971-44-7

Open access to the Proceedings of the 2018 USENIX Annual Technical Conference is sponsored by USENIX.

KylinX: A Dynamic Library Operating System for Simplified and Efficient Cloud Virtualization

Yiming Zhang
NiceX Lab, NUDT

Jon Crowcroft
University of Cambridge

Dongsheng Li, Chengfei Zhang
NUDT

Huiba Li
Alibaba

Yaozheng Wang, Kai Yu
NUDT

Yongqiang Xiong
Microsoft

Guihai Chen
SJTU

Abstract

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.

This paper examines 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). At the library level, KylinX supports shared libraries to be linked to a Unikernel appliance at runtime. KylinX enforces mapping restrictions against potential threats. 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.

1 Introduction

Commodity clouds (like EC2 [5]) 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 [24] and game servers [20], and are referred to as (virtual) appliances [56, 64]. The highly-specialized, single-purpose appliances need only a very small portion of traditional 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 [62, 52], inefficient to run [56], and vulnerable to bugs of unnecessary libraries [27].

This problem has recently motivated the design of Unikernel [56], a library operating system (LibOS) architecture that is targeted for efficient and secure appliances in the clouds. Unikernel refactors a traditional OS into libraries, and seals the application binary and requisite libraries into a specialized appliance image which could run directly on a hypervisor such as Xen [30] and KVM [22]. Compared to traditional VMs, Unikernel appliances eliminate unused code, and achieve smaller memory footprint, shorter boot times and lower overhead while guaranteeing the same level of isolation [56]. The hypervisor's steady interface avoids hardware compatibility problems encountered by early LibOSs [39].

On the downside, Unikernel strips off the *process* abstraction from its statically-sealed monolithic appliances, and thus sacrifices flexibility, efficiency, and applicability. 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 and performance of Unikernel.

In this paper, 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 restrictions.

We have implemented a prototype of KylinX based on Xen [30] (a type-1 hypervisor) by modifying Mini-OS [14] (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 [13] and a web server [11]) shows that KylinX achieves higher applicability and performance than static Unikernels while retaining the isolation guarantees.

The rest of this paper is organized as follows. Section 2 introduces the background and design options. Section 3 presents the design of dynamically-customized KylinX LibOS with security restrictions. Section 4 reports the evaluation results of the KylinX prototype implementation. Section 5 introduces related work. And Section 6 concludes the paper and discusses future work.

2 Preliminaries

2.1 VMs, Containers & Picoprocesses

There are several conventional models in the literature of virtualization and isolation: 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 [47]. 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.

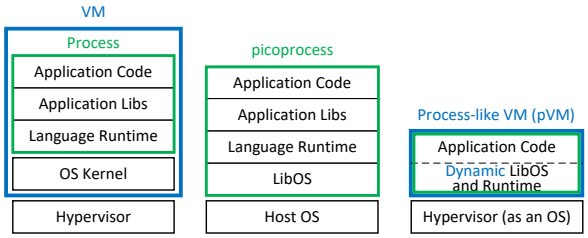

Figure 1: Alternative virtualization architectures.

VMs (Fig. 1(*left*)) have been widely used in multi-tenant clouds since it guarantees strong (type-1-hypervisor) isolation [55]. However, the current virtualization architecture of VMs is heavy with layers of hypervisor, VM, OS kernel, process, language runtime (such as glibc [16] and JVM [21]), libraries, and application, which are complex and could no longer satisfy the efficiency requirements of commercial clouds.

Containers (like LXC [9] and Docker [15]) leverage kernel features to package and isolate processes. They are recently in great demand [25, 7, 6] because they are lightweight compared to VMs. However, containers offer weaker isolation than VMs, and thus they often run in VMs to achieve proper security guarantees [58].

Picoprocesses [38] (Fig. 1 (*center*)) could be viewed as containers with stronger isolation but lighter-weight host obligations. They use a small interface between the host OSs and the guests to implement a LibOS realizing the host ABI and map high-level guest API onto the small interface. Picoprocesses are particularly suitable for client software delivery because client software needs to run on various host hardware and OS combinations [38]. They could also run on top of hypervisors [62, 32].

Recent studies [67, 32, 54] on picoprocesses relax the original static isolation model by allowing dynamics. For example, Graphene [67] supports picoprocess fork and multi-picoprocess API, and Bascule [32] allows OS-independent extensions to be attached to a picoprocess at runtime. 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 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, etc. 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) [58].

Recently, researchers propose to reduce VMs, instead of augmenting processes, to achieve secure and efficient cloud virtualization [56, 19, 49]. Unikernel [56] is focused on single-application VM appliances [26] and adapts the Exokernel [39] style LibOS to VM guests to enjoy performance benefits while preserving the strong isolation guarantees of a type-1 hypervisor. It breaks the traditional general-purpose virtualization architecture (Fig. 1 (*left*)) and implements the OS features (e.g., device drivers and networking) as libraries. Compared to other hypervisor-based reduced VMs (like Tiny Core Linux [19] and OS^{v} [49]), 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 [36], which minimizes the VMs by not only removing unnecessary libraries but also stripping off the duplicated management features from its LibOS. For example, Mirage [57] follows the multikernel model [31] and leverages the hypervisor for multicore scheduling, so that the single-threaded runtime could have fast sequential performance; MiniOS [14] relies on the hypervisor (instead of an in-LibOS linker) to load/link the appliance at boot time; and LightVM [58] achieves fast VM boot by redesigning Xen's control plane.

2.3 Motivation & Design Choices

Unikernel appliances and conventional UNIX processes 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 processes and compile-time determined monolithicity largely reduce Unikernel's flexibility, efficiency, and applicability.

As shown in Fig. 1 (*right*), KylinX provides the `pVM` abstraction by explicitly taking the hypervisor as an OS and the Unikernel appliance as a process. KylinX slightly relaxes Unikernel's compile-time monolithicity requirement to allow both page-level and library-level dynamic mapping, so that `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 extention of Graphene [67] (providing conventional multi-process compatibility) and Bascule [32] (providing runtime extensibility) to picoprocess.

	Static	**Dynamic**
Picoprocess	Embassies [43], Xax [38], etc.	Graphene [67], Bascule [32], etc
Unikernel	Mirage [57], MiniOS [14], etc.	**KylinX**

Table 1: Inspired by dynamic picoprocesses, KylinX explores new design space and extends the applicability of Unikernel.

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 (§3.2.3 and §3.3.4) 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 (§3.3.2). And third, it is natural for KylinX to follow Unikernel's minimalism philosophy (§2.2) of leveraging the hypervisor to eliminate duplicated guest LibOS features.

Backward compatibility is another tradeoff. The original Mirage Unikernel [56] takes an extreme position where existing applications and libraries have to be completely rewritten in OCaml [10] 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 that a large variety of legacy applications could run on KylinX with minimum effort for adaptation.

Threat model. KylinX assumes a traditional threat model [56, 49], the same context as Unikernel [56] 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 like SSL and SSH helps KylinX `pVMs` trust external entities.

Recent advance in hardware like Intel Software Guard eXtensions (SGX) [12] demonstrates the feasibility of shielded execution in *enclaves* to protect VMs/`pVMs` from the privileged hypervisor and dom0 [33, 28, 45], 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 [34].

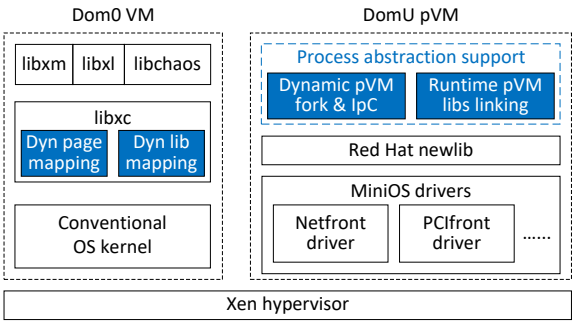

Figure 2: KylinX components. Blue parts are newly designed. DomU pVM is essentially a Unikernel appliance.

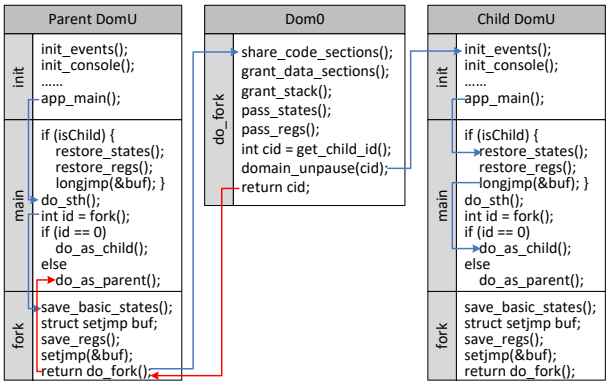

Figure 3: pVM fork. After *fork* is invoked, KylinX creates a child pVM by sharing the parent (caller) pVM's pages.

3 KylinX Design

3.1 Overview

KylinX extends Unikernel to realize desirable features that are previously applicable only to processes. Instead of designing a new LibOS from scratch, we base KylinX on MiniOS [27], 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 [59] and LightVM [58] leverage MiniOS to provide Erlang, Lua, Click and fast boot environments, respectively.

As shown in Fig. 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 memory sharing mechanism of Xen to implement the fork operation, which creates a *child* pVM by (i) duplicating the xc_dom_image structure and (ii) invoking Xen's unpause() API to fork the calling parent pVM and return its domain ID to the parent. As shown in Fig. 3, when fork() is invoked in the parent pVM, we use inline assemblies to get the current states of CPU registers and pass them to the child. The control domain (dom0) is responsible for forking and starting the child pVM. We modify *libxc* to keep the xc_dom_image structure in memory when the parent pVM was created, so that when fork() is invoked the structure could be directly mapped to the virtual address space of the child pVM and then the parent could share sections with the child using grant tables. Writable data is shared in a copy-on-write (CoW) manner.

After the child pVM is started via unpause(), it (i) accepts the shared pages from its parent, (ii) restores the CPU registers and jumps to the next instruction after fork, 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.

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 [67] where only mutually-trusted pVMs can communicate with each other, which will be discussed in more details in §3.2.3.

The two communicating pVMs use an event channel and shared pages 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() (§3.2.1), then KylinX will (i) verify their mutual trustworthiness (§3.2.3), (ii) initialize an event channel, and (iii) share dedicated pages between them.

Type	API	Description
Pipe	pipe	Create a pipe and return the *fd*s.
	write	Write *value* to a pipe.
	read	Read *value* from a pipe.
Signal	kill	Send signal to a domain.
	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.

Table 2: Inter-pVM communication API.

The event channel is used to notify events, and the shared pages are used to realize the communication. KylinX has already realized the following four types of inter-pVM communication APIs (listed in Table. 2).

(1) `pipe(fd)` creates a pipe and returns two file descriptors (`fd[0]` and `fd[1]`), one for `write` and the other for `read`.

(2) `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; `exit` and `wait` are implemented using `kill`.

(3) `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`.

(4) `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)`.

3.2.3 Dynamic Page Mapping Restrictions

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, so as to accommodate dynamic libraries after they are loaded.

As depicted in Fig. 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 (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()`) 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 (§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, etc.
- 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 [17] for conventional processes. Therefore, the boot times of KylinX pVMs are lower than that of previous Unikernel VMs. Further, compared to previous Unikernels which 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 [58] and 80K containers per machine in Flurries [71]), which is the single biggest factor [58] 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 [56] 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 [51], Ksplice [29], Ginseng [61], PoLUS [37], Katana [63], Kitsune [41], 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/variables, or change fields of structures. For library states, we expect all the states are stored as variables (or dynamically-allocated structures) that would be saved and restored during update.

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.

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, dom0 will (i) map the new library into dom0's virtual address space; (ii) share the loaded library with domU; (iii) verify whether the old library is quiescent by asking domU to check the call stack of each kernel thread of domU; (iv) wait until the old library is not in use and pause the execution; (v) modify the entries of affected symbols to the proper addresses; and finally (vi) release the old library. In the above 5th step, there are two kinds of symbols (*functions* and *variables*) which will be resolved as discussed below.

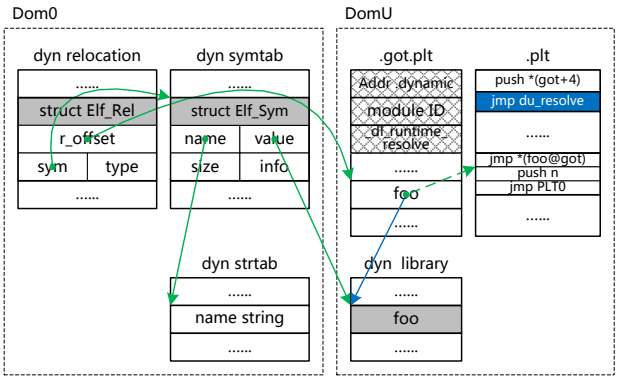

Figure 4: KylinX dynamic symbol resolution for functions. The green lines represent pointers for normal lazy binding of processes. The blue line represents the result of KylinX's resolution, pointing to the real function instead of the .plt table entry (dashed green line).

Functions. The dynamic resolution procedure for functions is illustrated in Fig. 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. In order to resolve symbols across different domains, we modify the 2nd line of assembly in the 1st entry of the .plt table (as shown in the blue region in Fig. 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 Fig. 4) is modified to point to the corresponding entry in the .plt table, i.e., the 2nd assembly (`push n`) shown by the dashed green line in Fig. 4. When a function (`foo`) of the library is called for the first time after `new_lib` is loaded, `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 Fig. 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 need extensions to allow developers to specify their intention for each variable.

(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 (§3.3.1) of KylinX pVMs and Unikernel VMs [58] is relatively slow. As evaluated in §4.1, it takes 100+ ms 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 and waits for running an application before calling the app_entry function of a *placeholder* dynamic library. The application is compiled into a shared library instead of a bootable image, using app_entry as its entry. To accelerate the booting of a pVM for the application, KylinX restores the checkpointed domain, and links the application library by replacing the placeholder library following the online update procedure (§3.3.2).

3.3.4 Dynamic Library Mapping Restrictions

KylinX should 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 (§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 (§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

We have implemented a prototype of KylinX on top of Ubuntu 16.04 and Xen. Following the default settings of MiniOS [14], we respectively use RedHat Newlib and lwIP as the libc/libm libraries and TCP/IP stack. Our testbed has two machines each of which has an Intel 6-core Xeon E5-2640 CPU, 128 GB RAM, and one 1GbE NIC.

We have ported a few applications to KylinX, among which we will use a multi-process Redis server [13] as well as a multi-thread web server [11] to evaluate the application performance of KylinX in §4.6. Due to the limitation of MiniOS and RedHat Newlib, currently two kinds of adaptations are necessary for porting 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.

4.1 Standard Boot

We evaluate the time of the standard boot procedure (§3.3.1) of KylinX pVMs, and compare it with that of MiniOS VMs and Docker containers, all running a Redis

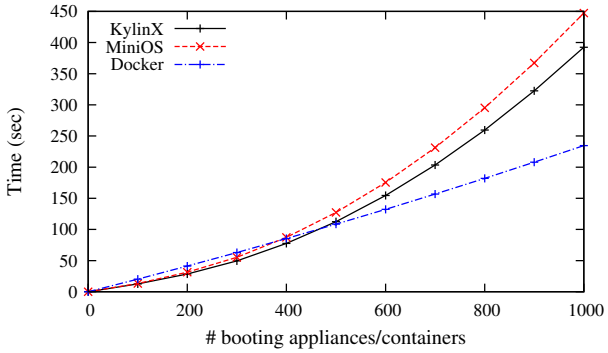

Figure 5: Total time of standard booting (reduced Redis).

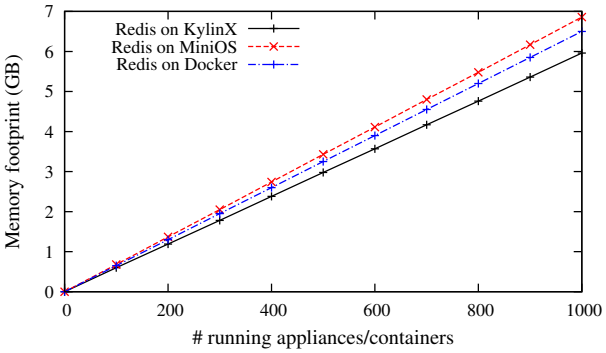

Figure 6: Memory usage (reduced Redis).

server. Redis is an in-memory key-value store that supports fast key-value storage/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.

We disable XenStore logging to eliminate the interference of periodic log file flushes. The C library (*libc*) of RedHat Newlib is static for use in embedded systems and difficult to be converted into a shared library. For simplicity, we compile *libc* 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. Since MiniOS cannot support `fork`, we (temporarily) remove the corresponding code in this experiment.

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 $\sim$ 4 in §3.3.1), and booting the image (step 5). Dynamic mapping is performed in the first stage. 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 Fig. 5. 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` [58].

4.2 Fork & Recycling

Compared to containers, KylinX's standard booting cannot scale well for a large number of `pVMs` due to the inefficient XenStore. Most recently, LightVM [58] completely redesigns Xen's control plane by implementing chaos/libchaos, `noxs` (no XenStore), and split toolstack, together with a number of other optimizations, so as to achieve ms-level booting times for a large number of VMs. We adopt LightVM's `noxs` for eliminating XenStore's affect and test the `pVM` fork mechanism running *unmodified* Redis emulating conventional process fork. LightVM's `noxs` enables the boot times of KylinX `pVMs` to increase linearly even for a large number of `pVMs`. The fork of a single `pVM` takes about 1.3 ms (not shown here due to lack of space), 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 fork.

4.3 Memory Footprint

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 Fig. 6) 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 (§3.3), and thus the shared libraries need to be loaded at most once. The memory footprint advantage facilitates ballooning [42] which could be used to dynamically share physical memory between VM appliances, and enables KylinX to achieve comparable memory efficiency with page-level deduplication [40] while introducing much less complexity.

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

Table 3: IpC vs. IPC in latency (μs). KylinX[1]: a pair of lineal pVMs which already have an event channel and shared pages. KylinX[2]: a pair of non-lineal pVMs.

4.4 Inter-pVM Communication

We evaluate the performance of inter-pVM communication (IpC) 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 §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 result is listed in Table 3, 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 through the pipe. The first-time communication latencies between non-lineal pVMs are several times higher due to the initialization cost.

4.5 Runtime Library Update

We evaluate runtime library update 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 4 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

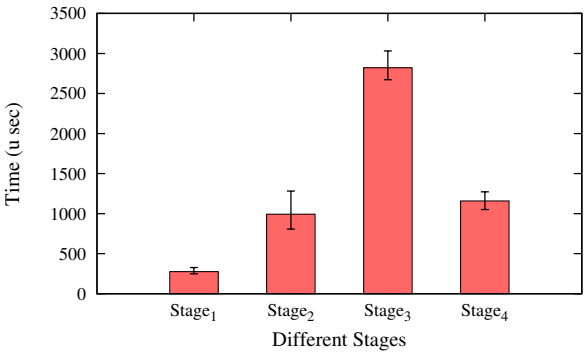

Figure 7: Runtime library update.

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. 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 (§3.3.2), so as to present an overview of the entire runtime update cost of *libm*.

The result is depicted in Fig. 7, 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 real function addresses and updating corresponding entries.

4.6 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 counterparts on Ubuntu, as evaluated in this subsection.

4.6.1 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. The Redis server uses select instead of epoll to realize asynchronous I/O, because epoll is not yet supported by the lwIP stack [4] used by MiniOS and KylinX.

We use the Redis benchmark [13] to evaluate the performance, which uses a configurable number of busy loops asynchronously writing KVs. We run different

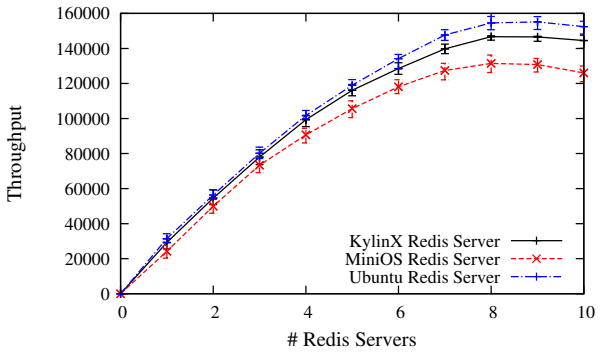

Figure 8: Redis server application.

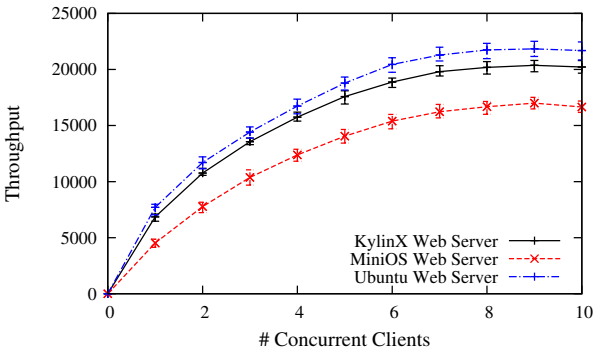

Figure 9: Web server application.

numbers of pVMs/VMs/processes (each for 1 server) servicing write requests from clients. We measure the write throughput as a function of the number of servers (Fig. 8). 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 8 instances before the lwIP stack becomes the bottleneck).

4.6.2 Web Server Application

We evaluate the JOS web server [11] 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 [8] clients on one machine, each continuously sending `GET` requests to the web server.

We evaluate the throughput 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 Fig. 9, where the KylinX web server achieves higher throughput than the MiniOS web server since it provides higher sequential performance. Both KylinX and MiniOS web servers are slower than the Ubuntu web server, because the asynchronous `select` is inefficiently scheduled with the netfront driver of MiniOS [27].

5 Related Work

KylinX is related to static Unikernel appliances [56, 27], reduced VMs [19, 48, 49], containers [66, 9, 15], and picoprocess [38, 62, 32, 54, 67, 33].

5.1 Unikernel & Reduced VMs

KylinX is an extension of Unikernel [56] and is implemented on top of MiniOS [27]. Unikernel OSs include Mirage [56], Jitsu [55], Unikraft [18], etc. For example, Jitsu [55] leverages Mirage [56] to design a power-efficient and responsive platform for hosting cloud services in the edge networks. LightVM [58] leverages Unikernel on Xen to achieve fast booting.

MiniOS [27] designs and implements a C-style Unikernel LibOS that runs as a para-virtualized guest OS within a Xen domain. 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 (like Mirage [56], MiniOS [27], and EbbRT [65]) lies in the `pVM` abstraction which explicitly takes the hypervisor as an OS and supports process-style operations like `pVM` fork/IpC and dynamic library mapping. Mapping restrictions (§3.3.4) make KylinX introduce as little vulnerability as possible and have no larger TCB than Mirage/MiniOS [56, 55]. KylinX supports source code (`C`) compatibility instead of using a type-safe language to rewrite the entire software stack [56].

Recent research [19, 49, 48] tries to improve the hypervisor-based type-1 VMs to achieve smaller memory footprint, shorter boot times, and higher execution performance. Tiny Core Linux [19] trims an existing Linux distribution down as much as possible to reduce the overhead of the guest. OS^v [49] 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 [70] and the net-channel networking stack [46]. RumpKernel [48] reduces the VMs by implementing a 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 [50].

5.2 Containers

Containers use OS-level virtualization [66] and leverage kernel features to package and isolate processes, instead of relying on the hypervisors. In return they do not need to trap syscalls or emulate hardware, and could run as normal OS processes. For example, Linux Containers (LXC) [9] and Docker [15] create containers by using a number of Linux kernel features (such as *namespaces* and *cgroups*) to package resource and run container-based processes.

Containers require to use the same host OS API [49], and thus expose hundreds of system calls and enlarging the attack surface of the host. Therefore, although LXC and Docker containers are usually more efficient than traditional VMs, they provide less security guarantees since attackers may compromise processes running inside containers.

5.3 Picoprocess

A picoprocess is essentially a container which implements a LibOS between the host OS and the guest, mapping high-level guest API onto a small interface. The original picoprocess designs (Xax [38] and Embassies [43]) only permit 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 [44], by providing a POSIX emulation layer and binding existing programs.

Recent studies relax the static and rigid picoprocess isolation model. For example, Drawbridge [62] is a Windows translation of the Xax [38] picoprocess, and creates a picoprocess LibOS which supports rich desktop applications. Graphene [67] broadens the LibOS paradigm by supporting multi-process API in a family (sandbox) of picoprocesses (using message passing). Bascule [32] allows OS-independent extensions to be attached safely and efficiently at runtime. Tardigrade [54] uses picoprocesses to easily construct fault-tolerant services. The success of these relaxations on picoprocess inspires our dynamic KylinX extension to Unikernel.

Containers and picoprocesses often have a large TCB since the LibOSs contain unused features. In contrast, KylinX and other Unikernels leverage the hypervisor's virtual hardware abstraction to simplify their implementation, and follow the *minimalism* philosophy [36] to link an application only against requisite libraries to improve not only efficiency but also security.

Dune [34] leverages Intel VT-x [69] to provide a process (rather than a machine) abstraction to isolate processes and access privileged hardware features. IX [35] incorporates virtual devices into the Dune process model and achieves high throughput and low latency for networked systems. lwCs [53] provides independent units of protection, privilege, and execution state within a process.

Compared to these techniques, KylinX runs directly on Xen (a type-1 hypervisor), which naturally provides strong isolation and enables KylinX to focus on the flexibility and efficiency issues.

6 Conclusion

The tension between strong isolation and rich features has been long lived in the literature of cloud virtualization. This paper 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 improve security through modularization [27], disaggregation [60], and SGX enclaves [33, 28, 45, 68]. We will improve the performance of KylinX by adopting more efficient runtime like MUSL [23], and adapt KylinX to the MultiLibOS model [65] which allows spanning pVMs onto multiple machines. Currently, the pVM recycling mechanism is still tentative and conditional: it can only checkpoint an empty domain; the recycled pVM cannot communicate with other pVMs using event channels or shared memory; the application can only be in the form of a self-contained shared library that does not need to load/link other shared libraries; and there are still no safeguards inspecting potential security threats between the new and old pVMs after recycling. We will address these shortcomings in our future work.

7 Acknowledgements

This work was supported by the National Basic Research Program of China (2014CB340303) and the National Natural Science Foundation of China (61772541). We thank Dr. Ziyang Li, Qiao Zhou, and the anonymous reviewers for their help in improving this paper. 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.

References

[1] http://erlangonxen.org/.

[2] http://luajit.org/.

[3] http://man7.org/linux/manpages/man2/syscalls.2.html.

[4] http://savannah.nongnu.org/projects/lwip/.

[5] https://aws.amazon.com/ec2/.

[6] https://aws.amazon.com/lambda/.

[7] https://azure.microsoft.com/en-us/services/container-service/.

[8] https://github.com/lighttpd/weighttp/.

[9] https://linuxcontainers.org/.

[10] https://ocaml.org/.

[11] https://pdos.csail.mit.edu/6.828/2014/labs/lab6/.

[12] https://qdms.intel.com/dm/i.aspx/5A160770-FC47-47A0-BF8A-062540456F0A/PCN114074-00.pdf.

[13] https://redis.io/.

[14] https://wiki.xenproject.org/wiki/Mini-OS.

[15] https://www.docker.com/.

[16] https://www.gnu.org/software/libc/.

[17] https://www.openbsd.org/papers/eurobsdcon2014_securelazy/slide003a.html.

[18] https://www.xenproject.org/developers/teams/unikraft.html.

[19] http://tinycorelinux.net/.

[20] http://www.gamesparks.com/.

[21] http://www.java.com/.

[22] http://www.linux-kvm.org/.

[23] http://www.musl-libc.org.

[24] http://www.sas.com/en_us/home.html.

[25] http://www.theregister.co.uk/2014/05/23/google_containerization_two_billion/.

[26] AL-KISWANY, S., SUBHRAVETI, D., SARKAR, P., AND RIPEANU, M. Vmflock: virtual machine co-migration for the cloud. In *Proceedings of the 20th international symposium on High performance distributed computing* (2011), ACM, pp. 159–170.

[27] ANDERSON, M. J., MOFFIE, M., AND DALTON, C. I. Towards trustworthy virtualisation environments: Xen library os security service infrastructure. *HP Tech Reort* (2007), 88–111.

[28] ARNAUTOV, S., TRACH, B., GREGOR, F., KNAUTH, T., MARTIN, A., PRIEBE, C., LIND, J., MUTHUKUMARAN, D., OKEEFFE, D., STILLWELL, M. L., ET AL. Scone: Secure linux containers with intel sgx. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Savannah, GA* (2016).

[29] ARNOLD, J., AND KAASHOEK, M. F. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the 4th ACM European conference on Computer systems* (2009), ACM, pp. 187–198.

[30] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review 37*, 5 (2003), 164–177.

[31] BAUMANN, A., BARHAM, P., DAGAND, P.-E., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 29–44.

[32] BAUMANN, A., LEE, D., FONSECA, P., GLENDENNING, L., LORCH, J. R., BOND, B., OLINSKY, R., AND HUNT, G. C. Composing os extensions safely and efficiently with bascule. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), ACM, pp. 239–252.

[33] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding applications from an untrusted cloud with haven. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014).

[34] BELAY, A., BITTAU, A., MASHTIZADEH, A., TEREI, D., MAZIÈRES, D., AND KOZYRAKIS, C. Dune: Safe user-level access to privileged cpu features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (2012), pp. 335–348.

[35] BELAY, A., PREKAS, G., KLIMOVIC, A., GROSSMAN, S., KOZYRAKIS, C., AND BUGNION, E. Ix: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014), pp. 49–65.

[36] BROCKMANN, R. J. The why, where and how of minimalism. In *ACM SIGDOC Asterisk Journal of Computer Documentation* (1990), vol. 14, ACM, pp. 111–119.

[37] CHEN, H., YU, J., CHEN, R., ZANG, B., AND YEW, P.-C. Polus: A powerful live updating system. In *Proceedings of the 29th international conference on Software Engineering* (2007), IEEE Computer Society, pp. 271–281.

[38] DOUCEUR, J. R., ELSON, J., HOWELL, J., AND LORCH, J. R. Leveraging legacy code to deploy desktop applications on the web. In *OSDI* (2008), vol. 8, pp. 339–354.

[39] ENGLER, D. R., KAASHOEK, M. F., ET AL. Exokernel: An operating system architecture for application-level resource management. In *Fifteenth ACM Symposium on Operating Systems Principles* (1995), ACM.

[40] GUPTA, D., LEE, S., VRABLE, M., SAVAGE, S., SNOEREN, A. C., VARGHESE, G., VOELKER, G. M., AND VAHDAT, A. Difference engine: Harnessing memory redundancy in virtual machines. *Communications of the ACM 53*, 10 (2010), 85–93.

[41] HAYDEN, C. M., SMITH, E. K., DENCHEV, M., HICKS, M., AND FOSTER, J. S. Kitsune: Efficient, general-purpose dynamic software updating for c. In *ACM SIGPLAN Notices* (2012), vol. 47, ACM, pp. 249–264.

[42] HEO, J., ZHU, X., PADALA, P., AND WANG, Z. Memory overbooking and dynamic control of xen virtual machines in consolidated environments. In *Integrated Network Management, 2009. IM'09. IFIP/IEEE International Symposium on* (2009), IEEE, pp. 630–637.

[43] HOWELL, J., PARNO, B., AND DOUCEUR, J. R. Embassies: Radically refactoring the web. In *NSDI* (2013), pp. 529–545.

[44] HOWELL, J., PARNO, B., AND DOUCEUR, J. R. How to run posix apps in a minimal picoprocess. In *USENIX Annual Technical Conference* (2013), pp. 321–332.

[45] HUNT, T., ZHU, Z., XU, Y., PETER, S., AND WITCHEL, E. Ryoan: a distributed sandbox for untrusted computation on secret data. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation* (2016), USENIX Association, pp. 533–549.

[46] JACOBSON, V., AND FELDERMAN, B. Speeding up networking. In *Ottawa Linux Symposium (July 2006)* (2006).

[47] KAMP, P.-H., AND WATSON, R. N. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference* (2000), vol. 43, p. 116.

[48] KANTEE, A., AND CORMACK, J. Rump kernels: no os? no problems! *The magazine of USENIX & SAGE 39*, 5 (2014), 11–17.

[49] KIVITY, A., LAOR, D., COSTA, G., ENBERG, P., HAREL, N., MARTI, D., AND ZOLOTAROV, V. Osv: optimizing the operating system for virtual machines. In *2014 usenix annual technical conference (usenix atc 14)* (2014), pp. 61–72.

[50] LAGAR-CAVILLA, H. A., WHITNEY, J. A., SCANNELL, A. M., PATCHIN, P., RUMBLE, S. M., DE LARA, E., BRUDNO, M., AND SATYANARAYANAN, M. Snowflock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European Conference on Computer Systems* (2009), ACM, pp. 1–12.

[51] LEE, I. Dymos: a dynamic modification system.

[52] LI, H., ZHANG, Y., ZHANG, Z., LIU, S., LI, D., LIU, X., AND PENG, Y. Parix: Speculative partial writes in erasure-coded systems. In *USENIX Annual Technical Conference* (2017), USENIX Association, pp. 581–587.

[53] LITTON, J., VAHLDIEK-OBERWAGNER, A., ELNIKETY, E., GARG, D., BHATTACHARJEE, B., AND DRUSCHEL, P. Light-weight contexts: An os abstraction for safety and performance. In *OSDI* (2016), USENIX, pp. 49–64.

[54] LORCH, J. R., BAUMANN, A., GLENDENNING, L., MEYER, D. T., AND WARFIELD, A. Tardigrade: Leveraging lightweight virtual machines to easily and efficiently construct fault-tolerant services. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015* (2015), pp. 575–588.

[55] MADHAVAPEDDY, A., LEONARD, T., SKJEGSTAD, M., GAZAGNAIRE, T., SHEETS, D., SCOTT, D., MORTIER, R., CHAUDHRY, A., SINGH, B., LUDLAM, J., ET AL. Jitsu: Just-in-time summoning of unikernels. In *12th USENIX Symposium on Networked System Design and Implementation* (2015).

[56] MADHAVAPEDDY, A., MORTIER, R., SMITH, S., HAND, S., AND CROWCROFT, J. Unikernels: Library operating systems for the cloud. In *ACM ASPLOS* (2013), ACM, pp. 461–472.

[57] MADHAVAPEDDY, A., MORTIER, R., SOHAN, R., GAZAGNAIRE, T., HAND, S., DEEGAN, T., MCAULEY, D., AND CROWCROFT, J. Turning down the lamp: software specialisation for the cloud. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, HotCloud* (2010), vol. 10, pp. 11–11.

[58] MANCO, F., LUPU, C., SCHMIDT, F., MENDES, J., KUENZER, S., SATI, S., YASUKATA, K., RAICIU, C., AND HUICI, F. My vm is lighter (and safer) than your container.

[59] MARTINS, J., AHMED, M., RAICIU, C., OLTEANU, V., HONDA, M., BIFULCO, R., AND HUICI, F. Clickos and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (2014), USENIX Association, pp. 459–473.

[60] MURRAY, D. G., MILOS, G., AND HAND, S. Improving xen security through disaggregation. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (2008), ACM, pp. 151–160.

[61] NEAMTIU, I., HICKS, M., STOYLE, G., AND ORIOL, M. *Practical dynamic software updating for C*, vol. 41. ACM, 2006.

[62] PORTER, D. E., BOYD-WICKIZER, S., HOWELL, J., OLINSKY, R., AND HUNT, G. C. Rethinking the library os from the top down. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2011), ACM, pp. 291–304.

[63] RAMASWAMY, A., BRATUS, S., SMITH, S. W., AND LOCASTO, M. E. Katana: A hot patching framework for elf executables. In *Availability, Reliability, and Security, 2010. ARES'10 International Conference on* (2010), IEEE, pp. 507–512.

[64] RUTKOWSKA, J., AND WOJTCZUK, R. Qubes os architecture. *Invisible Things Lab Tech Rep 54* (2010).

[65] SCHATZBERG, D., CADDEN, J., DONG, H., KRIEGER, O., AND APPAVOO, J. Ebbrt: A framework for building per-application library operating systems. In *OSDI* (2016), USENIX, pp. 671–688.

[66] SOLTESZ, S., PÖTZL, H., FIUCZYNSKI, M. E., BAVIER, A., AND PETERSON, L. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review* (2007), vol. 41, ACM, pp. 275–287.

[67] TSAI, C.-C., ARORA, K. S., BANDI, N., JAIN, B., JANNEN, W., JOHN, J., KALODNER, H. A., KULKARNI, V., OLIVEIRA, D., AND PORTER, D. E. Cooperation and security isolation of library oses for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 9.

[68] TSAI, C.-C., PORTER, D. E., AND VIJ, M. Graphene-sgx: A practical library os for unmodified applications on sgx. In *2017 USENIX Annual Technical Conference (USENIX ATC)* (2017).

[69] UHLIG, R., NEIGER, G., RODGERS, D., SANTONI, A. L., MARTINS, F., ANDERSON, A. V., BENNETT, S. M., KÄGI, A., LEUNG, F. H., AND SMITH, L. Intel virtualization technology. *Computer 38*, 5 (2005), 48–56.

[70] UHLIG, V., LEVASSEUR, J., SKOGLUND, E., AND DANNOWSKI, U. Towards scalable multiprocessor virtual machines. In *Virtual Machine Research and Technology Symposium* (2004), pp. 43–56.

[71] ZHANG, W., HWANG, J., RAJAGOPALAN, S., RAMAKRISHNAN, K., AND WOOD, T. Flurries: Countless fine-grained nfs for flexible per-flow customization. In *CoNEXT* (2016), ACM, pp. 3–17.