Unikernels as Processes

Dan Williams
IBM T.J. Watson Research Center
djwillia@us.ibm.com

Ricardo Koller
IBM T.J. Watson Research Center
kollerr@us.ibm.com

Martin Lucina
robur.io / Center for the Cultivation of Technology
martin@lucina.net

Nikhil Prakash
BITS Pilani
niks3089@gmail.com

ABSTRACT

System virtualization (e.g., the virtual machine abstraction) has been established as the de facto standard form of isolation in multi-tenant clouds. More recently, unikernels have emerged as a way to reuse VM isolation while also being lightweight by eliminating the general purpose OS (e.g., Linux) from the VM. Instead, unikernels directly run the application (linked with a library OS) on the virtual hardware. In this paper, we show that unikernels do not actually require a virtual hardware abstraction, but can achieve similar levels of isolation when running as processes by leveraging existing kernel system call whitelisting mechanisms. Moreover, we show that running unikernels as processes reduces hardware requirements, enables the use of standard process debugging and management tooling, and improves the already impressive performance that unikernels exhibit.

CCS CONCEPTS

• **Security and privacy** → **Virtualization and security**;
• **Computer systems organization** → *Cloud computing*;
• **Software and its engineering** → *Operating systems*;

KEYWORDS

unikernels, cloud computing, security, virtualization

ACM Reference Format:
Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. 2018. Unikernels as Processes. In *Proceedings of SoCC '18: ACM Symposium on Cloud Computing, Carlsbad, CA, USA, October 11–13, 2018 (SoCC '18),* 13 pages.
https://doi.org/10.1145/3267809.3267845

1 INTRODUCTION

Unikernels have emerged as a lightweight way to run applications in isolation from one another in a cloud environment. They consist of an application linked against only those parts of a library OS that the application needs to run directly on a virtual hardware abstraction, thereby inheriting the desirable isolation properties of virtual machines (VMs). Although originally limited to OCaml-based applications with MirageOS [34], unikernel communities have now emerged around many different languages [2, 3, 10, 18, 42] and some even support common applications and runtimes [9, 27] like `nginx`, `redis`, Node.js `express`, Python, etc. Due to their lightweight characteristics and strong isolation, unikernels are well suited for microservices [23, 44], network function virtualization [36], or emerging highly-responsive cloud workloads like so-called "serverless computing" [1, 7], where multiple mistrusting cloud users run small snippets of code in an event-based manner [29].

However, despite the isolation benefits, adopting VM-like characteristics introduces several issues. First, existing tooling for VMs was not designed for lightweight, highly-responsive cloud services and requires redesign for unikernels or lightweight VMs [35] to address memory density, performance, and startup time. Second, unikernels are like black-box VMs, which raises questions of how to debug them in production [19]. Finally, unikernels running as VMs cannot be deployed on top of—and take advantage of the elasticity properties of—already virtualized infrastructure without the performance, complexity and security implications of nested virtualization [17].

In this paper, we describe a technique to run unikernels *as processes*, while retaining their VM-like isolation properties. We believe that running unikernels as processes is an important step towards running them in production, because, as processes, they can reuse lighter-weight process or container tooling, be debugged with standard process debugging tools, and run in already-virtualized infrastructure.

Our approach is based on two observations. First, unikernels are more like applications than kernels and, as such, do not require supervisor-level hardware features (e.g., ring 0 on x86). Second, if isolation is achieved for VMs by sufficiently limiting access to the underlying host via a thin hypercall interface, then it should also be achievable for processes by sufficiently limiting access to the underlying host. To this point, kernel mechanisms exist (e.g., `seccomp` in Linux [24]) that can restrict the system call interface for processes.

We demonstrate unikernels running as processes by modifying the open-source `ukvm` unikernel monitor.[1] After setting up virtual device access with the host, instead of launching

[1] https://github.com/solo5/solo5

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SoCC '18, October 11–13, 2018, Carlsbad, CA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6011-1/18/10…$15.00
https://doi.org/10.1145/3267809.3267845

RIGHTSLINK

the unikernel as a guest VM, the monitor dynamically loads the unikernel into its own address space. The modified monitor, dubbed a *tender*, then switches to a restricted execution mode before jumping into the unikernel code. Importantly, we show there is a simple and direct mapping between hypercalls and system calls, which results in similar levels of isolation.

We evaluate the isolation of unikernels as processes on familiar cloud applications and runtimes and compare them to their virtualized equivalents, tracing system call and kernel execution in the host system. We find that when running as processes, unikernels invoke fewer system calls and subsequently access 50% fewer host kernel functions than when running as a VM. We also use fuzz testing to compare possible kernel access against normal processes, showing a 98% reduction in accessible kernel functions. When evaluating the performance of unikernels as processes, we find that running unikernels as processes can increase throughput up to 245%, reduce startup times by up to 73%, and increase memory density by 20%.

2 UNIKERNEL ISOLATION

In a multi-tenant cloud environment, mutually mistrusting cloud users share physical compute resources. On a single machine, we refer to *isolation* as the property that no cloud user can read or write state belonging to another cloud user or modify its execution. While isolation is built from hardware-based protection (e.g., page tables) which we trust,[2] we focus on deficiencies in the software exploitable through the software interfaces. As such, our threat model is one of a malicious tenant breaking out of its isolation through compromise of the interface below. Compromise is related to the attack surface of the layer underneath; we assume that the amount of code reachable through an interface is a metric for its attack surface [30, 48]. We do not consider covert channels, timing channels, or resource starvation.

In the remainder of this section, we review a typical unikernel architecture that uses virtualization techniques, discuss why it achieves isolation, and then identify some limitations with using a virtualization-based approach.

2.1 Unikernel Architecture

Figure 1 shows the typical architecture for unikernels, using virtualization hardware for isolation. In the diagram, the KVM module in the Linux kernel[3] interfaces with virtualization hardware (depicted VT-x) and exposes it to a userspace monitor process. For this paper, we limit the discussion to environments where the monitor process is a unikernel monitor, such as `ukvm` [47], which can be viewed as similar to QEMU, but specialized for running unikernels. While not ubiquitous, `ukvm` is used by diverse unikernel ecosystems, including MirageOS [34], IncludeOS [18], and Rumprun [9] on

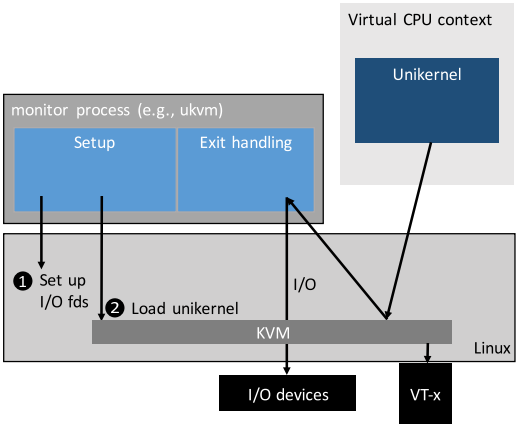

Figure 1: Unikernel isolation using a monitor process with virtualization technology

multiple systems (Linux, FreeBSD, OpenBSD) and architectures (x86_64 and ARM64).

The userspace monitor process has two main tasks: setup and exit handling. Setup consists of allocating the memory (typically specified on the command line) and virtual CPU structures necessary to start the unikernel as a VM. During setup, file descriptors are opened for system resources that the unikernel will need during execution. For example, the monitor may open a network tap device or a backing file for a virtual block device. The monitor will communicate with the KVM module in the kernel via system calls to actually load and start running the unikernel.

During execution, the unikernel will exit to the monitor via hypercalls, usually to perform I/O. For example, to send a network packet, the unikernel will exit to the monitor, passing a reference to the packet. Subsequently, the monitor will write the packet to the file descriptor associated with the network tap device.

2.2 Are Unikernels Isolated?

When using virtualization technology, isolation is derived from the interface between the unikernel and the monitor process. This interface is thin: the unikernel can exit to `ukvm` via at most 10 hypercalls.[4] Of the 10 hypercalls possible in `ukvm`, 6 are for I/O (`read`, `write`, and `info` for the network and block device) and the other four are to get the time, poll for events, output to the console, and halt (see Table 1).

When compared to the Linux system call interface, the interface presented to the unikernel by the monitor is much thinner and is thus considered better for isolation [35]: for reference Linux has over 300 system calls, whereas Xen has about 20 hypercalls, and `ukvm` has 10 hypercalls.

We note that isolation does not come from the interface between the monitor and the host; indeed, the monitor is

[2] Additional discussion of trusting hardware appears in Section 5.1.3.
[3] Xen [16] is also a popular choice for unikernels, but we limit discussion to Linux/KVM for clarity.

[4] This is an upper bound, because `ukvm` is modular and can be tailored to a particular unikernel [47]. For example, unikernels that do not need a network device use a monitor that does not expose a network interface and therefore exposes fewer than 10 hypercalls.

RIGHTSLINK

a process on Linux, potentially with increased privilege to communicate with the KVM module. If an attacker compromised the monitor process, it would be able to launch attacks across the entire Linux system call interface.

2.3 Limitations of Hardware Virtualization

Despite being a convenient mechanism to construct a thin interface for isolation, using hardware extensions for virtualization has drawbacks including tooling, memory density, performance, and hardware availability.

As VMs became the dominant unit of execution on the cloud, common process-level tools, such as debugging tools, were used within the guest OS. With unikernels, there is no guest OS equipped with these familiar debugging tools [19]. While techniques traditionally used for kernel development are possible, such as connecting a client to a `gdb` stub implemented in the monitor, they are often cumbersome for application development.

VMs have also struggled with memory density because all guests typically perform file caching independently, even if many guests utilize the same (read-only) system files. To combat inefficient use of memory in VMs, techniques like memory ballooning [45] and kernel samepage merging (KSM) [8, 26] have been proposed that trade memory efficiency for extra CPU cycles and complexity.

On the performance side, every hypercall involves a world switch between the virtual CPU context and monitor context. We found that this more than doubles the number of cycles that a hypercall consumes when compared to a direct function call. We note that this overhead will occur on every hypercall. A detailed performance analysis appears in Section 5.

Finally, while the physical availability of virtualization hardware is no longer a concern in the modern data center, existing infrastructure-as-a-service clouds are often used as a base upon which to build new cloud offerings. For example, the OpenWhisk [7] serverless infrastructure is typically deployed on top of a collection of VMs. Unfortunately, nested virtualization is rarely enabled by cloud providers, leaving these VMs without hardware support for virtualization.

3 UNIKERNELS AS PROCESSES

When running on a unikernel monitor, unikernels appear much more similar to applications than kernels. A running unikernel is conceptually a single process that runs the same code for its lifetime, so there is no need for it to manage page tables after setup. Unikernels use cooperative scheduling and event loops with a blocking call like `poll` to the underlying system for asynchronous I/O, so they do not even need to install interrupt handlers (nor use interrupts for I/O).[5]

[5]Some unikernels, like OSv [27], do not (yet) run on a unikernel monitor and thus do not strictly fit this description. Determining whether such unikernels could be ported to unikernel monitors is a subject of future research.

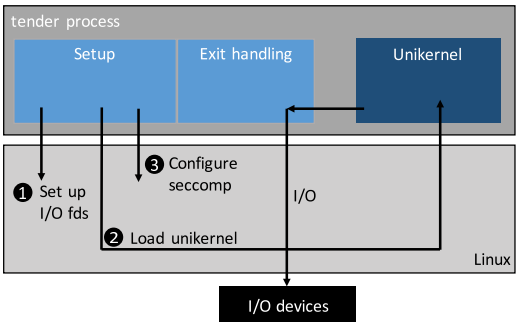

Figure 2: Unikernel isolation using a tender process with seccomp technology

In this section, we describe how, by leveraging existing system call whitelist mechanisms in the kernel, unikernels can maintain their isolation while running as processes, thereby avoiding the limitations of using virtualization hardware. We first provide background on these mechanisms, describe the unikernel-as-process architecture, then highlight how the unikernel changes to fit the process model.

3.1 Background: Linux and `seccomp`

Many modern operating systems have a notion of system call whitelisting allowing processes to transition into a mode with a more restricted system call interface available to them. For example, Linux has `seccomp` [24], OpenBSD has `pledge` [39], and FreeBSD has `capsicum` [46]. For clarity of exposition, we will focus this discussion on Linux and `seccomp`.

`seccomp` is a system call filtering framework that has been in Linux as of 2.6.12, since 2005 [24]. It was originally intended for CPU sharing [21], in which a process makes a one-way transition into *secure computing mode*. When a process is in secure computing mode, it can only perform a limited set of system calls on file descriptors that have already been opened. Furthermore, a process can register a custom BPF[38] program to specify filters that run on every system call.

A common problem with `seccomp` is that, in general, it is difficult to determine what system calls an application may use. An overly strict `seccomp` policy will result in unnecessary program termination. Thus, in practice, default `seccomp` policies tend to be large and overly permissive: Docker runs containers under a default policy that allows them to perform more than 250 system calls [11].

3.2 Unikernel/Process Architecture

Figure 2 shows an overview of how to run unikernels as processes, by replacing the monitor with a similar component we call a *tender*. Like the monitor in Figure 1, which relies on KVM and virtualization hardware, the tender process has two main tasks: setup and "exit" handling. In this case, the setup step begins the same: file descriptors are opened for I/O. However, instead of loading memory and register state

RIGHTSLINK

via the KVM module, the tender performs the following operations.

First, the tender dynamically loads the unikernel code into it's own address space, just like any other dynamically-loaded library. Note that the unikernel is self contained, except for explicit hypercalls. It does not reuse any other functions from the tender; for example, the unikernel has its own implementation of `libc` and/or other libraries. The unikernel will need heap space, so the tender allocates memory for the unikernel, according to the amount of memory requested.[6] There is no need to allocate register state or a temporary stack for the unikernel because it will inherit them from the tender. Starting execution of the unikernel is as simple as calling the entry function of the (loaded) unikernel code with boot time parameters (e.g., the size and location of memory) as arguments.

Once started, the unikernel executes as usual, but instead of invoking hypercalls when necessary, the unikernel simply does a normal procedure call to the hypercall implementation in the tender. The hypercall implementation in the tender is identical to the monitor implementation in the virtualization case; the tender will likely perform a system call to Linux, then return the result to the unikernel.

Importantly, before jumping to the entry point of the unikernel, the tender must configure `seccomp` filters to only allow system calls that correspond to unikernel exits for I/O and make the one-way transition.

3.3 Are Unikernels as Processes Isolated?

Unlike VMs, in which isolation stems from the interface between the unikernel and the monitor, when running unikernels as processes, isolation stems from the interface between the tender and the host. This is because the tender essentially *becomes* the unikernel when it first jumps to guest code. It is crucial that the tender ensures that the appropriate `seccomp` rules are in place (via a one-way transition) before becoming the unikernel. Unlike the uncertainty-ridden estimation-based approach of selecting an appropriate `seccomp` policy used by most applications, the tender can specify exactly which system calls should be allowed, based on the mapping from hypercalls to system calls, since `ukvm`-based unikernels only use those 10 hypercalls by design.

Table 1 shows the one-to-one mapping from hypercalls to system calls for `ukvm`-based unikernels. The hypercalls `blkinfo` and `netinfo` return information about devices that were setup at boot time and thus do not need to make system calls. Both `walltime` and `halt` are essentially directly passed through to system calls without the monitor imposing additional restrictions to arguments. The remaining calls have a specific resource associated with them, usually implicitly enforced by the monitor. For example, the monitor will ensure that a `netwrite` hypercall always results in a `write` system call to the file descriptor that it set up for the network device.

[6]The amount of memory is passed to the tender as a command-line parameter, just like the memory parameter for starting guest VMs.

Hypercall	System Call	Resource
`walltime`	`clock_gettime`	-
`puts`	`write`	*stdout*
`poll`	`ppoll`	*net_fd*
`blkinfo`	-	-
`blkwrite`	`pwrite64`	*blk_fd*
`blkread`	`pread64`	*blk_fd*
`netinfo`	-	-
`netwrite`	`write`	*net_fd*
`netread`	`read`	*net_fd*
`halt`	`exit_group`	-

Table 1: The entire set of `ukvm` hypercalls and the system call/resource pair that correspond to them.

Fortunately, `seccomp` filters allow the tender to specify such associations before it transitions to run the unikernel. For example, at setup time, after the tender knows what file descriptor corresponds to each device, it creates a `seccomp` filter that checks that the file descriptor argument to the system call matches the resources specified in Table 1. Explicit checks, like ensuring every `blkwrite` is of a length equal to some fixed sector size, can also be specified in the filter.

In every case, the checks that the monitor would implicitly or explicitly perform in the virtualization case are performed by the tender via its `seccomp` filters. We therefore consider unikernels as processes to exhibit an equal degree of isolation to unikernels as VMs. We provide further evaluation of isolation in Section 5.

3.4 Other Benefits

The host system provides many benefits to processes. Unikernels gain many of these benefits when run as processes. Recall that the tender loads a unikernel directly into its memory space as a dynamically loaded library. With no modifications, the host system can provide:

- **Address space layout randomization (ASLR).** ALSR is inherited from the dynamic loader; a unikernel (which is just a library) will be loaded at a different offset on each execution, making it difficult for an attacker to target known gadget addresses.
- **Support for common tooling.** When debugging, there is no need for running a `gdb` server in the monitor (as done in `ukvm` or `QEMU`), nor reloading the symbol tables after the unikernels are loaded into memory. The unikernel and the tender together make up a normal process albeit with some dynamic library loading. Most process-based tools, like `gdb`, `perf` and `valgrind` are able to handle dynamic libraries with no modifications.
- **Memory sharing.** If multiple copies of a unikernel are executed on the same machine, the tender can ensure that the memory containing the unikernel binary is automatically shared by the host (e.g., `mmap`ing

RIGHTSLINK

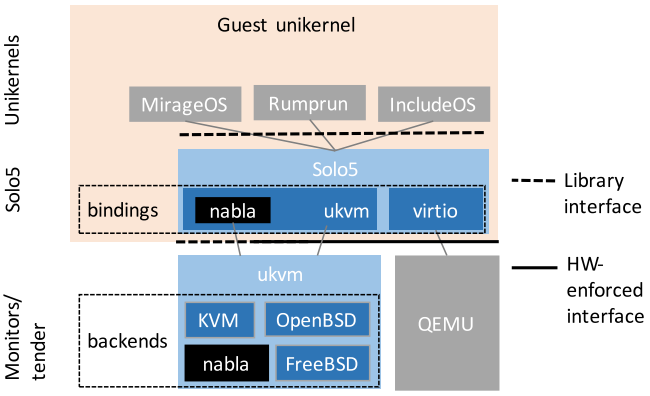

Figure 3: The Solo5 unikernel ecosystem. We implemented `nabla` as a new Solo5 binding and a new backend for the `ukvm` monitor.

the unikernel binary with `MAP_SHARED`), rather than resorting to tools such as KSM (kernel samepage merging) [8].

- **Architecture independence.** Processes do not need architecture-dependent code for bootup, configuring the MMU, or installing and handling interrupts, as is common in typical unikernels running as VMs. Like other applications, the problem is reduced to being able to compile the application code for a different architecture.

4 IMPLEMENTATION

We built a prototype system, called `nabla`,[7] to demonstrate unikernels running as processes. For the most part, `nabla` is a straightforward implementation of the tender design discussed in Section 3.2. In this section, we 1) outline the architecture, showing how our implementation fits with existing open-source components; 2) describe parts of the guest unikernel code (e.g., the libOS) that we modified, specifically thread-local storage; and 3) describe the tender's loading mechanism.

4.1 Architecture

We implemented `nabla` as part of the Solo5 unikernel ecosystem [12], as shown in Figure 3. Solo5 is a unikernel base layer, which provides an easy-to-port-to interface (see Table 1) upon which various unikernel library OSes run, including MirageOS, includeOS, and Rumprun unikernels. Solo5 contains *bindings* that allow it to run on different *monitors*, including `virtio`-based monitors like QEMU or the `ukvm` unikernel monitor. `ukvm` is essentially a direct "pass through" of the Solo5 interface upon which unikernels run, and, as such, it is distributed with Solo5. `ukvm` itself contains *backends* to run on different systems and architectures, from Linux/KVM to FreeBSD and OpenBSD, from x86 to ARM.

[7]The name `nabla` comes from the Greek symbol ∇, which evokes an image of a thin interface under the unikernel (to the system below).

We extended Solo5 and `ukvm` in two ways. First, we added a new Linux backend to the `ukvm` monitor, which enables it to function as a tender. Second, we modified the Solo5 `ukvm` binding to eliminate some hardware-specific setup and use a function call mechanism rather than a hypercall mechanism to access the tender. In all, our modifications resulted in 21 changed files with 1,586 additions and 64 deletions.

4.2 Thread-local storage

In Section 3, we claimed that unikernels are more like applications than kernels and that they do not need special handling (like *ring 0* or direct access to kernel-managed registers). However, in practice, we encountered one case where this was not true. Specifically, Rumprun unikernels [9] make use of segment-based thread local storage (TLS) as a convenient mechanism for threads to access local data. Unfortunately, on the AMD64 architecture, segment-based TLS requires full management of the *%fs* register. However, Linux processes need to make an `arch_prctl` system call in order to write into this register. Allowing unikernels as processes to perform the `arch_prctl` is undesirable because it violates our one-to-one mapping between `ukvm` hypercalls and system calls.

To solve this problem, we decided to eliminate segment-based TLS from Rumprun. The unmodified NetBSD code used in Rumprun already has support for multiple architectures, including ones without segment-based TLS; we simply directed the build to disable TLS. In the other parts of Rumprun, we manually replaced TLS accesses with explicit calls to access thread-specific variables, a total change of 10 files with 49 insertions and 125 deletions. In Section 5.2, we evaluate the performance implications of this implementation choice.

4.3 Dynamic unikernel loading

In our design, the `nabla` tender loads a unikernel directly into its memory space as a dynamically loaded library.[8] In practice, however, unikernels are built as static binaries (they are intended to run as VMs, after all). While it is possible to traverse the various unikernel build systems to ensure all code is compiled as position independent, this is cumbersome and tedious. We followed this approach for MirageOS and found that some libraries did not respect `CFLAGS` passed from the build system; we patched them accordingly. Though we didn't encounter any in MirageOS or its libraries, we are aware of some non-inline assembly code in Rumprun and would have had to refactor it to build Rumprun unikernels as position independent.

Instead, for our initial implementation, we sidestep the effort of building dynamic binaries by dynamically loading unikernels built as static binaries. All segments in the unikernel ELF marked as loadable (`PT_LOAD`) are memory mapped with an `mmap` fixed to the respective address. These

[8]Ideally, the loading and parsing of the ELF headers would be done entirely in userspace, after transitioning via seccomp, so that a maliciously crafted unikernel binary cannot attack the system before the transition occurs.

RIGHTSLINK

memory segments will be shared by all instances using the same unikernel binary (e.g., by all `node.js` unikernels). The unikernel expects memory to begin at address 0x0; if it is loaded at 0x0, all of the addresses contained in the (statically-built) unikernel remain valid. As a result, the monitor code must reside *above* the unikernel in memory. We specify a custom linker-script to ensure the monitor is loaded at 1GB.[9] Furthermore, we ensure the monitor code does not dynamically load any other libraries to avoid conflicts with the reserved unikernel. In practice, no unikernel pages are loaded into the 0x0 address of the process because the unikernels are linked with '.text' starting at 1 MB, leaving low memory empty.

We plan to modify the unikernels for dynamic loading to regain the ability to benefit from many process-like features that the dynamic loading case enjoys, including out-of-the-box ASLR, debugging support[10] and memory sharing. To date, we have modified the build process for only one class of unikernels (MirageOS) to produce position-independent unikernel binaries.

Name	Description
includeos-TCP	IncludeOS TCP server (receiver side)
includeos-UDP	IncludeOS UDP server (receiver side)
mirage-HTTP	MirageOS web server (1KB static pages)
nginx	`nginx` v1.8.0 (212B static pages)
nginx-large	`nginx` (6.4KB png images)
redis-get	`redis` v3.0.6 *get* tests of redis benchmark
redis-set	`redis` *set* tests of redis benchmark
node-express	Nodejs v4.3.0 (express v4.16.2 web server)
node-fib	Nodejs asynch fibonacci calculator (stresses green threads)
py-chameleon	Chameleon from the Python benchmark suite [13]
py-2to3	2to3 from the Python bench. suite (disk I/O intensive)
py-tornado	Tornado from the Python bench. suite (web server and client run in the same process/unikernel to stress localhost networking)

Table 2: Workloads used for the isolation and performance evaluations. Unless otherwise specified, the applications use Rumprun as their library OS.

5 EVALUATION

We evaluate unikernels as processes in two dimensions: isolation and performance. Throughout the evaluation, we use a variety of unikernels, shown in Table 2, in which the applications are built both using virtualization with `ukvm` (v0.2.2) and as processes with `nabla`. All applications use Rumpun unless otherwise specified. For comparison points in some experiments, we run the same application code as a native Linux process or as a VM using the QEMU monitor (v2.5.0) instead of `ukvm`.

For isolation, we find that even though there is indeed a one-to-one mapping between system calls permitted by `nabla` and the hypercall interface in `ukvm`, the kernel experiences one fewer system call when running `nabla` unikernels, surprisingly resulting in access to only half of the number of kernel functions as `ukvm` under various workloads. We also find that the `nabla seccomp` policy effectively reduces the amount of accessible kernel functions by 98% compared to a normal process.

For performance, we find that running unikernels as processes (`nabla`) improves performance over virtualization-based unikernels (`ukvm`) in all cases, up to over 245% in terms of throughput. Furthermore, compared to `ukvm`, `nabla` reduces CPU utilization by up to 12%, achieves 20% higher memory density, and reduces the already impressive startup times of `ukvm` by up to 73%.

5.1 Isolation Evaluation

As described in Section 2 and 3, isolation stems from a thin interface to the host. In this subsection, we quantify 1) how much of the host kernel a `nabla` unikernel (process) accesses as compared to a `ukvm` unikernel; and 2) how much of the host kernel a `nabla` unikernel could possibly access given the `nabla` isolation mechanism (its `seccomp` policy). Throughout this evaluation, a mechanism is more isolated than another if it has a thinner interface (e.g., uses less system calls) and consequently has access to less kernel code.

5.1.1 How much kernel access is needed? To measure which kernel functions were accessed by a unikernel, we use the Linux kernel's `ftrace` functionality [5]. `ftrace` is a kernel facility that can be configured to produce a trace containing a graph of kernel functions called by a specific process or set of processes (specified as a list of PIDs).

We are most interested in the kernel functions that the guest unikernel is able to access, not those that the monitor or tender accesses during setup time. Thus, rather than using an existing tool like `trace-cmd` to start and stop the tracing, we implemented a `ftrace` module for `ukvm`. The module forks a tracing process from the unikernel process and handles synchronization to ensure that the tracing is started and stopped at the right times without introducing extra system calls in the target. Once we obtain the function call graph from a unikernel execution, we filter out function call trees that are due to interrupts (`smp_irq_work_interrupt`, `smp_apic_timer_interrupt`, etc.) which may contain kernel work on behalf of other processes. The module works both for `ukvm` and `nabla` unikernels. It is implemented in approximately 400 lines of C code and is open source.[11]

For these experiments, in order to avoid contaminating the trace, we ran both the unikernel and its monitor/tender inside a Linux VM with a stock Ubuntu Linux kernel, version 4.10.0-38-generic. We interact with the test Linux VM

[9] Due to limitations with the loader (`ld`), we were unable to specify a higher loading address, so unikernels are limited to 1GB in size.

[10] While `gdb` can be used in our prototype, symbols for the unikernel are not automatically loaded as is the case for dynamic loading.

[11] https://github.com/djwillia/solo5/tree/ftrace

RIGHTSLINK

	ukvm	nabla	process
nginx	4	3	43
nginx-large	4	3	43
node-express	6	5	43
redis-get	4	3	36
redis-set	4	3	36

Table 3: Number of syscalls issued under various systems.

entirely through the serial console to avoid any tracing contamination from SSH. We bridge the Linux test VM's network device with the unikernel's tap interface. We note that nested virtualization is in use on the host to support the virtualization cases (`ukvm`) in the test VM.

For comparison, we also gather traces for the applications running as regular processes on Linux, as obtained by `trace-cmd` and similarly filtering out interrupts.

System calls. As a first analysis of the trace data, we calculate the number of system calls issued by the unikernel by extracting top-level calls prefixed with `sys` (ignoring case). Table 3 shows the results. For most of these applications, a `nabla` unikernel issues 3 calls: `ppoll`, `read` and `write`. For *node-express*, we see an additional two for block operations (`pread64` and `pwrite64`). In all cases, `ukvm` adds a single extra call: `ioctl` (`KVM_RUN`) is used to restart the guest after a hypercall. For comparison, `nabla` runs the applications using 8 to 14 times fewer system calls than native processes.

We also instrumented `ukvm` to compare with the hypercalls that are made in the `ukvm` case, as this is where `ukvm` unikernels derive their isolation. We confirm the one-to-one mapping between hypercalls and system calls, as described in Table 1 in Section 3.

Total coverage. We count the number of unique function calls in the traces and report them as the total number of kernel functions accessed. Figure 4 shows the results of tracing a variety of applications from Table 2 using this method. Each bar shows the mean of 5 trials and the error bars show the maximum and minimum number of functions observed.

Unlike the system call analysis, the difference in total kernel functions accessed is dramatic: in all cases `nabla` unikernels access about half of the number of functions accessed by `ukvm` unikernels. Further inspection shows much of this difference to be virtualization-related functions that `nabla` avoids, for example 43 with *kvm* in the function name, 20 with *vmx*, 15 with *vcpu*, etc. For reference, processes access about 5-6 times more kernel functions than `nabla` and 2-3 more times more than `ukvm` unikernels.

We also investigated functions that were in the `nabla` trace but not in the `ukvm` trace. Specifically, we found that `seccomp` filtering (for `nabla`) adds about five additional calls are made to perform more checks, indicating a relatively low-complexity (in terms of number of functions) implementation for `seccomp`.

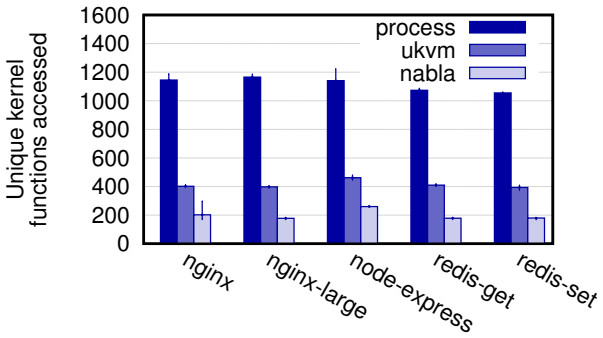

Figure 4: Unique kernel functions accessed when running applications as processes or `ukvm`/`nabla` unikernels.

5.1.2 How much of the kernel is accessible? Up to this point, we have been quantifying how much access to the underlying host kernel a unikernel needs to function correctly. However, to convince ourselves that a unikernel as a process is isolated, we would like to know not just how much of the kernel a unikernel *needs to access*, but how much of the kernel a unikernel *could possibly access* through its interface. This section describes experiments that employ fuzz testing to demonstrate how little of the host kernel a unikernel as a process can access.

We use `trinity`, a system call fuzzer, which is used to try and find unexpected bugs in the Linux kernel [14]. Rather than just calling completely random system calls with completely random arguments, `trinity` incorporates knowledge of the types of each system call argument. For example, `trinity` knows that a particular argument to a particular system call expects a file descriptor. In addition, `trinity` has a concept called an *fd provider* to further narrow down the search space. File descriptors are created in a particular way from one or more fd providers. Whenever `trinity` fuzzes a system call that expects a file descriptor, it obtains a valid file descriptor from the fd providers.

In order to use `trinity` to examine the effectiveness of `nabla`'s `seccomp` profile in isolating the unikernel as a process, we made three additions/modifications to it:

- A mechanism to start each system call fuzz test in its own subprocess. This is needed so that `nabla`'s `seccomp` rules can be added before issuing the syscall, without hindering the fuzzing framework from continuing to the next system call.
- A mechanism to enable and disable kernel tracing at the right times so that only the syscall under test is traced, without the fuzzing framework polluting the measurement. As before, we filter interrupts from the `ftrace` results.
- A new *nabla fd provider*, which opens file descriptors in the usual `nabla` way, then gives them to `trinity` when necessary.

Throughout all of the fuzz tests, `trinity` occasionally encounters errors or otherwise misbehaves. We automatically

RIGHTSLINK

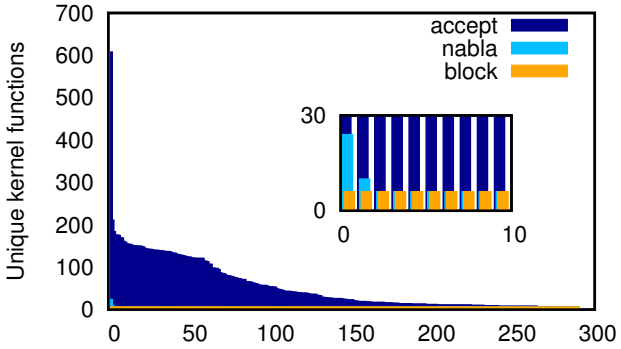

Figure 5: Histogram of kernel function coverage achieved through system call fuzzing with `trinity`.

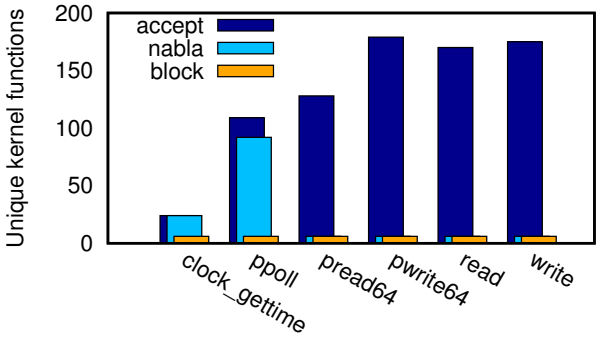

Figure 6: Coverage of the kernel achieved through targeted system call fuzzing of *nabla* system calls with `trinity`.

perform sanity-checks on the test runs to filter out empty traces and unexpected behavior. In practice, about 90% of the tests are usable, resulting in slightly under 14000 data points for each scenario in these experiments.

We examine three different `seccomp` policies: *accept*, which accepts everything; *nabla*, which corresponds to a policy that accepts only the system call and resource pairs that correspond to the `ukvm` hypercalls (shown in Table 1); and *block*, which blocks everything. In total, the three policies allow `trinity` to access 2682, 44, and 6 unique kernel functions respectively. The *nabla* policy reduces the amount of kernel functions accessible by 98%.

To visualize the data, Figure 5 shows a histogram with each system call on the x-axis with the total coverage achieved (in number of functions) on the y-axis. The largest (dark blue) region shows how much of the kernel trinity is able to access. Notably, some system calls result in many more kernel functions being accessed than others; these correspond to the largest potential attack surface so can be considered the most dangerous under this metric. For example, the `unshare` call shows up as the worst (touching 608 unique kernel functions) because it is the call that is responsible for the creation of all Linux *namespaces*, including the network namespace.

The lightest (orange) region in Figure 5 is barely visible as the lower bound for accessible kernel functions. It is fixed at 6 functions, which correspond to running the `seccomp` filter and encountering a denial. In the bottom left-hand corner, magnified in the graph insert, there are a few (light blue) bars that rise above the (orange) baseline. These correspond to the only system calls that are allowed by the *nabla* policy. To better examine these, we perform a second, more targeted experiment.

We configure `trinity` to only target the system calls permitted by the *nabla* policy, with the exception of `exit_group`. The system calls permitted by the *nabla* policy only allow for some specific arguments: like only permitting the disk *fd* and *count* equal to 512 (bytes) for the `pwrite64` system call. Figure 6 shows the number of kernel functions `trinity` was able to access when focusing all of its approximately 14000 tests on these calls. Some calls, like `clock_gettime` are fully permitted, so the *nabla* policy matches the *accept* policy. Others are much closer to the *block* policy. This is because the *nabla* policy does not fully open system calls. For example, although `pwrite64` is the sixth most dangerous system call under the metric discussed above, `trinity` was unable to find a set of arguments to the system call that would be permitted by the *nabla* policy yet also touch many kernel functions. Under the *accept* policy, when using all fd providers, `trinity` can touch much more of the kernel than the *nabla* policy will allow even through the same few system calls.

5.1.3 Limitations. The analysis of isolation presented here is based on an attack surface metric related to how much of the kernel code is accessed/accessible. Some researchers have proposed using more refined attack surface metrics. Kurmus et al. proposes using call graphs (obtained through static analysis) to count how many lines of code can be reached from the set of allowed syscalls [30]. The metric can be extended with weights given to each function; for example, more weight can be given to complex code (e.g., with the McCabe cyclomatic complexity measure [37]), code that has historically led to more vulnerabilities (i.e., CVEs), or based on a metric that distinguishes between popular and unpopular kernel code paths [32].

Since our attack surface metric is based on code coverage, it penalizes mechanisms that use additional code to improve the handling and accounting of data in the kernel. For example, Linux *namespaces* should improve these aspects but will show an additional "cost" in our metrics.

Another limitation in our analysis is that it only examines direct system calls. We filter out interrupts, which may provide more insight into the relative complexity of isolation mechanisms. Furthermore, other entry points into the kernel or kernel structures, for example through memory maps such as `vdso`, may not be captured by these metrics.

Finally, as discussed in Section 2, we trust the hardware. Recent hardware security vulnerabilities, like Spectre [28] and Meltdown [33], have demonstrated that this trust may

RIGHTSLINK

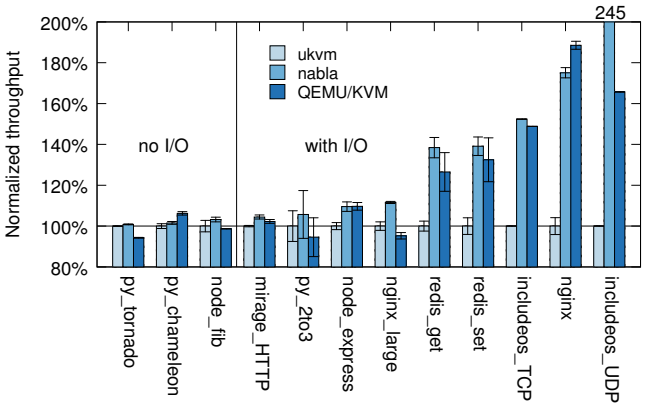

Figure 7: Normalized throughput for all applications in Table 2.

be misplaced. Our isolation analysis assumes that hardware abstractions are sound, so does not capture differences in hardware features that are intended to provide similar functionality. For example, we assume that standard page tables are an effective way to isolate memory, as are extended page tables (EPT). However, the meltdown attack successfully broke isolation in standard page tables but not EPT, demonstrating that perhaps hardware specifics should be considered when drawing conclusions about isolation.

5.2 Performance evaluation

We next investigate the performance implications of running unikernels as processes, both in terms of how the application implemented in the unikernel performs (e.g., throughput), as well as how it affects resource utilization in the data center (e.g., CPU utilization, memory utilization, and startup time).

Experimental setup. We ran our tests on two set of machines. The throughput experiments were performed on a pair of p50 Lenovo laptops with 4 i7-6820HQ CPU @ 2.70GHz cores (8 threads), 32GBs of RAM, a SATA 6.0Gb/s SSD, and both running Ubuntu 16.04.4 with kernel version 4.13.0-38-generic. The client/server experiments used these two p50s connected over 1Gbps Ethernet, except for *includeos-TCP* and *includeos-UDP* (Table 2), which we ran on a single machine to avoid the 1Gbps limit. The second set of machines, used for boot time measurements, are 8 Intel(R) Xeon(R) CPUs E5520 @ 2.27GHz cores (16 threads) running Red Hat Enterprise Linux 7 3.10.0-514.6.1.el7.x86_64 with 64GB of RAM. Once again, we use the workloads detailed in Table 2. The web based workloads were exercised with the `wrk` client using 100 concurrent connections and 8 threads (machine has 8 processors).

Throughput. Figure 7 shows the normalized maximum throughput for all the applications and workloads described above. `nabla` achieves higher throughput than `ukvm` in all cases, from 101% in *py-tornado* up to 245% in *includeos-TCP*. The workloads with the lowest performance improvements are the non-I/O based ones: *py-tornado*, *py-chameleon*, and *node-fib*. This can be explained by examining `ukvm`'s rate of `vmexit`s per second; without I/O, the only reasons for the VM to exit are to get the time or to sleep. Conversely, I/O-bounded workloads like *includeos-TCP* have the highest rate of `vmexit`s, and therefore get the biggest gains when running on `nabla`, as compared to `ukvm`. We will later see the benefits of avoiding `vmexit`s in terms of CPU utilization.

Workload	Seconds	Reduction
1T no TLS	9.10 ± 0.03	
1T TLS	7.49 ± 0.14	-21.49%
2T no TLS	15.84 ± 0.14	
2T TLS	14.59 ± 0.02	-8.46%

Table 4: Segment-based TLS increases the performance of a simple microbenchmark by 21.49% with a single thread (1T) and by 8.46% with two threads (2T).

The relative superiority of `nabla` for some benchmarks is misleading because it hides the baseline performance of `ukvm`, especially in terms of networking. `ukvm` makes a bold design choice: in order to maintain its extremely thin and simple interface, `ukvm` only allows a single packet to be sent per hypercall. To put the performance in context, we also run the unikernels on the QEMU/KVM monitor which uses a `virtio` I/O interface that allows for batching and elimination of memory copying. In most cases, `ukvm` performs within a few percentage points of QEMU/KVM, even exceeding QEMU/KVM for workloads with little or no I/O. However, the poor performance of `ukvm` for network I/O is evident on workloads with heavy I/O: for example, `nginx` on QEMU/KVM achieves 1.8× the throughput on `ukvm`. Despite this, even though it uses (and gets the isolation benefits of) the thin `ukvm` interface, `nabla` performs well by avoiding the expensive `vmexit` cost on every I/O operation. As evidence, `nabla` is within 8% of the QEMU/KVM baseline performance in all cases.

Effects of Thread Local Storage (TLS) on throughput. To evaluate the performance implications of eliminating segment-based TLS, we ran all macrobenchmarks listed in Table 2 on `ukvm` with and without TLS. Note that this is the original `ukvm` running on KVM (Linux version 4.15). In every case, we observed no statistically significant difference in performance. To further investigate the worst-case implications of avoiding segment-based TLS, we crafted a microbenchmark application: a C program on Rumprun as a `ukvm` unikernel which starts 1 or 2 `pthreads`, each executing a loop consisting of a thread-local variable increment followed by a *sched_yield* call.

RIGHTSLINK

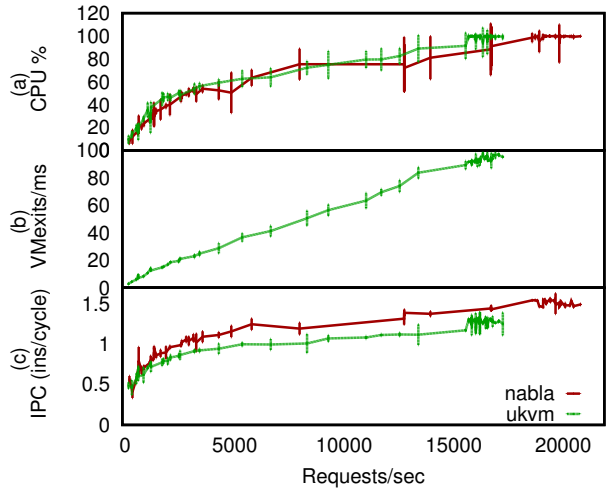

Figure 8: CPU utilization (single core) with an increasing number of concurrent clients.

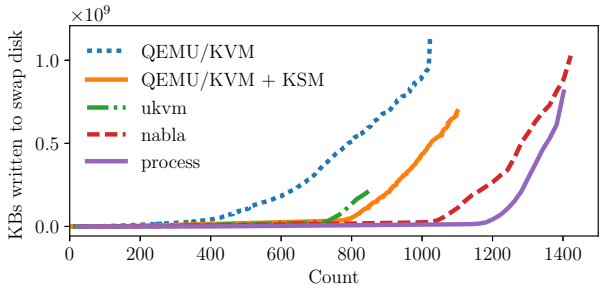

Figure 9: KBs written to swap and system CPU for an increasing number of *py-tornado* web servers. One way of reading the figure is: the system starts paging after about 900 `nabla` unikernel instances started running.

Table 4 (top two rows) shows the duration in seconds when running the program with 1 thread. It shows that segment-based TLS improves time from 9.10 to 7.49 (a 21.49% reduction). The no TLS case is expected to be slower as each access to the thread control block has to go through multiple steps of pointer indirection. With two threads, segment-based TLS decreases the time from 15.84 to 14.59 (8.46% decrease). The reduction is smaller when using 2 threads as the 1T case spends most of the time doing pointer indirection (the 2 threads experiment amortizes some of that time by switching threads).

We conclude that avoiding segment-based TLS when running unikernels as processes does not present performance issues for unikernel-native applications (e.g., MirageOS or IncludeOS-based) or for the selection of applications representative of cloud workloads we tested in Table 2, which make limited or no use of TLS on the critical path. For applications which make heavy use of TLS the performance degradation will be at most 21%.

CPU utilization. To further investigate the effect of `vmexit`s on performance, we study the CPU utilization of `ukvm`- and `nabla`-based unikernels under increasing load. We used *mirage-HTTP* (Table 2) at an increasing number of concurrent connections, increasing the load on the server. Figure 8(a) shows the CPU utilization as a percentage versus requests-per-second. The figure shows that `nabla` can reduce CPU utilization up to 12% (at 12K requests-per-second). This reduction is mostly due to the effect of `vmexit` rates (Figure 8(b)) on the actual *instructions executed per cycle* (IPC in Figure 8(c)). We measured IPC using the `perf-kvm` Linux tool. Between the cycle overhead (as discussed in Section 2.3) and the toll on CPU caches and TLBs, `vmexit`s reduce the speed at which instructions in the guest, monitor, and host kernel are executed (IPC in Figure 8(c)), leading to more cycles spent to achieve the same load and faster CPU saturation (at 17K requests-per-second).

Memory utilization. Memory density is a challenge for VMs; in this experiment we measure the memory density of `ukvm` unikernels vs `nabla` unikernels. To measure memory density, we started an increasing number of self-contained *py-tornado* unikernels (configured with 64MB of memory) and measured host swap activity. Each unikernel ran a client and server thread in which the client requested a page from the server every 1 second, infrequently enough not to become CPU bound. Unlike the other experiments, we configured the test machine with only 8GBs of RAM, as the CPU cores get bottlenecked before achieving 32GBs of total working set usage.

Figure 9 shows the results of the experiment described above. The important points to notice are the points of inflection: the count after which the load starts to rapidly increase. `ukvm` achieves 740 whereas `nabla` achieves 900 instances, about a 20% improvement in memory density. The main difference between the two is that `nabla` uses `mmap` during its setup phase to map the unikernel binaries, and therefore inherits sharing from the host kernel. When disabling `mmap`, `nabla` only achieves 760 instances (just 20 more than `ukvm`).

For reference, regular processes can scale up to 1050 instances. This can be explained by examining the working set sizes. The referenced pages over one minute (a measure of working set size) of `nabla` were 6168 KBs of anonymous memory, and 2840 KBs of shared file system pages (from the unikernel image)[12]. The regular process does not preallocate memory. It has a smaller working set, referencing 5396 KBs of anonymous memory and 2508 KBs of shared file system pages, because it also loads python modules as

[12]Referenced memory was measured using /proc/PID/clear_refs and /proc/PID/smaps. This technique is not applicable to KVM virtual machines and therefore we only show numbers for `nabla` and process

RIGHTSLINK

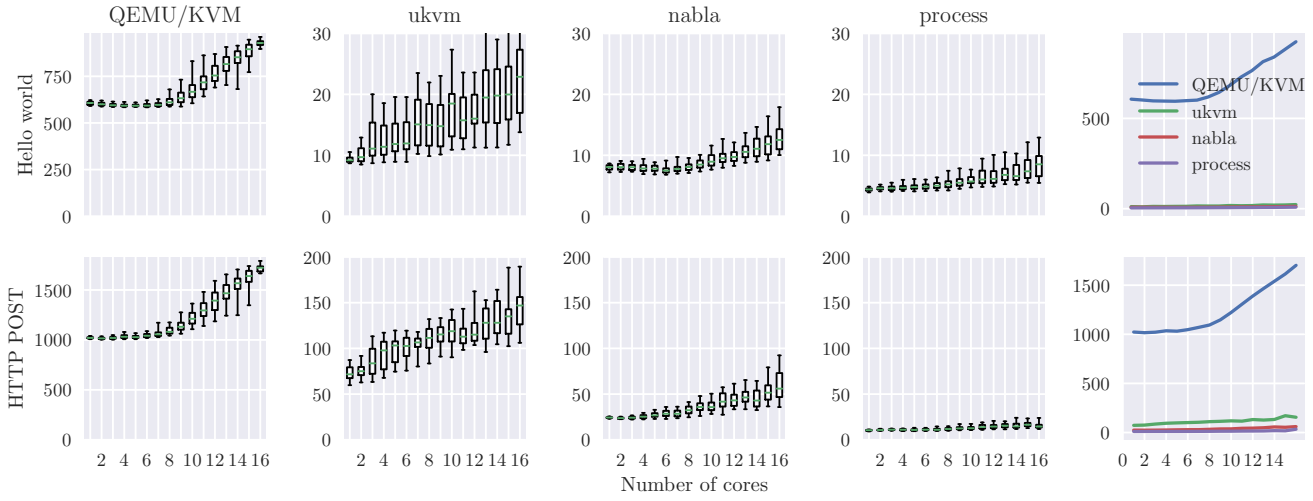

Figure 10: Startup and runtime in ms for an increasing number of cores running short instances sequentially (per core).

shared libraries and does not need the extra data pages referenced by the *nabla* tender and rumprun (e.g., for thread management).

Finally, we compare to unikernels using the QEMU monitor, which can only achieve 350 instances. QEMU/KVM does not achieve high density because there is no sharing between the pages. A further experiment using KSM (Kernel Samepage Merging) deduplication shows that QEMU/KVM can increase density to 750 instances, at the cost of increasing CPU usage by 10%.

Startup time. We next measured and compared `nabla` unikernel startup time. This metric is important for network function virtualization (NFV) [36] and serverless workloads in which a new unikernel instance is launched per request. We measured the startup time of short lived instances, sequentially starting, running, and exiting on an increasingly number of cores: no more than one instance per core. We used two MirageOS applications:[13] hello-world and HTTP-POST. The HTTP-POST application posts one HTTP request to a local multithreaded HTTP server.

Figure 10 shows the startup and runtime for these two actions. The HTTP-POST action has higher latency in all cases because it requires the creation of a network tap device in all configurations except as a process (where it uses the host's local interface). Latency is between 30% to 370% higher when using `ukvm` than `nabla`; this is due to the extra cost of starting a KVM VCPU context, compared to starting a process. Furthermore, increasing the number of cores increases startup time in all cases; but the effect is more pronounced for `ukvm` than `nabla` (2.4× vs. 1.88×) because of contention from the KVM module.

For reference, we also show results for starting a unikernel on a general-purpose monitor (QEMU/KVM) and as a native process (recall MirageOS can be configured to produce an OCaml binary rather than a unikernel [34]). The general-purpose monitor results in an unacceptable startup time of up to 1.6 seconds. The `nabla` unikernel is only as $1.5 - 3\times$ slower to start than a native process, compared to $2 - 10\times$ slower under `ukvm`.

6 RELATED WORK

Unikernels. Unikernel communities have emerged around many different languages and use-cases: MirageOS for OCaml [34], IncludeOS for C++ [18], LING for Erlang [3], HalVM for Haskell [42], runtime.js for Javascript [10], Clive for GO [2], and ClickOS for instances of the Click router [36]. More general-purpose unikernels include Rumprun [9] and OSv [27]. As of now, `nabla` has been tested on unikernels that run on `ukvm` (MirageOS, includeOS and Rumprun). Research into whether concepts like multi-cores (used by e.g., OSv) can be supported by `ukvm` is ongoing and we believe `nabla` will inherit any advances made by `ukvm` in this regard.

Running libOSes as processes. Drawbridge "picoprocesses" [40] are similar to unikernels as processes in that they use library OS techniques inside processes. Like `nabla`, they run atop a security monitor that restricts the available system calls. Unlike `nabla`, picoprocesses focus on running Microsoft Windows applications atop a slightly higher-level system call interface than `ukvm` consisting of approximately 35 calls. Most similarly, `frankenlibc` [4] contains tooling to execute rump kernels as processes and even contains a

[13] We also performed some startup measurements with Rumprun `node.js` and Python applications. For these, startup times for the runtimes (and `node.js` imports) alone range from 50ms to 200ms and obscure the differences between `nabla` and `ukvm`.

RIGHTSLINK

set of `seccomp` rules to lock down the system call interface. However, `frankenlibc` does not take advantage of the thin `ukvm` interface, resulting in a more permissive `seccomp` policy: 29 system calls are permitted (at least in part) as compared to the 7 system calls for `nabla`, including some of the most dangerous as determined by our `trinity`-derived metric (Section 5.1.2). The Chromium Native Client (NaCl) [50] ensures isolation at compilation time via software fault isolation. NaCl ensures memory is accessed within the allowed boundaries and only makes a small set of system calls for memory allocation and I/O on already opened sockets. ZeroVM [15] uses the Native Client (NaCl) as the sandboxing mechanism, and implements a libc and an in-memory file system. gVisor [6] is a recently-announced sandboxing approach which uses `ptrace` in order to trap system calls and redirect them to a secondary process which implements the libOS, which makes lower-level system calls to the host. Each approach differs in how system calls are intercepted and what is in the trusted computing base: `seccomp` in `nabla`'s case, or another sandboxing mechanism in gVisor or NaCl.

Other forms of isolation. We have been looking at achieving isolation by reducing the attack surface to the kernel. Other approaches toward isolation involve hardening monolithic kernels or hypervisors. The "Nested Kernel Architecture" [22] is a type of intra-kernel isolation where the kernel is divided in two fault isolated pieces: one with, and the other without read/write access to memory configuration structures. Other similar work split the kernel in different ways: Nooks [43] moves drivers code into less privileged hardware levels. Other systems split the kernel at compilation time using software fault isolation and control flow integrity [20, 25]. The "Split Kernel" [31] generates kernel images with two copies of each kernel function, one hardened with additional processing. Hypervisors have been hardened by reducing their size: KVM be moved to user-space, as in DeHype [49]. CloudVisor [51] reduces the TCB by adding a small nested hypervisor below an unmodified Xen hypervisor. Nexen [41] splits the hypervisor (Xen in this case) into two pieces, one with higher privileges and in charge of managing the MMU.

7 CONCLUSION

The term "unikernel" is in some sense a misnomer, as the fact that they contain kernel-like code is due to an unnecessary choice to run as a VM for isolation. In this paper, we have shown that running unikernels as processes can *improve* isolation over VMs, cutting their access to kernel functions in half. At the same time, running unikernels as processes moves them into the mainstream; unikernels as process inherit many process-specific characteristics—including high memory density, performance, startup time, etc.—and tooling that were previously thought to be necessary sacrifices for isolation. Going forward, we believe this work provides insight into how to more generally architect processes and containers to be isolated in the cloud.

REFERENCES

[1] AWS Lambda. https://aws.amazon.com/lambda/. (Accessed on 2018-08-28).
[2] Clive: Removing (most of) the software stack from the cloud. http://lsub.org/ls/clive.html. (Accessed on 2018-08-28).
[3] Erlang on Xen. http://erlangonxen.org. (Accessed on 2018-08-28).
[4] frankenlibc - tools for running rump unikernels in userspace. https://github.com/justincormack/frankenlibc. (Accessed on 2018-08-28).
[5] ftrace - Function Tracer. https://www.kernel.org/doc/Documentation/trace/ftrace.txt. (Accessed on 2018-08-28).
[6] gvisor - Container Runtime Sandbox. https://github.com/google/gvisor. (Accessed on 2018-08-28).
[7] IBM Cloud functions. https://www.ibm.com/cloud/functions. (Accessed on 2018-08-28).
[8] Kernel Samepage Merging. https://www.linux-kvm.org/page/KSM. (Accessed on 2018-08-28).
[9] The rumprun unikernel and toolchain for various platforms. http://repo.rumpkernel.org/rumprun. (Accessed on 2018-08-28).
[10] runtime.js - javascript library operating system for the cloud. http://runtimejs.org/. (Accessed on 2018-08-28).
[11] Seccomp security profiles for Docker. https://docs.docker.com/engine/security/seccomp/. (Accessed on 2018-08-28).
[12] Solo5 - A sandboxed execution environment for unikernels. https://github.com/solo5/solo5. (Accessed on 2018-08-28).
[13] The Python Performance Benchmark Suite. http://pyperformance.readthedocs.io/. (Accessed on 2018-08-28).
[14] Trinity - A Linux System call fuzz tester. http://codemonkey.org.uk/projects/trinity/. (Accessed on 2018-08-28).
[15] Zerovm. http://www.zerovm.org/. (Accessed on 2018-08-28).
[16] 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. In *Proc. of ACM SOSP* (Bolton Landing, NY, Oct. 2003).
[17] Ben-Yehuda, M., Day, M. D., Dubitzky, Z., Factor, M., Har'El, N., Gordon, A., Liguori, A., Wasserman, O., and Yassour, B.-A. The turtles project: Design and implementation of nested virtualization. In *Proc. of USENIX OSDI* (Vancouver, BC, Canada, Oct. 2010).
[18] Bratterud, A., Walla, A.-A., Haugerud, H., Engelstad, P. E., and Begnum, K. IncludeOS: A minimal, resource efficient unikernel for cloud services. In *Proc. of IEEE CLOUDCOM* (Vancouver, BC, Canada, Nov. 2015).
[19] Cantrill, B. Unikernels are unfit for production. https://www.joyent.com/blog/unikernels-are-unfit-for-production, Jan. 2016. (Accessed on 2018-08-28).
[20] Castro, M., Costa, M., Martin, J.-P., Peinado, M., Akritidis, P., Donnelly, A., Barham, P., and Black, R. Fast byte-granularity software fault isolation. In *Proc. of ACM SOSP* (Big Sky, MT, Oct. 2009).
[21] Corbet, J. Securely renting out your CPU with Linux. https://lwn.net/Articles/120647/, Jan. 2005. (Accessed on 2018-08-28).
[22] Dautenhahn, N., Kasampalis, T., Dietz, W., Criswell, J., and Adve, V. Nested kernel: An operating system architecture for intra-kernel privilege separation.
[23] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. *Microservices: Yesterday, Today, and Tomorrow.* Springer International Publishing, Cham, 2017, pp. 195–216.
[24] Edge, J. A seccomp overview. https://lwn.net/Articles/656307/, Sept. 2015. (Accessed on 2018-08-28).
[25] Erlingsson, Ú., Abadi, M., Vrable, M., Budiu, M., and Necula, G. C. XFI: Software guards for system address spaces. In *Proc. of USENIX OSDI* (Seattle, WA, Nov. 2006).
[26] 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. In *Proc. of USENIX OSDI* (San Diego, CA, Dec. 2008).

RIGHTSLINK

[27] Kivity, A., Laor, D., Costa, G., Enberg, P., HarEl, N., Marti, D., and Zolotarov, V. OSv optimizing the operating system for virtual machines. In *Proc. of USENIX Annual Technical Conf.* (Philadelphia, PA, June 2014).

[28] Kocher, P., Horn, J., Fogh, A., , Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. Spectre attacks: Exploiting speculative execution. In *Proc. of IEEE Security and Privacy* (San Francisco, CA, May 2019).

[29] Koller, R., and Williams, D. Will serverless end the dominance of linux in the cloud? In *Proc. of ACM/SIGOPS HotOS* (Whistler, BC, Canada, May 2017).

[30] Kurmus, A., Tartler, R., Dorneanu, D., Heinloth, B., Rothberg, V., Ruprecht, A., Schröder-Preikschat, W., Lohmann, D., and Kapitza, R. Attack surface metrics and automated compile-time os kernel tailoring. In *Proc. of Internet Society NDSS* (San Diego, CA, Feb. 2013).

[31] Kurmus, A., and Zippel, R. A tale of two kernels: Towards ending kernel hardening wars with split kernel. In *Proc. of ACM CCS* (Nov. 2014).

[32] Li, Y., Dolan-Gavitt, B., Weber, S., and Cappos, J. Lock-in-Pop: Securing privileged operating system kernels by keeping on the beaten path. In *Proc. of USENIX Annual Technical Conf.* (Santa Clara, CA, July 2017).

[33] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. Meltdown: Reading kernel memory from user space. In *Proc. of USENIX Security Symposium* (Baltimore, MD, Aug. 2018).

[34] Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., and Crowcroft, J. Unikernels: Library operating systems for the cloud. In *Proc. of ACM ASPLOS* (Houston, TX, Mar. 2013).

[35] 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. In *Proc. of ACM SOSP* (Shanghai, China, Oct. 2017).

[36] Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R., and Huici, F. ClickOS and the art of network function virtualization. In *Proc. of USENIX NSDI* (Seattle, WA, Apr. 2014).

[37] McCabe, T. J. A complexity measure. *IEEE Transactions on Software Engineering SE-2*, 4 (Dec 1976), 308–320.

[38] McCanne, S., and Jacobson, V. The BSD packet filter: A new architecture for user-level packet capture. In *Proc. of Winter USENIX Conference* (San Diego, CA, 1993).

[39] OpenBSD. `PLEDGE(2) - restrict system operations` OpenBSD man page.

[40] Porter, D. E., Boyd-Wickizer, S., Howell, J., Olinsky, R., and Hunt, G. C. Rethinking the library os from the top down. In *Proc. of ACM ASPLOS* (Newport Beach, CA, Mar. 2011).

[41] Shi, L., Wu, Y., Xia, Y., Dautenhahn, N., Chen, H., Zang, B., Guan, H., and Li, J. Deconstructing xen. In *Proc. of Internet Society NDSS* (San Diego, CA, Feb. 2017).

[42] Stengel, K., Schmaus, F., and Kapitza, R. Esseos: Haskell-based tailored services for the cloud. In *Proc. of ACM/IFIP/USENIX ARM* (Beijing, China, Dec. 2013).

[43] Swift, M. M., Bershad, B. N., and Levy, H. M. Improving the Reliability of Commodity Operating Systems. *ACM Transactions on Computer Systems 23*, 1 (Feb. 2005), 77–110.

[44] Thnes, J. Microservices. *IEEE Software 32*, 1 (Jan 2015), 116–116.

[45] Waldspurger, C. A. Memory resource management in VMware ESX server. In *Proc. of USENIX OSDI* (Boston, MA, Dec. 2002).

[46] Watson, R. N., Anderson, J., Laurie, B., and Kennaway, K. Capsicum: Practical capabilities for unix. In *Proc. of USENIX Security Symposium* (Washington, DC, Aug. 2010).

[47] Williams, D., and Koller, R. Unikernel monitors: Extending minimalism outside of the box. In *Proc. of USENIX HotCloud* (Denver, CO, June 2016).

[48] Williams, D., Koller, R., and Lum, B. Say goodbye to virtualization for a safter cloud. In *Proc. of USENIX HotCloud* (Boston, MA, July 2018).

[49] Wu, C., Wang, Z., and Jiang, X. Taming hosted hypervisors with (mostly) deprivileged execution. In *Proc. of Internet Society NDSS* (San Diego, CA, Feb. 2013).

[50] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. Native client: A sandbox for portable, untrusted x86 native code. In *Proc. of IEEE Security and Privacy* (Oakland, CA, May 2009).

[51] Zhang, F., Chen, J., Chen, H., and Zang, B. CloudVisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proc. of ACM SOSP* (Cascais, Portugal, Oct. 2011).

RIGHTSLINK