Unikernel Monitors: Extending Minimalism Outside of the Box

Dan Williams Ricardo Koller
IBM T.J. Watson Research Center

Abstract

Recently, unikernels have emerged as an exploration of minimalist software stacks to improve the security of applications in the cloud. In this paper, we propose extending the notion of minimalism beyond an individual virtual machine to include the underlying monitor and the interface it exposes. We propose *unikernel monitors*. Each unikernel is bundled with a tiny, specialized monitor that only contains what the unikernel needs both in terms of interface and implementation. Unikernel monitors improve isolation through minimal interfaces, reduce complexity, and boot unikernels quickly. Our initial prototype, `ukvm`, is less than 5% the code size of a traditional monitor, and boots MirageOS unikernels in as little as 10ms (8× faster than a traditional monitor).

1 Introduction

Minimal software stacks are changing the way we think about assembling applications for the cloud. A minimal amount of software implies a reduced attack surface and a better understanding of the system, leading to increased security. Even better, if the minimal amount of software necessary to run an application is calculated automatically, inevitable human errors (and laziness) when trying to follow best practices can be avoided. Recently this sort of automated, application-centered, dependency-based construction of minimal systems has been explored to what some believe is its fullest extent: unikernels [21] are stand-alone, minimal system images—built entirely from fine-grained modules that the application depends on—that run directly on virtual hardware.

Yet the exploration of minimal systems for the cloud via unikernels is only complete when viewed within a box: the box in this case being a virtual machine (VM). In this paper, we think outside the box and ask, in terms of the dependency-based construction of minimal systems, why stop at VM images? Is the interface between the application (unikernel) and the rest of the system, as defined by the virtual hardware abstraction, minimal? Can application dependencies be tracked *through* the interface and even define a minimal virtual machine monitor (or in this case a *unikernel monitor*) for the application, thus producing a maximally isolated, minimal execution unit for the application on the cloud? How would that work?

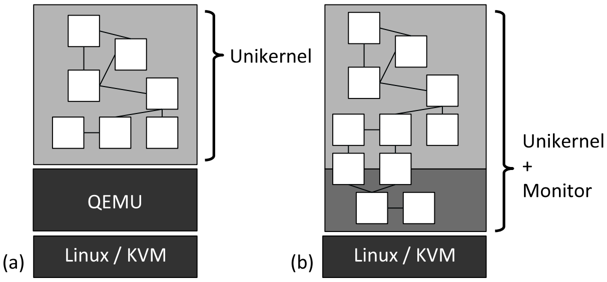

Figure 1: The unit of execution in the cloud as (a) a unikernel, built from only what it needs, running on a VM abstraction; or (b) a unikernel running on a specialized *unikernel monitor* implementing only what the unikernel needs.

As shown in Figure 1, we propose that executables for the cloud should contain both the application (e.g., a unikernel) and a *monitor*. The monitor is responsible both for efficiently launching the application in an isolated context and providing a specialized interface for the application to exit out of the context (e.g., for I/O), containing only what the application needs, no more, no less. The bundling of each application with its own custom monitor enables better isolation than either VMs or containers, with a simple, customized, high-performing interface. The ability of a unikernel monitor to boot unikernels quickly (as low as 10ms) makes them well suited for future cloud needs, including transient microservices [3, 5] and zero-footprint [4] operation.

In this position paper, we discuss how unikernel monitors could be automatically assembled from modules; specifically, how techniques used in package management to track application dependencies could extend through interface modules as well as monitor implementations. We also discuss the dangers and difficulties of running many different monitors in the cloud and how the small size of unikernel monitors (0.2% of a unikernel binary and 5% of the code base of traditional monitors like QEMU [11]) admits mitigation techniques like code analysis and certification. Finally, we discuss how our prototype implementation, `ukvm`, demonstrates the feasibility of unikernel monitors by efficiently booting MirageOS unikernels [21] with specialized interfaces.

2 Why Specialize the Monitor?

We argue that applications in the cloud should sit on top of specialized interfaces and the software layer underneath it, the *monitor*, should not be general-purpose. The desire to eliminate general-purpose OS abstractions is not new [13]. As such, there have been many approaches to specialize application software stacks for performance or isolation, from seminal library OS work [14, 19] to its more recent incarnation on the cloud under the unikernel moniker [22, 23, 29, 27, 12, 16, 8, 7]. Yet specializing the underlying monitor has been less studied.

The cloud suffers from unnecessary problems because applications use general-purpose monitors and interfaces. Current clouds try to fit all applications as VMs with the x86 interface, or as containers with the POSIX interface. Despite an extremely wide range of possible interface levels to explore, we argue that *any* general-purpose abstraction will suffer the same issues. More specifically, in this section, we describe how general purpose abstractions are not minimal, impose unnecessary complexity, and may introduce performance overheads.

Minimal Interfaces. In today's clouds, the interface to the rest of the system—whether full virtualization [11], paravirtualization [10], or OS-level (i.e., containers) [24]—is wide and general-purpose, including many unnecessary entry points into the monitor. Since each application has different requirements, *a general-purpose interface cannot be minimal.* For example, the virtual hardware abstraction exposed by KVM/QEMU is not minimal for an application because the VMM does not know whether a guest VM (application) will require a particular virtual device or interface. Exposing virtual device interfaces when they are not necessary can be disastrous for security, as demonstrated by the VENOM vulnerability in QEMU [9]. With VENOM, a bug in virtual floppy drive emulation code could be exploited to break out of the guest, regardless of whether a virtual floppy drive is instantiated.

A specialized monitor can expose a minimal interface, determined by what the application needs, resulting in fewer vulnerabilities available to exploit. A specialized monitor exposes an off-by-default interface. Rather than trying to block interface exit points via a blacklist-style policy (e.g., Default Allow in AppArmor [2]), exit points are explicitly introduced due to application needs, more like a whitelist.

In some cases, it may even be possible to eliminate seemingly-fundamental interfaces, like the network. Suppose a number of microservices in the cloud are intended to be chained together to implement a larger service. In today's clouds, each microservice would utilize the network to communicate. By specializing the monitor, network interfaces could be eliminated in favor of simpler serial input and output in a familiar pattern:

```
echo 1 | bundle1 | bundle2 | bundle3
```

Even in the case of compromise, each microservice would not have a network device available to use for communication with the outside world.

Simplicity. Regardless of the width or the level of the interface, general-purpose monitors adhere to a general-purpose interface. Any implementation in the monitor (underneath the interface) must be general enough to work for the full range of applications above, thereby introducing complexity. Simplicity is somehow related to the choice of interface level: any functionality implemented underneath the interface (in the monitor) must pay a "generality tax". For example, for an interface at the TCP level, the monitor must manage multiple tenants and resource sharing in the network stack. At the packet level, the monitor must only multiplex a NIC. In general, a lower-level interface needs to pay less "generality tax". However, even at the low layer, general-purpose monitors are still complex. Virtual hardware devices adhere to legacy standards (BIOS, PCI devices, DMA address restrictions, memory holes, etc.) so that general-purpose guests can operate them.

Specialized monitors, on the other hand, create opportunities to simplify both the guest and the monitor. Legacy standards are unnecessary for most applications in the cloud. For example, both the `virtio` [25] front-end (in the guest) and back-end (in the monitor) can be completely removed in lieu of simpler, direct packet-sending interfaces. Furthermore, with a specialized monitor, complex VM introspection techniques [15], which are brittle and suffer from inconsistencies and synchronization issues [28], can be replaced by introducing interfaces to facilitate introspection techniques and deal with

```
/* UKVM_PORT_NETWRITE */
struct ukvm_netwrite {
    void *data; /* IN */
    int len;    /* IN */
    int ret;    /* OUT */
};
```

Figure 2: An example interface to send a network packet.

synchronization issues. Finally, specialized interfaces to integrate with software written for general-purpose operating systems [26] could simplify certain applications and their development.

It may be still advised to implement low-level interfaces rather than high-level interfaces in specialized monitors for security reasons (see Section 3), but specialized monitors do not incur a "generality tax".

Faster Boot Time. Boot time is especially important for emerging application domains including the Internet of Things (IoT) [20], network function virtualization (NFV) [22], and event triggered, subsecond-metered services like Amazon Lambda [3]. In such environments, cloud-based services are expected to be created on the fly and then destroyed after they have performed their function.[1] As described above, guests running on general-purpose monitors often perform cumbersome virtual hardware negotiation and emulation, which—in addition to adding complexity—also increases boot time (e.g., to enumerate the virtual PCI bus). Efforts to improve the boot time on general-purpose monitors [1] will eventually hit a limit where any further specialization of the monitor and guest to eliminate common discovery and negotiation may diminish the set of guests supported by the monitor. Such specialization is unacceptable for today's cloud, where there is one monitor that must support all guest workloads.

In situations where further specialization is acceptable—including the bundling of application-specific monitors with the applications themselves as we suggest—better performance has been demonstrated. For example, unikernels like ClickOS [22] and MirageOS [21] with Jitsu [20] have been shown to boot in as low as 20ms on modified (specialized) VMM toolstacks.

3 Unikernel Monitors

We propose that each unikernel be distributed with its own specialized monitor. This monitor should have two tasks: 1) creating an isolated context to run the unikernel, and 2) taking action whenever the unikernel exits the isolated context. The monitor thereby maintains complete control over the unikernel. One of the actions the monitor may take is to destroy the unikernel.

A straightforward implementation of a unikernel monitor is as a specialized virtual machine monitor. In this case, hardware protection provides an isolated context, using hardware support for virtualization. If the unikernel exits its context for any reason (e.g., an I/O port operation, an illegal instruction, etc.) the hardware will trap into the monitor.

The default behavior for a monitor is to maintain complete isolation for the unikernel. A completely self-contained unikernel is bundled with an extremely simple monitor. The monitor simply sets up the hardware-isolated context and runs the unikernel. It does not expose any interfaces to the unikernel: every unikernel exit results in the monitor immediately destroying the unikernel and reclaiming its resources. At this time, since the monitor is specialized for the (now destroyed) unikernel, the monitor no longer has work to do and can safely exit.

Of course, a unikernel that runs in complete isolation may not be terribly useful for the cloud. Interfaces between the unikernel and monitor are provided on a per-application basis and do not need to adhere to established standards. Interfaces can exploit the fact that the monitor is able to access the memory contents of the unikernel. For instance, Figure 2 shows an example interface to send a network packet. By writing the address of an instance of this structure to the I/O port defined by UKVM_PORT_NETWRITE, a unikernel will exit to the monitor. The monitor directly accesses the network packet in the unikernel's memory at the specified memory location, checks or sanitizes the packet, and then sends the packet to the physical network.

Building Monitors. In theory, a unikernel strives to be a single application assembled with a minimal amount of software to allow it to run. Simply running a library operating system is insufficient for minimalism. In addition, only the functions needed by the application should be included in the library OS for any specific unikernel. Some unikernel approaches apply a clever use of package management and dependency tracking to approximate a minimal build.

For example, MirageOS [21], which produces OCaml-based unikernels, leverages the OCaml package manager, OPAM, to track dependencies between components of their library OS. As depicted in Figure 3(a), even modules that would typically be included by default in a monolithic OS, such as the TCP stack, are packages with tracked dependencies. In this example, the application requires TCP, so at compile time, the toolchain selects both TCP and a network interface driver to interface with the virtual NIC exposed by the VMM. Since the application does not use a filesystem, the toolchain ex-

[1] This is sometimes called a *zero-footprint cloud*. [4]

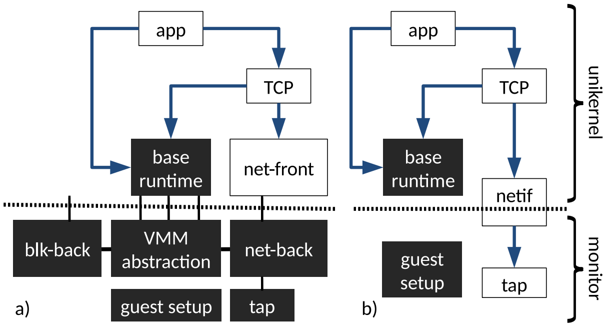

Figure 3: Application dependencies determine software that is assembled into (a) a standard unikernel; or (b) a unikernel and monitor bundle. Dark boxes are included by default.

cludes filesystem modules and block device driver modules from the build. It is important to note that the back-end virtual devices and their interfaces may still be present in the overall system regardless of whether the application needs them. In Figure 3, dark gray boxes are included by default, whereas white boxes are selected based on application dependencies.

We propose extending the dependency-tracking notion through the monitor interface, more specifically by modifying the toolchain and the package manager. Figure 3(b) shows the same application that depends on TCP. At build time, the modified toolchain selects TCP and a network interface driver. Unlike the standard unikernel in Figure 3(a), the network interface spans the unikernel and monitor; it is not written assuming a generic virtual network device implementation such as `virtio` [25]. Furthermore, the network interface module carries an explicit dependency on the backend network implementation via a TAP device [18]. In this case, the toolchain not only excludes filesystem modules and device driver modules from the unikernel, but from the monitor as well. If the application did not have a dependency chain to the network tap device, the toolchain would have excluded the tap driver, the interface, and the TCP module from the unikernel and monitor. The only default component in the monitor, *guest setup*, is the component that is responsible for booting the unikernel (and destroying it on any unhandled exit).

To realize such a system, there are many interesting issues to solve around how to specify or encode packages, especially those that span the interface, what granularity packages should be, and how to automatically build an entire unikernel monitor from such packages.

Securing the Monitors Unlike traditional virtual machine monitors in the cloud, there is not a single unikernel monitor for the cloud. From a cloud operation perspective, this implies that the cloud must evolve to support multiple monitors, a potentially different one for each unikernel. At first glance, this is a daunting proposition: it's hard enough to maintain a single virtual machine monitor in production, it would be near impossible to maintain a boundless number!

However, after further reflection, we believe that monitors are small enough to be bundled with unikernels and safely run on the ubiquitous Linux KVM system. Implementation-wise, the unikernel monitor can be similar to a type-II hypervisor: essentially a userspace program that leverages the host for most resource management. For example, a unikernel monitor that occupies the same place in the stack as QEMU in a standard Linux KVM/QEMU system can run on any Linux host with the KVM module. As we describe in Section 4, our prototype is indeed similar to a type-II hypervisor.

It should be noted that, in this circumstance, the monitor will execute in the isolation context of a normal user process, which may not be secure enough for multi-tenant clouds. Given the fact that the interface between the monitor and the unikernel is customized, the less-than-ideally-isolated monitor appears to be a straightforward channel for a unikernel to bypass its hardware-based isolation. We believe that well-defined interfaces and a modular, minimal approach to monitor construction will help assuage these fears. Unikernels are already touted to be small, but the overall size of the monitor is but a fraction of the size of the unikernel, making them amenable to formal verification [17] or audit. For example, our prototype monitor is just over 1000 lines of code, with a binary just .02% of a MirageOS-based static Web server unikernel binary. A cloud provider could mandate that each monitor be built from a set of certified modules.

4 A Prototype: `ukvm`

In order to show the feasibility of this new unit of execution on the cloud, we now describe a prototype implementation of a unikernel monitor called `ukvm`. The source code is freely available [6]. `ukvm` boots and acts as a monitor for a unikernel based on Solo5 [6], a thin open-source unikernel base, written in C, that (among other things) supports the MirageOS [21] runtime and components. A Mirage application binary (compiled from OCaml code) is statically linked to the Solo5 kernel.

`ukvm` is a specialized monitor for a Solo5-based unikernel. Architecturally, `ukvm` is a replacement for QEMU (specifically the user level side of a KVM/QEMU system). It is a user level program that loads a kernel ELF executable ($solo5 + mirage$), creates a KVM VCPU, and configures memory and registers so the Solo5 kernel can start in 64-bit privileged mode as

		QEMU	ukvm
Solo5 Kernel	malloc	6282	6282
	runtime	2689	2272
	virtio	727	-
	loader	886	-
	total	*10484*	*8552*
Monitor	QEMU	25003	-
	ukvm	-	990 (+ 172 tap)
	total	*25003*	*1162*

Table 1: Lines of code for the kernel and the monitor for the general-purpose QEMU, and the specialized ukvm.

a regular C *main*(). The memory and register setup includes setting a linear page table (a unikernel has a single address space), a stack, and loading registers with some arguments for the kernel (like the memory size).

The I/O interfaces between ukvm and Solo5 look like the one in Figure 2. They provide zero-copy IO by allowing any address of memory to be used as a buffer (of any size), and making the call with no more than a single VM exit (no need to probe if the PCI bus is ready, as would be done with virtio). We implemented basic disk and network backends in ukvm by using TAP [18] and host file reads and writes.

Table 4 shows the lines of code needed for implementing Solo5 on top of QEMU versus ukvm. Most of the reduction in Solo5 comes by removing virtio and the loader. Also, notice how an application configured not to use the network would have 10% less code in ukvm. For a concrete estimate of the size of the monitor in relation to the unikernel, the ukvm binary is 23KB compared to the 11MB Solo5 executable image when linked against the *www* Mirage application (only 0.2%).

Our prototype implementation does not automatically select the minimal configuration needed to run; automatic selection is limited to the MirageOS components.

Boot Time. We measured boot time for ukvm and compared it against traditional virtualization approaches like QEMU, and to the more recent lkvm (used by kvmtool in clear containers [1]). QEMU exposes a physical machine abstraction and lkvm is a more lightweight monitor that skips the BIOS and bootloader phase and jumps directly to the 64-bit kernel. lkvm and QEMU were configured to use virtio network and block devices. The three monitors were configured to use 512 MB of memory, and one 2.90GHz CPU core. Furthermore, the monitors were instrumented to trace the first VM instruction, the first serial output, the first network output, and the final halt instruction.

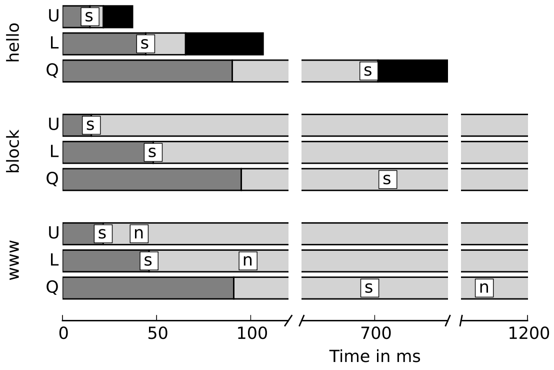

Figure 4: Boot times for ukvm (U), lkvm (L), and QEMU (Q) for some applications. 's' and 'n' indicate the first serial and network output, respectively.

Figure 4 shows the boot times for QEMU, lkvm, and ukvm for 3 MirageOS applications: *hello* (prints "hello" to the console then exits), *block* (tests disk reads and writes), and *www* (serves static Web pages).

The gray bars on the left show the time spent on monitor initialization. As expected, QEMU takes the longest, with 80ms compared to lkvm that takes an average of 45ms, and ukvm with 10ms. ukvm and lkvm load the 64-bit kernel immediately, so the kernel produces its first serial output (the 's') quicker than QEMU, which unpacks an ISO file in real mode to load the kernel. The *www* bars show that ukvm is able to do real work as soon as the kernel starts as the kernel sends its first network packet (the 'n') 18 milliseconds after its first serial output. lkvm and QEMU, on the other hand, first discover these devices, then initialize them before sending, resulting in at least 100ms delay.

5 Conclusion

We propose a new unit of execution for the cloud, built from the bundling of unikernels and specialized *unikernel monitors*. As a first step, with our prototype monitor, ukvm, we have shown that such monitors can be small and simple, yet powerful enough to run real unikernels. We believe the advantages of specializing cloud software stacks—including the monitor—are key to realizing the security and responsiveness needs of future clouds.

References

[1] An introduction to Clear Containers. https://lwn.net/Articles/644675/. (Accessed on 2016-03-06).

[2] AppArmor. http://wiki.apparmor.net/index.php/Main_Page. (Accessed on 2016-03-04).

[3] AWS Lambda. https://aws.amazon.com/lambda/. (Accessed on 2016-03-04).

[4] Erlang on Xen. http://erlangonxen.org/case/a-zero-footprint-cloud. (Accessed on 2016-03-04).

[5] IBM OpenWhisk. https://developer.ibm.com/open/openwhisk/. (Accessed on 2016-03-04).

[6] The Solo5 Unikernel. https://github.com/djwillia/solo5. (Accessed on 2016-05-10).

[7] Javascript library operating system for the cloud. http://runtimejs.org/, Apr. 2015.

[8] The rumprun unikernel and toolchain for various platforms. http://github.com/rumpkernel/rumprun, Apr. 2015.

[9] The VENOM vulnerability. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3456, 2015.

[10] 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).

[11] BELLARD, F. QEMU, a fast and portable dynamic translator. In *Proc. of USENIX Annual Technical Conf. (FREENIX Track)* (Anaheim, CA, Apr. 2005).

[12] BRATTERUD, A., WALLA, A.-A., HAUGERUD, H., ENGELSTAD, P. E., AND BEGNUM, K. Includeos: A minimal, resource efficient unikernel for cloud services.

[13] ENGLER, D. R., AND KAASHOEK, M. F. Exterminate all operating system abstractions. In *Proc. of USENIX HotOS* (Orcas Island, WA, May 1995).

[14] ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE, J. W. Exokernel: An operating system architecture for application-level resource management. In *Proc. of ACM SOSP* (Copper Mountain, CO, Dec. 1995).

[15] GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., AND BONEH, D. Terra: A virtual machine-based platform for trusted computing. In *ACM SIGOPS Operating Systems Review* (2003), vol. 37, ACM, pp. 193–206.

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

[17] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. seL4: Formal verification of an OS kernel. In *Proc. of ACM SOSP* (Big Sky, MT, Oct. 2009).

[18] KRASNYANSKY, M. Universal TUN/TAP device driver, 1999.

[19] KRIEGER, O., AUSLANDER, M., ROSENBURG, B., WISNIEWSKI, R. W., XENIDIS, J., DA SILVA, D., OSTROWSKI, M., APPAVOO, J., BUTRICO, M., MERGEN, M., ET AL. K42: building a complete operating system. In *ACM SIGOPS Operating Systems Review* (2006), vol. 40, ACM, pp. 133–145.

[20] MADHAVAPEDDY, A., LEONARD, T., SKJEGSTAD, M., GAZAGNAIRE, T., SHEETS, D., SCOTT, D., MORTIER, R., CHAUDHRY, A., SINGH, B., LUDLAM, J., CROWCROFT, J., AND LESLIE, I. Jitsu: Just-in-time summoning of unikernels. In *Proc. of USENIX NSDI* (Oakland, CA, May 2015).

[21] 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).

[22] 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).

[23] PORTER, D. E., BOYD-WICKIZER, S., HOWELL, J., OLINSKY, R., AND HUNT, G. C. Rethinking the library os from the top down. *ACM SIGPLAN Notices 46*, 3 (2011), 291–304.

[24] PRICE, D., AND TUCKER, A. Solaris Zones: Operating system support for consolidating commercial workloads. In *Proc. of USENIX LISA* (Atlanta, GA, Nov. 2004).

[25] RUSSELL, R. virtio: Towards a de-facto standard for virtual I/O devices. *SIGOPS OSR 42*, 5 (2008), 95–103.

[26] SCHATZBERG, D., CADDEN, J., KRIEGER, O., AND APPAVOO, J. A way forward: Enabling operating system innovation in the cloud. In *Proc. of USENIX HotCloud* (Philadelphia, PA, June 2014).

[27] STENGEL, K., SCHMAUS, F., AND KAPITZA, R. Esseos: Haskell-based tailored services for the cloud. In *Proceedings of the 12th International Workshop on Adaptive and Reflective Middleware* (New York, NY, USA, 2013), ARM '13, ACM, pp. 4:1–4:6.

[28] SUNEJA, S., ISCI, C., DE LARA, E., AND BALA, V. Exploring vm introspection: Techniques and trade-offs. *SIGPLAN Not. 50*, 7 (Mar. 2015), 133–146.

[29] 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.