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?

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, uikvm, 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 whitelists.

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
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 isolation for 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-

---

1This is sometimes called a zero-footprint cloud. [4]
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 \texttt{ukvm}. The source code is freely available \cite{ukvm}. \texttt{ukvm} boots and acts as a monitor for a unikernel based on Solo5 \cite{Solo5}, a thin opensource unikernel base, written in C, that (among other things) supports the MirageOS \cite{MirageOS} runtime and components. A Mirage application binary (compiled from OCaml code) is statically linked to the Solo5 kernel.

\texttt{ukvm} is a specialized monitor for a Solo5-based unikernel. Architecturally, \texttt{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 (\texttt{solos5 + mirage}), creates a KVM VCPU, and configures memory and registers so the Solo5 kernel can start in 64-bit privileged mode as if it were a real guest setup.
Table 1: Lines of code for the kernel and the monitor for the general-purpose QEMU, and the specialized ukvm.

<table>
<thead>
<tr>
<th>Solo5 Kernel</th>
<th>QEMU</th>
<th>ukvm</th>
</tr>
</thead>
<tbody>
<tr>
<td>malloc</td>
<td>6282</td>
<td>6282</td>
</tr>
<tr>
<td>runtime</td>
<td>2689</td>
<td>2272</td>
</tr>
<tr>
<td>virtio</td>
<td>727</td>
<td>-</td>
</tr>
<tr>
<td>loader</td>
<td>886</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>10484</td>
<td>8552</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Monitor</th>
<th>QEMU</th>
<th>ukvm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>25003</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>990  (+ 172 tap)</td>
</tr>
<tr>
<td>total</td>
<td>25003</td>
<td>1162</td>
</tr>
</tbody>
</table>

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.

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