Unikernels as Processes

Dan Williams, Ricardo Koller (IBM Research)
Martin Lucina (robur.io/Center for the Cultivation of Technology)
Nikhil Prakash (BITS Pilani)
What is a unikernel?

• An application linked with **library OS** components
• Run on **virtual hardware** (like) abstraction

• Language-specific
  • MirageOS (OCaml)
  • IncludeOS (C++)

• Legacy-oriented
  • Rumprun (NetBSD-based)
    • Can run nginx, redis, node.js, python, etc..
Why unikernels?

• **Lightweight**
  - Only what the application needs

• **Isolated**
  - VM-isolation is the “gold standard”

• Well suited for the **cloud**
  - Microservices
  - Serverless
  - NFV
Virtualization is a mixed bag

• Good for isolation, but...

• **Tooling** for VMs not designed for lightweight (e.g., lightVM)
• How do you debug **black-box** VMs?
• Poor VM **performance** due to vmexits
• **Deployment** issues on already-virtualized infrastructure
Why not run unikernels as processes?

• Unikernels are a single process anyway!

• Many benefits as a process
  • Better performance
  • Common tooling (gdb, perf, etc.)
  • ASLR
  • Memory sharing
  • Architecture independence

• Isolation by limiting process interface to host
  • 98% reduction in accessible kernel functions
Outline

• Introduction

• Where does unikernel isolation come from?
• Unikernels as processes
• Isolation evaluation
• Performance evaluation
• Summary
Isolation: definitions and assumptions

• **Isolation**: no cloud user can read/write state or modify its execution

• Focus on **software deficiencies** in the host
  • Code reachable through interface is a metric for attack surface

• We **trust HW** isolation (page tables, etc.)

• We do not consider covert channels, timing channels or resource starvation
Unikernel architecture

- ukvm unikernel monitor
  - Userspace process
  - Uses Linux/KVM

- Setup and loading
- Exit handling
Unikernel architecture

- **ukvm unikernel monitor**
  - Userspace process
  - Uses Linux/KVM
- Setup and loading
- Exit handling
Unikernel architecture

• ukvm unikernel monitor
  • Userspace process
  • Uses Linux/KVM

• Setup and loading
• Exit handling
Unikernel architecture

- ukvm unikernel monitor
  - Userspace process
  - Uses Linux/KVM

- Setup and loading
- Exit handling

Diagram:

- Monitor process (e.g., ukvm)
- Setup
- Exit handling
- Load unikernel
- I/O
- KVM
- Internet
- I/O devices
- VT-x
Unikernel isolation comes from the interface

- 10 hypercalls
- 6 for I/O
  - Network: packet level
  - Storage: block level
- vs. >350 syscalls

<table>
<thead>
<tr>
<th>Hypercall</th>
</tr>
</thead>
<tbody>
<tr>
<td>walltime</td>
</tr>
<tr>
<td>puts</td>
</tr>
<tr>
<td>poll</td>
</tr>
<tr>
<td>blkinfo</td>
</tr>
<tr>
<td>blkwrite</td>
</tr>
<tr>
<td>blkread</td>
</tr>
<tr>
<td>netinfo</td>
</tr>
<tr>
<td>netwrite</td>
</tr>
<tr>
<td>netread</td>
</tr>
<tr>
<td>halt</td>
</tr>
</tbody>
</table>
Observations

• Unikernels are not kernels!
  • No page table management after setup
  • No interrupt handlers: cooperative scheduling and poll

• The ukvm monitor doesn’t “do” anything!
  • One-to-one mapping between hypercalls and system calls

• Idea: maintain isolation by limiting syscalls available to process
Unikernel as process architecture

• **Tender**: modified ukvm unikernel monitor
  - Userspace process
  - Uses `seccomp` to restrict interface

• Setup and loading
Unikernel as process architecture

- **Tender**: modified ukvm unikernel monitor
  - Userspace process
  - Uses `seccomp` to restrict interface

- Setup and loading
Unikernel as process architecture

- **Tender**: modified ukvm unikernel monitor
  - Userspace process
  - Uses **seccomp** to restrict interface

- Setup and loading
Unikernel as process architecture

- **Tender**: modified ukvm unikernel monitor
  - Userspace process
  - Uses **seccomp** to restrict interface

- Setup and loading
Unikernel as process architecture

- **Tender**: modified ukvm unikernel monitor
  - Userspace process
  - Uses `seccomp` to restrict interface

- Setup and loading
- “Exit” handling
Unikernel isolation comes from the interface

• 10 hypercalls

• 6 for I/O
  • Network: packet level
  • Storage: block level

• vs. >350 syscalls

<table>
<thead>
<tr>
<th>Hypercall</th>
</tr>
</thead>
<tbody>
<tr>
<td>walltime</td>
</tr>
<tr>
<td>puts</td>
</tr>
<tr>
<td>poll</td>
</tr>
<tr>
<td>blkinfo</td>
</tr>
<tr>
<td>blkwrite</td>
</tr>
<tr>
<td>blkread</td>
</tr>
<tr>
<td>netinfo</td>
</tr>
<tr>
<td>netwrite</td>
</tr>
<tr>
<td>netread</td>
</tr>
<tr>
<td>halt</td>
</tr>
</tbody>
</table>
Unikernel isolation comes from the interface

• Direct mapping between 10 hypercalls and system call/resource pairs

• 6 for I/O
  • Network: packet level
  • Storage: block level

• vs. >350 syscalls

<table>
<thead>
<tr>
<th>Hypercall</th>
<th>System Call</th>
<th>Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>walltime</td>
<td>clock_gettime</td>
<td></td>
</tr>
<tr>
<td>puts</td>
<td>write</td>
<td>stdout</td>
</tr>
<tr>
<td>poll</td>
<td>ppoll</td>
<td>net_fd</td>
</tr>
<tr>
<td>blkinfo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>blkwrite</td>
<td>pwrite64</td>
<td>blk_fd</td>
</tr>
<tr>
<td>blkread</td>
<td>pread64</td>
<td>blk_fd</td>
</tr>
<tr>
<td>netinfo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>netwrite</td>
<td>write</td>
<td>net_fd</td>
</tr>
<tr>
<td>netread</td>
<td>read</td>
<td>net_fd</td>
</tr>
<tr>
<td>halt</td>
<td>exit_group</td>
<td></td>
</tr>
</tbody>
</table>
Implementation: nabla

- Extended Solo5 unikernel ecosystem and ukvm
- Prototype supports:
  - MirageOS
  - IncludeOS
  - Rumprun

- https://github.com/solo5/solo5
Measuring isolation: common applications

- Code reachable through interface is a metric for attack surface
- Used kernel _ftrace_

- Results:
  - Processes: 5-6x more
  - VMs: 2-3x more
Measuring isolation: fuzz testing

• Used kernel **ftrace**
• Used **trinity** system call fuzzer to try to access more of the kernel

• Results:
  • Nabla policy reduces by 98% over a “normal” process
Measuring performance: throughput

• Applications include:
  • Web servers
  • Python benchmarks
  • Redis
  • etc.

• Results:
  • 101%-245% higher throughput than ukvm
Measuring performance: CPU utilization

• vmexits have an effect on instructions per cycle

• Experiment with MirageOS web server

• Results:
  • 12% reduction in cpu utilization over ukvm
Measuring performance: startup time

• Startup time is important for serverless, NFV

• Results:
  • Ukvm has 30-370% higher latency than nabla

• Mostly due avoiding KVM overheads
Summary and Next Steps

• Unikernels should run as processes!
  • Maintain isolation via thin interface
  • Improve performance, etc.

• Next steps: can unikernels as processes be used to improve container isolation?
  • Nabla containers
  • https://nabla-containers.github.io/