gVisor and Unikernel

Xuyang Cao, slides adapted from Prof. Yiyeng Zhang
gVisor
Recap: container in the OS

Apps are contained as lightweight entities

However, still loose isolation and many possible threats:

• Kernel attack containers
• Apps attack containers
• Containers attack containers
• Containers attack kernel
  • Containers share the same kernel logics (e.g., TCP/IP stack)
  • Share the same hardware drivers

“Containers do not really contain” — Dan Walsh, 2014
Apps/Containers Attack Kernel

Through system calls

2046 Common Vulnerabilities and Exposures (CVE) since 1999
Recap: Conventional VM

On the other end

- Independent guest kernels
- Virtual hardware interface
  - Clear privilege separation and state encapsulation
- But virtualized hardware interface is inflexible
  - e.g., can’t change number of virtualized cores at run time
- and VM is heavy weight

Q: Why/how this extra guest OS layer really helps improve security?
Container  Somewhere in between?  VM
Sandboxing
Adding an extra layer around container

- Rule-based: restricting what applications can access
  - e.g., AppArmor, SELinux, Seccomp-bpf
- Rules can be fragile (not properly capture threats) and can’t prevent side channel attacks
gVisior
An Application Kernel to Intercept Common Syscalls

Key features:

• Implement a substantial portion of system calls (221 calls so far).
• Not rule-based filters.
• Flexible resource footprints.
• Written in Go, a memory/type-safe language.

Q: Can sandboxing prevent apps exploit/attack container?
gVisior Internal Architecture

runsc:
• interface to run the user app.

Sentry:
• Implementation/emulation of syscalls.
• Intercept syscalls from the user app.
• Itself sandboxed by seccomp, so avoid Sentry security loophole.

Gofer:
• Additional layer of isolation.
• Handle file access/management

Q: Why Gofer rather than putting file access in Sentry as well?
How Sentry intercepts Syscalls

Approach 1: ptrace
• ptrace to interference with code running.

Approach 2: KVM
• Augment Sentry as a guest OS on KVM to intercept syscalls.
Performance and Cautions

• 15MB memory usage
• 150ms startup time

What it IS good for:
• Small containers
• Spin up quickly
• High density

What it’s NOT good for:
• Trusted images (which can run on normal containers for better performance)
• Syscall heavy workloads
• Direct access to hardware
• Applications that use syscalls not supported by gVisor

Figure 3: **System Call Overhead.** The bars show the average latency for `gettimeofday` across 100M executions.
Unikernel
Recap: Different OS Structure

- **User-Mode**
  - Monolithic Kernel
  - MicroKernel
  - ExoKernel (Library OS/LibOS)

- **Kernel-Mode**
Take an extreme but fair position...

- VMs are usually single-purpose: each running single app.
  - So why still have unnecessary modules exist at all, even they can be unloaded.

- Backward compatibility, hmm, not very important.
  - We can just upgrade the apps. And even more libs can be removed.

- Create an extremely light libOS for a specific app.
  - Literally only the needed libs are included.

Q: any drawback of single-image/purpose VM?
Some background

• LibOS, system functions implemented as libraries in userspace.

• Unikernel: specialized, sealed, single-purpose libOS VMs that run directly on the hypervisor.

• MirageOS: the prototype unikernel instance.
MirageOS/Unikernel Solution

Configuration and compilation together programmatically.

Information needed to decide all required system libraries for this appliance.

Decidable, deterministic, and fast through static analysis.

Configuration Files
- Application Binary
- Language Runtime
- Parallel Threads
User Processes
- OS Kernel
- Hypervisor
Hardware

Mirage Compiler
- application source code
- configuration files
- hardware architecture
- whole-system optimisation

Mirage Runtime
- Hypervisor
- Hardware

OCaml
Using OCaml with strong type-checking

Extremely light binary (e.g., from hundreds MB to hundreds KB)
VM, Container, gVisor, Unikernels

Virtual Machines

- VM
  - Hypervisor
  - Hardware
  - Guest OS
  - Apps

Containers

- Container
  - Host OS
  - Hardware
  - LibOS
  - Apps

- Container
  - Host OS
  - Hardware
  - LibOS
  - Apps

Unikernels

- VM
  - Hypervisor
  - Hardware
  - LibOS
  - Apps

- VM
  - Hypervisor
  - Hardware
  - LibOS
  - Apps
Discussion: comparison between gVisior and Unikernel

- Security
- Weight
- Complexity
- Performance
VM, Container, gVisior, Unikernels

+ Strong isolation/security
- Heavy-weight

- Weak isolation/security
+ Light-weight

+ Strong isolation/security
+ Light-weight

VM, Container, Unikernels

VM

App, App
Guest OS

Hypervisor

Hardware

Virtual Machines

VM

App, App
Guest OS

Hypervisor

Hardware

Containers

Container
App, App
LibOS

Host OS

Hardware

Unikernels
Unikernel Designs

- Integrating configurations into the compilation process
  - All related services, applications packed into a single application
  - Features not used are not compiled => extensive dead-code elimination
- Single-purpose libOS VMs perform only what the application needs and rely on hypervisor for isolation and resource multiplexing
- Within a unikernel VM, there’s no privilege difference between application and libOS (single address space)
  - Which mode/ring should unikernel run in?
  - What does single address space imply?
- Single (type-safe) language for everything
- Unikernel is sealed at run time and cannot dynamically add code (better security)
  - No writable and executable, no heap expansion
Unikernel Benefits

• Lightweight
  • Only what the application uses is compiled and deployed
• Faster startup time (compared to VMs)
• Better security
  • Isolates libOS’s by hypervisor
  • Small attack surface
  • Single type-safe language, page table sealing, compile-time address space randomization
• Fits many new cloud environments well
  • Serverless, microservices, NFV
Other MirageOS Key Features

- Security
  - Defense-in-depth approach:
    - (1) Compile-time specialization
      - Dependency graph is deterministic so no misconfiguration.
    - (2) Pervasive type-safety
      - Single user with single address space, so no ACL.
      - Rely on OCaml strong type checking.
Other MirageOS Key Features (Continued)

- Compile-time Address Space Randomization
- Sealing the appliance after compilation
  - i.e., executable part is not writable.

- OCaml
  - System codes (e.g., IP/TCP stack) in C have to be rewritten.
  - Modified language runtime’s memory management and concurrency.
Modified Memory Layout

External I/O pages for communicating with other VMs

fast minor heap for short-lived values

larger major heap for long-lived values

Apps’ data and codes

IP/TCP network data

Fetched disk data

Figure 2: Specialised virtual memory layout of a 64-bit Mirage unikernel running on Xen.
Zero-Copy Device I/O

The very and only place to buffer IO data

Store metadata instead

Network encapsulation pipeline (in OCaml)

Q: good idea to rewrite system code (e.g., network stack)?

Figure 4: Example zero-copy write for an HTTP GET. The application writes its request into an I/O page, and the network stack segments it into fragments to write to the device ring. When a response arrives the pages are collected and the write thread notified.
Evaluation

Bootup Time (with optimization)

![Graph showing bootup time for Mirage and Linux PV with varying memory sizes.]

- Suitable for migration, as small downtime.

Figure 6: Boot time using an asynchronous Xen toolstack.
Raw TCP/IP Comparative Performance

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Throughput [ std. dev. ] (Mbps)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 flow</td>
<td>10 flows</td>
</tr>
<tr>
<td>Linux to Linux</td>
<td>1590 [ 9.1 ]</td>
<td>1534 [ 74.2 ]</td>
</tr>
<tr>
<td>Linux to Mirage</td>
<td>1742 [ 18.2 ]</td>
<td>1710 [ 15.1 ]</td>
</tr>
<tr>
<td>Mirage to Linux</td>
<td>975 [ 7.0 ]</td>
<td>952 [ 16.8 ]</td>
</tr>
</tbody>
</table>

Figure 8: Comparative TCP throughput performance with all hardware offload disabled.

Interestingly,

- Mirage has higher received throughput given lack of userspace data copy.
- Outbound throughput is low due to higher CPU usage (overhead by OCaml).
Disk I/O Comparative Performance

Bottlenecked because of the extra system buffer.

Figure 9: Random block read throughput, +/- 1 std. dev.
Application-Level Performance

Though some “tricks”/patches are done to improve the performance.

Q: are those patches only possible on MirageOS? Or can they also be applied to other system to improve performance?
Other Unikernels

- **OSv**: new OS designed to run single application
- **Rumprun**: running unmodified POSIX software as a unikernel
- **runtime.js**: libOS that runs JavaScript
- **IncludeOS**: libOS for running C++ code on virtual hardware
- **ClickOS**: A high-performance, virtualized software middlebox platform (e.g., for NFV)
- **Clive**: an OS designed to work in distributed and cloud environments, written in Go
- **Erlang on Xen**

*Find more info at [http://unikernel.org/projects/](http://unikernel.org/projects/)*
Conclusion

• Library OS (Exokernel) is an old idea aiming to expose more hardware interface directly to applications running in user space

• Unikernels: run app+libOS as VMs on hypervisor
  • Better isolation
  • Much more lightweight
  } \textit{Better fit for modern cloud environments}

• But need a lot of reimplementation and can’t use existing tooling
Discussion

- Pros and cons of unikernel? (vs. container, traditional VM, etc.)
- Do you think unikernel will be widely adopted?
Thank you!