University of Glasgow

School of Computing Science

60 YEARS OF COMPUTING AT GLASGOW

Virtualisation: Jails and Unikernels

Advanced Operating Systems

Lecture 18

Lecture Outline

- Alternatives to full virtualisation
- Sandboxing processes:
  - FreeBSD jails and Linux containers
  - Container management
- Unikernels and library operating systems

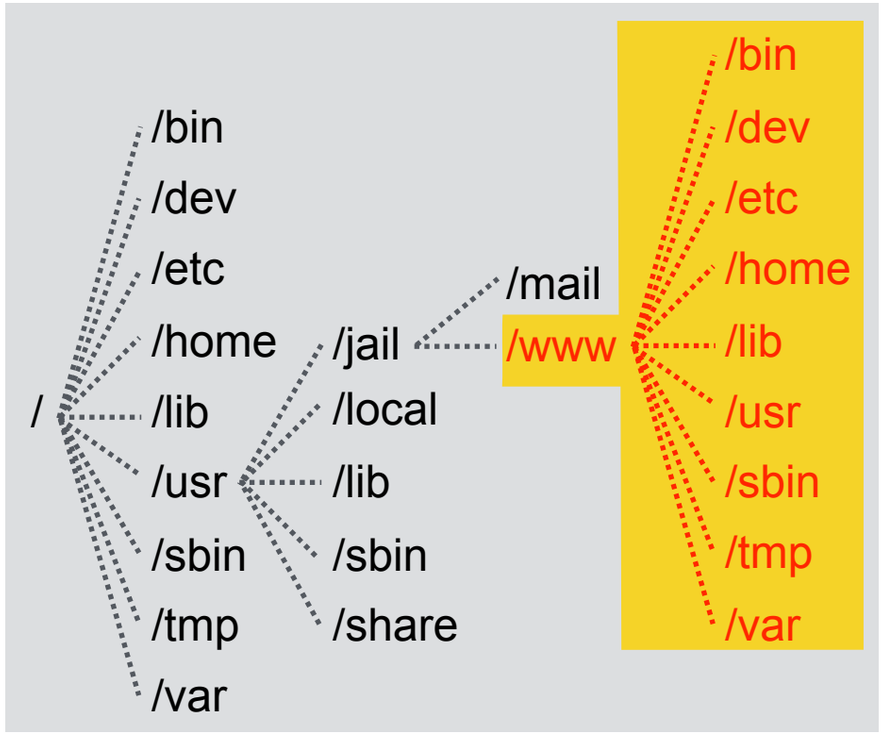
Alternatives to Full Virtualisation

- Full operating system virtualisation is expensive
  - Need to run a hypervisor
  - Need to run a complete guest operating system instance for each VM
  - High memory and storage overhead, high CPU load from running multiple OS instances on a single machine – but excellent isolation between VMs
- Running multiple services on a single OS instance can offer insufficient isolation
  - All must share the same OS
  - A bug in a privileged service can easily compromise entire system – all services running on the host, and any unrelated data
- A *sandbox* is desirable – to isolate multiple services running on a single operating system

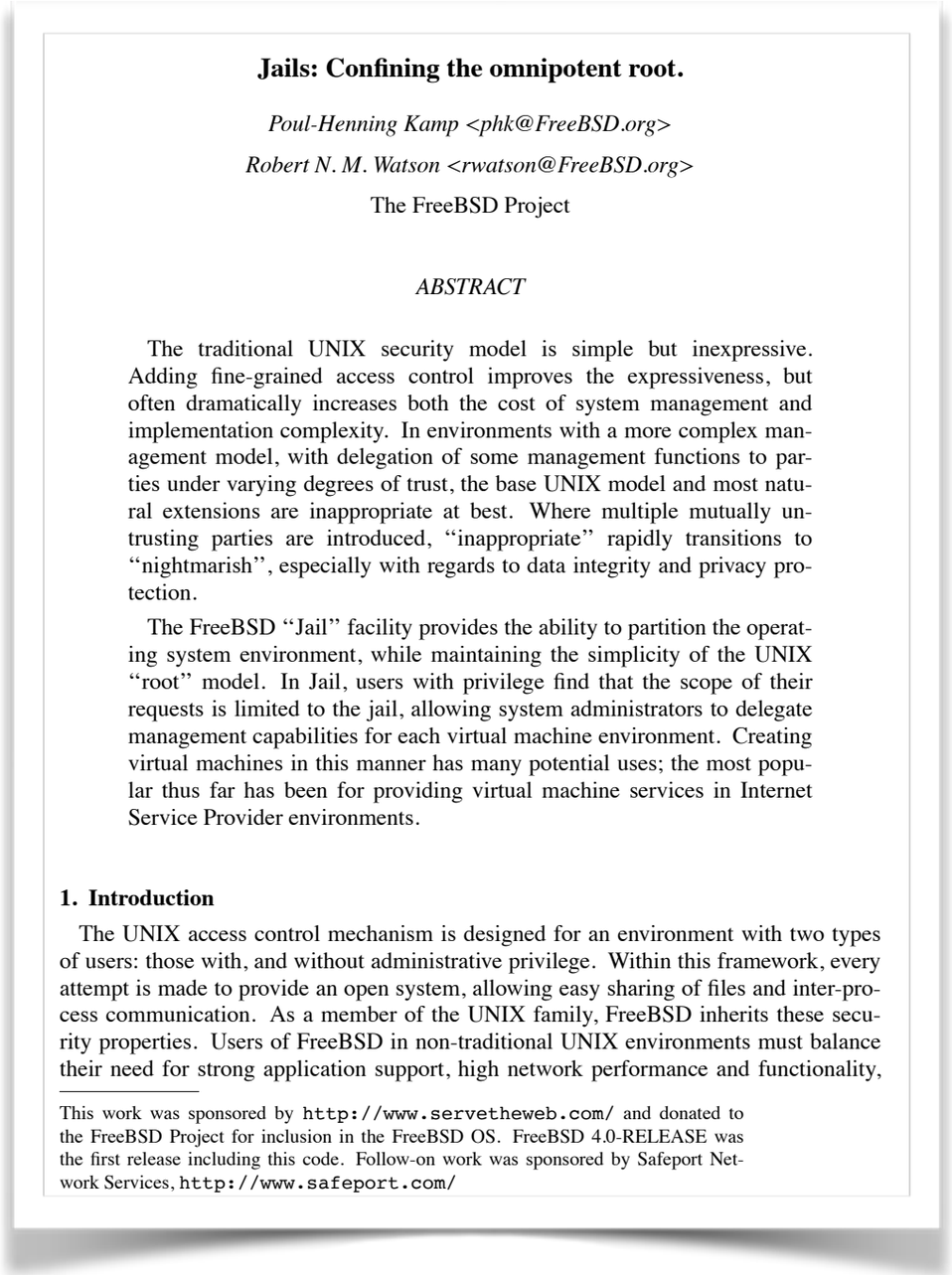
Sandboxing: Jails and Containers

- Concept – lightweight virtualisation
  - A single operating system kernel
  - Multiple services
  - Each service runs within a jail a service specific container, running just what's needed for that application

- Jail restricted to see only a subset of the filesystem and processes of the host
  - A sandbox within the operating system
  - Partially virtualised



Example: FreeBSD Jail subsystem



Jails: Confining the omnipotent root.

Poul-Henning Kamp <phk@FreeBSD.org>

Robert N. M. Watson <rwatson@FreeBSD.org>

The FreeBSD Project

ABSTRACT

The traditional UNIX security model is simple but inexpressive. Adding fine-grained access control improves the expressiveness, but often dramatically increases both the cost of system management and implementation complexity. In environments with a more complex management model, with delegation of some management functions to parties under varying degrees of trust, the base UNIX model and most natural extensions are inappropriate at best. Where multiple mutually untrusting parties are introduced, ''inappropriate'' rapidly transitions to ''nightmarish'', especially with regards to data integrity and privacy protection.

The FreeBSD ''Jail'' facility provides the ability to partition the operating system environment, while maintaining the simplicity of the UNIX ''root'' model. In Jail, users with privilege find that the scope of their requests is limited to the jail, allowing system administrators to delegate management capabilities for each virtual machine environment. Creating virtual machines in this manner has many potential uses; the most popular thus far has been for providing virtual machine services in Internet Service Provider environments.

1. Introduction

The UNIX access control mechanism is designed for an environment with two types of users: those with, and without administrative privilege. Within this framework, every attempt is made to provide an open system, allowing easy sharing of files and inter-process communication. As a member of the UNIX family, FreeBSD inherits these security properties. Users of FreeBSD in non-traditional UNIX environments must balance their need for strong application support, high network performance and functionality,

This work was sponsored by `http://www.servetheweb.com/` and donated to the FreeBSD Project for inclusion in the FreeBSD OS. FreeBSD 4.0-RELEASE was the first release including this code. Follow-on work was sponsored by Safeport Network Services, `http://www.safeport.com/`

- Unix has long had `chroot()`
  - Limits process to see a subset of the filesystem
  - Used in, e.g., web servers, to prevent buggy code from serving files outside chosen directory
- Jails extend this to also restrict access to other resources for jailed processes:
  - Limits process to see subset of the processes (the children of the initially jailed process)
  - Limits process to see specific network interface
  - Prevents a process from mounting/un-mounting file systems, creating device nodes, loading kernel modules, changing kernel parameters, configuring network interfaces, etc.
  - Even if running with `root` privilege
- Processes outside the jail can see into the jail – from outside, jailed processes look like regular processes

Benefits of Jails

- Extremely lightweight implementation
  - Processes in the jail are regular Unix processes, with some additional privilege checks
  - FreeBSD implementation added/modified ~1,000 lines of code in total
  - Contents of the jail are limited: it's not a full operating system – just the set of binaries/libraries needed for the application
- Straightforward system administration
  - Few new concepts: no need to learn how to operate the hypervisor
  - Visibility into the jail eases debugging and administration; no new tools
- Portable
  - No separate hypervisor, with its own device drivers, needed
  - If the base system runs, so do the jails

Disadvantages of Jails

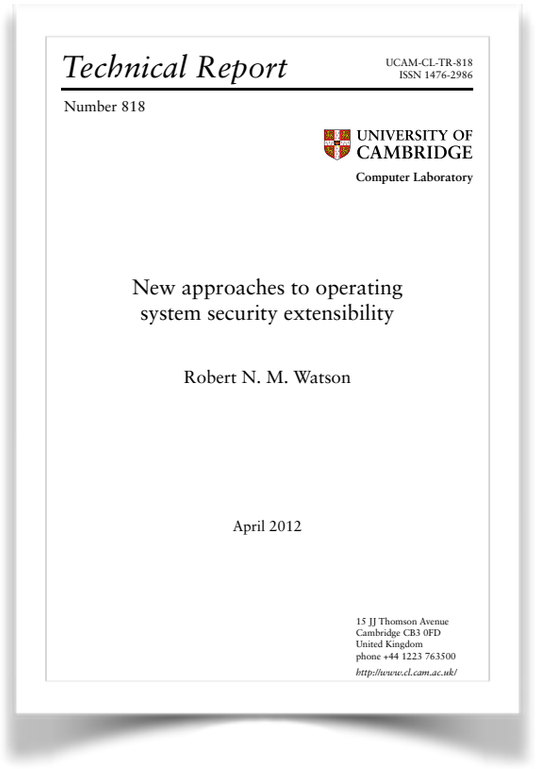
- Virtualisation is imperfect
  - Full virtual machines can provide stronger isolation and performance guarantees – processes in jail can compete for resources with other processes in the system, and observe the effects of that competition
  - A process can discover that it's running in a jail – a process in a VM cannot determine that it's in a VM
- Jails are tied to the underlying operating system
  - A virtual machine can be paused, migrated to a different physical system, and resumed, without processes in the VM being aware – a jail cannot

Example: Linux Containers

- Two inter-related kernel features:
  - Namespace isolation – separation of processes
  - Control groups “cgroups” – accounting for process CPU, memory, disk and network I/O usage
  - Both part of the mainline Linux kernel
- Combined to provide container-based virtualisation
  - Implemented via tools such as “lxc”, “OpenVZ”, “Linux-VServer”
  - Very similar functionality to FreeBSD Jails, but using the Linux kernel as the underlying OS
  - Run a subset of processes within a containers – can be a full OS-like environment, or something much for limited

Example: iOS/macOS Sandbox

- All apps on iOS, and all apps from the Mac App Store on macOS, are sandboxed
  - Sandbox functionality varies with macOS/iOS version – gradually getting more sophisticated and more secure
  - Kernel heavily based on FreeBSD, but sandboxing doesn't use Jails
    - Mandatory access control – http://www.trustedbsd.org/mac.html
    - Apps are constrained to see a particular directory hierarchy – but given flexibility to see other directories outside this based on user input
    - Policy language allows flexible control over what OS functions are restricted – restricts access to files, processes, and other namespaces like Jails, but more flexibility in what parts of the system are exposed
    - At most restrictive, looks like a FreeBSD Jail containing a single-process and it's data directories
    - Can allow access to other OS services/directories as needed – more flexible, but more complex and hence harder to sandbox correctly
  - Heavily based on Robert Watson's PhD and work on FreeBSD MAC and Capsicum security framework
    - http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-818.pdf
- Trade-off: flexibility, complexity, security


Technical Report
UCAM-CL-TR-818
ISSN 1476-2986
Number 818
UNIVERSITY OF CAMBRIDGE
Computer Laboratory
New approaches to operating system security extensibility
Robert N. M. Watson
April 2012
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500
http://www.cl.cam.ac.uk/

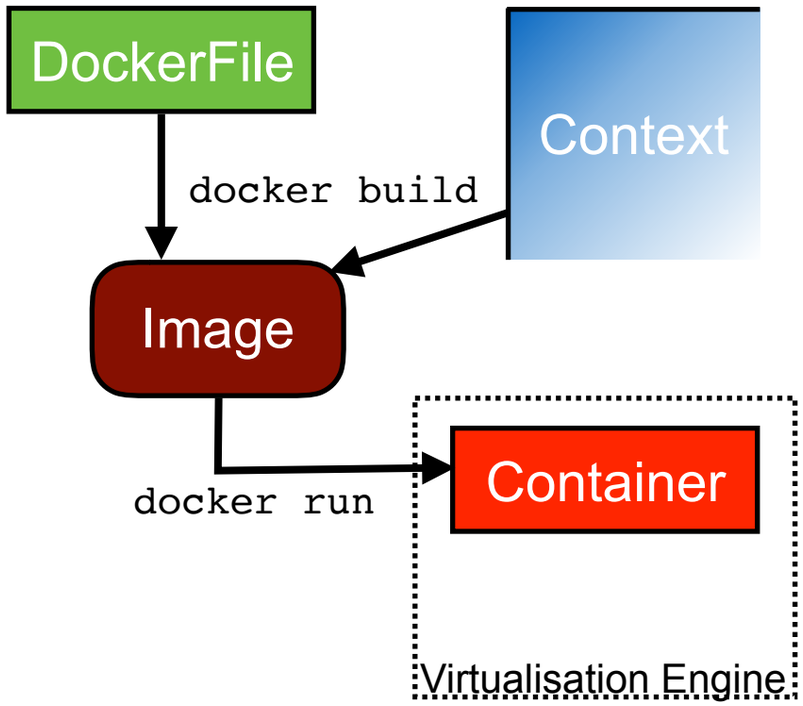
Container Management

- Full virtualisation – VMs run complete OS, and can use standard OS update/management procedures
- Jails run a container with a subset of the full OS
  - Want minimal stack needed to support services running in the container
  - Less software installed in container → security vulnerability less likely

- How to ensure all images are up-to-date?
  - No longer sufficient to patch the OS, must also patch each container
  - Need to minimally subset the OS for the services in the container – can this be automated?
  - How to share service/container configuration?

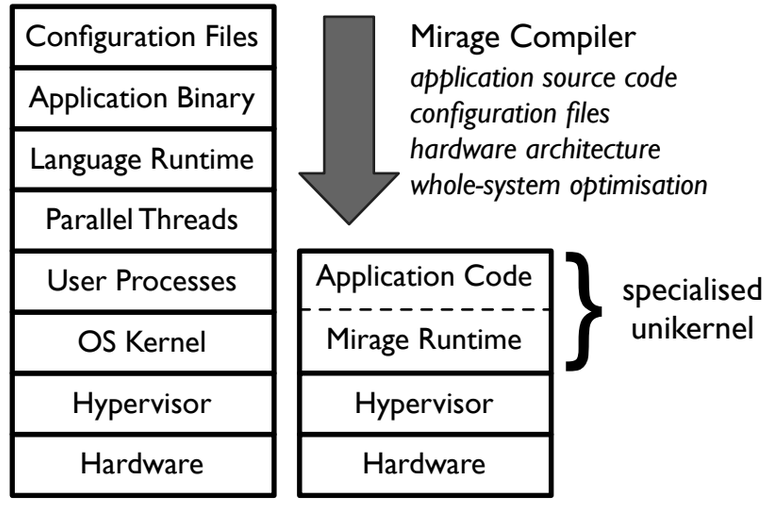
Example: Docker

- DockerFile and context specify an image
  - Base system image
    - Docker maintains official, managed and security updated, images for popular operating systems
    - Images can build on/extend other images
  - Install additional binaries and libraries
  - Install additional data
  - Commands to execute when image is instantiated
  - Metadata about resulting image
- Images are immutable → instantiated in container
  - Virtualisation uses Linux containers, the macOS sandbox, or FreeBSD jails to instantiate images, depending on underlying operating system running the container
    - macOS and Windows run Linux in a hypervisor, and use that to execute containers
    - FreeBSD relies on native Linux system call emulation, and jails
  - Image can specify external filesystems to mount, to hold persistent data accessible by running image
  - A standardised way of packaging a software image to run in a container – plus easy-to-manage tools for instantiating a container



Unikernels: Library Operating Systems

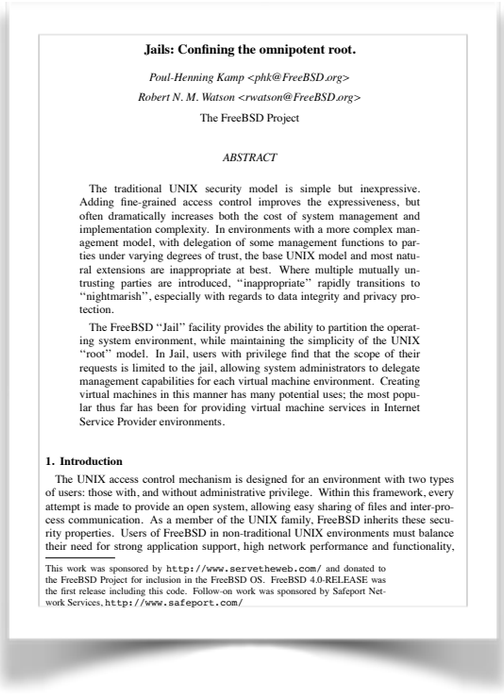
- Taking the idea of service-specific containers to its logical extreme – a library operating system
- Well-implemented kernels have clear interfaces between components
  - Device drivers, filesystems, network protocols, processes, …
- Implement as a set of libraries, that can link with an application
  - Only compile in the subset of functions needed to run the desired application (e.g., you don't use UDP, it isn't linked in to your kernel, etc…)
  - Link entire operating system kernel and application as a single binary
  - Run on bare hardware, or within a hypervisor
- Typically written in high-level, type-safe, languages, with expressive module systems – not backwards compatible with Unix
  - E.g., MirageOS largely written in OCaml
  - Type-safe languages give clear interface boundaries, to enable automatic minimisation of images



Source: A. Madhavapeddy *et al.*, "Unikernels: Library Operating Systems for the Cloud", Proc. ACM ASPLOS, Houston, TX, USA, March 2013. DOI:10.1145/2451116.2451167

Further Reading

- P.-H. Kamp and R. Watson, “Jails: Confining the omnipotent root”, System Administration and Network Engineering Conference, May 2000. http://www.sane.nl/events/sane2000/papers/kamp.pdf
  - Trade-offs vs. complete system virtualisation?
  - Overheads vs. flexibility vs. ease of management?
  - Benefits of Docker-style configuration of images



Jails: Confining the omnipotent root.

Poul-Henning Kamp <phk@FreeBSD.org>
Robert N. M. Watson <rwatson@FreeBSD.org>
The FreeBSD Project

ABSTRACT

The traditional UNIX security model is simple but inexpressive. Adding fine-grained access control improves the expressiveness, but often dramatically increases both the cost of system management and implementation complexity. In environments with a more complex management model, with delegation of some management functions to parties under varying degrees of trust, the base UNIX model and most natural extensions are inappropriate at best. Where multiple mutually untrusting parties are introduced, ‘‘inappropriate’’ rapidly transitions to ‘‘nightmarish’’, especially with regards to data integrity and privacy protection.

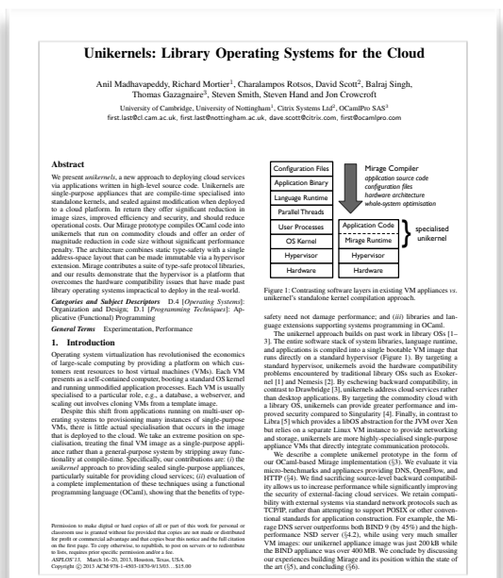
The FreeBSD ‘‘Jail’’ facility provides the ability to partition the operating system environment, while maintaining the simplicity of the UNIX ‘‘root’’ model. In Jail, users with privilege find that the scope of their requests is limited to the jail, allowing system administrators to delegate management capabilities for each virtual machine environment. Creating virtual machines in this manner has many potential uses; the most popular thus far has been for providing virtual machine services in Internet Service Provider environments.

1. Introduction

The UNIX access control mechanism is designed for an environment with two types of users: those with, and without administrative privilege. Within this framework, every attempt is made to provide an open system, allowing easy sharing of files and inter-process communication. As a member of the UNIX family, FreeBSD inherits these security properties. Users of FreeBSD in non-traditional UNIX environments must balance their need for strong application support, high network performance and functionality,

This work was sponsored by http://www.servetheweb.com/ and donated to the FreeBSD Project for inclusion in the FreeBSD OS. FreeBSD 4.0-RELEASE was the first release including this code. Follow-on work was sponsored by Safeport Network Services, http://www.safeport.com/

- A. Madhavapeddy et al., “Unikernels: Library Operating Systems for the Cloud”, ACM ASPLOS, Houston, TX, USA, March 2013. DOI:10.1145/2451116.2451167
  - Is optimising operating system for single application going too far?
  - Are unikernels maintainable?
  - Relation to containers and hypervisors?



Unikernels: Library Operating Systems for the Cloud

Anil Madhavapeddy, Richard Mortier[1], Charalampos Rotsos, David Scott[2], Balraj Singh, Thomas Gazagnaire[3], Steven Smith, Steven Hand and Jon Crowcroft
University of Cambridge, University of Nottingham[1], Citrix Systems Ltd[2], OCamlPro SAS[3]
first.last@cl.cam.ac.uk, first.last@nottingham.ac.uk, dave.scott@citrix.com, first@ocamlpro.com

Abstract

We present *unikernels*, a new approach to deploying cloud services via applications written in high-level source code. Unikernels are single-purpose appliances that are compile-time specialised into standalone kernels, and sealed against modification when deployed to a cloud platform. In return they offer significant reduction in image sizes, improved efficiency and security, and should reduce operational costs. Our Mirage prototype compiles OCaml code into unikernels that run on commodity clouds and offer an order of magnitude reduction in code size without significant performance penalty. The architecture combines static type-safety with a single address-space layout that can be made immutable via a hypervisor extension. Mirage contributes a suite of type-safe protocol libraries, and our results demonstrate that the hypervisor is a platform that overcomes the hardware compatibility issues that have made past library operating systems impractical to deploy in the real-world.

Categories and Subject Descriptors D.4 [*Operating Systems*]: Organization and Design. D.1 [*Programming Techniques*]: Applicative (Functional) Programming

General Terms Experimentation, Performance

1. Introduction

Operating system virtualization has revolutionised the economics of large-scale computing by providing a platform on which customers rent resources to host virtual machines (VMs). Each VM presents as a self-contained computer, booting a standard OS kernel and running unmodified application processes. Each VM is usually specialised to a particular role, e.g., a database, a webserver, and scaling out involves cloning VMs from a template image.

Despite this shift from applications running on multi-user operating systems to provisioning many instances of single-purpose VMs, there is little actual specialisation that occurs in the image that is deployed to the cloud. We take an extreme position on specialisation, treating the final VM image as a single-purpose appliance rather than a general-purpose system by stripping away functionality at compile-time. Specifically, our contributions are: (*i*) the *unikernel* approach to providing sealed single-purpose appliances, particularly suitable for providing cloud services; (*ii*) evaluation of a complete implementation of these techniques using a functional programming language (OCaml), showing that the benefits of type-safety need not damage performance; and (*iii*) libraries and language extensions supporting systems programming in OCaml.

Figure 1: Contrasting software layers in existing VM appliances *vs.* unikernel's standalone kernel compilation approach.

The unikernel approach builds on past work in library OSs [1–3]. The entire software stack of system libraries, language runtime, and applications is compiled into a single bootable VM image that runs directly on a standard hypervisor (Figure 1). By targeting a standard hypervisor, unikernels avoid the hardware compatibility problems encountered by traditional library OSs such as Exokernel [1] and Nemesis [2]. By eschewing back-ward compatibility, in contrast to Drawbridge [3], unikernels address cloud services rather than desktop applications. By targeting the commodity cloud with a library OS, unikernels can provide greater performance and improved security compared to Singularity [4]. Finally, in contrast to Libra [5] which provides a IaOS abstraction for the JVM over Xen but relies on a separate Linux VM instance to provide networking and storage, unikernels are more highly-specialised single-purpose appliance VMs that directly integrate communication protocols.

We describe a complete unikernel prototype in the form of our OCaml-based Mirage implementation (§3). We evaluate it via micro-benchmarks and appliances providing DNS, OpenFlow, and HTTP (§4). We find sacrificing source-level backward compatibility allows us to increase performance while significantly improving the security of external-facing cloud services. We retain compatibility with external systems via standard network protocols such as TCP/IP, rather than attempting to support POSIX or other conventional standards for application construction. For example, the Mirage DNS server outperforms both BIND 9 (by 45%) and the high-performance NSD server (§4.2), while using very much smaller VM images: our unikernel appliance image was just 200 kB while the BIND appliance was over 400 MB. We conclude by discussing our experiences building Mirage and its position within the state of the art (§5), and concluding (§6).

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.
Copyright © 2013 ACM 978-1-4503-1870-9/13/03…$15.00