Towards a smaller and safer OS

Thomas Gazagnaire

thomas@gazagnaire.org

École Normale Supérieure
Année 2017-2018
Systèmes et Réseaux
Context

how can we build trustable systems?
Some Good News

Static analysis tools are becoming mainstream: FramaC, Astrée, Coccinelle, Infer, …

‣ Target C/C++/Objective-C/Java applications

‣ Scaling from partial core logic to complete “real-world” application (current sizes: 100k — 10m loc)

‣ Difficult balance between scaling and power of analysis (bound checks v.s dynamic allocation, data races, …)
So why do we still see this?
Some bad news

Complexity of today’s traditional system software stack makes full system analysis impossible

- Application code is a small % of the runtime environment
- Runtime is historically split into abstraction layers with different communities
- In deployment environments, developers do not control that stack (vs. operators)
MirageOS proposal

MirageOS is a **library OS** and a **compiler** which can build specialised images containing only the runtime environment which is needed by an application.
MirageOS proposal

- Cut the complexity by designing the layers as independent libraries
- The MirageOS compiler transforms an application manifest into a specialised image
- Rely on the OCaml compiler to perform modular static analysis, dead-code elimination, etc.
- Rely on the OCaml runtime as the sole trusted runtime environment (and selected C bindings)
Plan

1. Modern Runtime Environments
2. Unikernels
3. MirageOS
   a) library OS
   b) compiler
   c) interesting use-cases
4. Conclusion & Future
Modern Runtime Environments
Application

- What developers care about
- What developers can control
- Static analysers, verifying compilers, code extraction from proof assistants, etc. exist and are used

<table>
<thead>
<tr>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration files</td>
</tr>
<tr>
<td>Language Runtime</td>
</tr>
<tr>
<td>Shared Libraries</td>
</tr>
<tr>
<td>Kernel</td>
</tr>
<tr>
<td>Hypervisor</td>
</tr>
<tr>
<td>Firmware</td>
</tr>
</tbody>
</table>
Applications need runtime specialisation, available via CLI options or configuration files

- formats are not fully specified and need to interact with POSIX

- Often a neglected problem (because “boring”)

<table>
<thead>
<tr>
<th>Application</th>
<th>Configuration files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Language Runtime</td>
</tr>
<tr>
<td></td>
<td>Shared Libraries</td>
</tr>
<tr>
<td></td>
<td>Kernel</td>
</tr>
<tr>
<td></td>
<td>Hypervisor</td>
</tr>
<tr>
<td></td>
<td>Firmware</td>
</tr>
</tbody>
</table>
Language Runtime

- GC, bound checks, etc need dynamic code instrumentation

- Usually written in C:
  - buffer overflows, NULL pointers exceptions, etc.
  - interaction with POSIX

- The runtime can be statically or dynamically linked

<table>
<thead>
<tr>
<th>Application</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration files</td>
<td></td>
</tr>
<tr>
<td><strong>Language Runtime</strong></td>
<td></td>
</tr>
<tr>
<td>Shared Libraries</td>
<td></td>
</tr>
<tr>
<td>Kernel</td>
<td></td>
</tr>
<tr>
<td>Hypervisor</td>
<td></td>
</tr>
<tr>
<td>Firmware</td>
<td></td>
</tr>
</tbody>
</table>
Shared Libraries

- Language runtime or C bindings call the C “standard library”

- glibc is the most popular
  - 500k lines of C code
  - To avoid duplication and license issues the is dynamically loaded and can be swapped by operators

- Static library: musl (Alpine Linux)
The kernel orchestrates unprivileged processes and ensures fair access to physical resources of the machine.

Rich API for communicating between processes and kernel: Linux has +300 syscalls, Windows +500. API is layered:
- low-level syscalls: read, write
- high-level: socket API

Linux 4.14 kernel is 20 millions lines of C code.
Hypervisor: a.k.a “Cloud”

- Allow to run multiple operating systems on a same machine
- Can be seen as the kernel for orchestrating “virtual” OS
Hypervisor: Xen

- New CPU instructions available since 2006 for virtualisation support (VT-x and AMD-V)

- Xen is a new reimplementation of basic OS mechanism (orchestration, resources allocation, …)

- I/O is done via device emulation in a privileged Linux VM. Use QEMU or custom Xen devices. Virtual devices have a very stable API

- Rare vulnerabilities in Xen itself (but still)

- QEMU is 1 million line of C code
  - included a user-space TCP/IP stack
  - VENOM (use floppy disk devices to take over the host)
Hypervisor: KVM

- Similar architecture
- The VM orchestrator is the Linux kernel (e.g. VMs are processes)
- I/O emulation is similarly provided by QEMU
Firmware

• Major unknown for developers & operators today

• What we know:
  • Implement the UEFI spec: a full OS on its own (comparable in complexity to the Linux kernel)
  • Management Unit is running Minix (TCP/IP + web server)
  • Spectre / Meltdown / …
  • Attacks are permanent: malicious software cannot be removed!

• Open-source CPUs (RISK-V) are coming soon!
Unikernels
Assumptions

Restricted Runtime Environment

- one specific application: no new installation nor updates
- no support for multi-users: only a single user
- deployment platform known at compile time
- configuration options known at compile-time
- access to all the source-code, for all the layers
Unikernels

1. Library OS: develop a modular OS

1. Extended linker: extend the linking step to merge configuration files, user-space and kernel libraries into a single, specialised image

2. Library VMM (Virtual Machine Monitor): develop a modular QEMU for safer virtualised I/O
1. Library OS

1/2 cutting existing OS into pieces

- **rumpkernel**: splits NetBSD kernel into libraries. Upstreamed.

- **LKL**: splits Linux kernel into libraries. Not plan to upstream the patches yet.

- **Drawbridge**: splits Windows into libraries. Used to port (Win32) SQL server to Linux.
1. Library OS
2/2 writing new Operating Systems

- Ambitious but possible
  - FoxNet: re-implement a TCP/IP stack in SML in the 90s
  - seL4: certified micro-kernel implementation (took >3 years)
  - MirageOS! :-)

- Fundamental issue: devices drivers become obsolete pretty quickly (rate of new hardware is increasing): new OSes are not maintainable.
  Solutions:
  - run devices in user-space (like L4)
  - target only virtual I/O devices and use MiniOS (a minimal OS for Xen)
  - support NetBSD devices by using rumpkernel
## 1. Library OS

<table>
<thead>
<tr>
<th>Projet</th>
<th>Langage</th>
<th>Pilotes</th>
<th>Déploiement</th>
</tr>
</thead>
<tbody>
<tr>
<td>MirageOS</td>
<td>OCaml</td>
<td>MiniOS &amp; solo5</td>
<td>Xen, KVM</td>
</tr>
<tr>
<td>rumprun</td>
<td>C</td>
<td>rump kernel</td>
<td>Xen, KVM</td>
</tr>
<tr>
<td>HalVM</td>
<td>Haskell</td>
<td>MiniOS</td>
<td>Xen</td>
</tr>
<tr>
<td>ClickOS</td>
<td>C</td>
<td>MiniOS</td>
<td>Xen</td>
</tr>
<tr>
<td>IncludeOS</td>
<td>C++</td>
<td>MiniOS, solo5</td>
<td>Xen, KVM</td>
</tr>
<tr>
<td>DeferPanic</td>
<td>Go</td>
<td>rump kernel</td>
<td>Xen, KVM</td>
</tr>
<tr>
<td>LING</td>
<td>Erlang</td>
<td>MiniOS</td>
<td>Xen</td>
</tr>
<tr>
<td>OSv</td>
<td>Java</td>
<td>MiniOS</td>
<td>Xen</td>
</tr>
<tr>
<td>runtime.js</td>
<td>Javascript</td>
<td>custom</td>
<td>KVM</td>
</tr>
</tbody>
</table>
2. Extended Linker

- An optimising compiler can remove dead-code and inline code across traditionally opaque layers: resulting images are usually have a size of few MiB.

- Configuration can be partially evaluated at compile-time: extreme specialisation enables boot-time of a few ms.

- If something (e.g. networking) is not specified an the application manifest, it will not be available at runtime. Minimal runtime environments need a few MiB of RAM.

- Kernel and user-space share the same address-space; lots of runtime checks are removed: static analysis is key.
3. Library VMM

- IBM’s *solo5* replaces QEMU by a custom monitor, specialised to run one custom unikernel.

- The same application manifest, used to generate the unikernel, can be used to generate a monitor to handle I/O devices.

- Solo5 defines a minimal and portable interface between privileged monitors and unikernels running in sandbox.
  - KVM + QEMU or ukvm (specialised monitor)
  - FreeBSD + bhyve
  - Muen (certified static micro-kernel)
MirageOS
Quick Recap

- 2007: Anil Madhavapeddy’s PhD: Towards a "functional" Internet at University of Cambridge
- 2008: Citrix/XenServer libraries (Anil, Dave Scott, me)
- 2013: MirageOS 1.0 (opam support, mirage types, CLI)
- 2014: MirageOS 2.0 (ARM support, irmin, ocaml-tls)
- 2015: Unikernel System founded and acquired by Docker
- 2017: MirageOS 3.0 (solo5 support, result type, logs, …)
MirageOS
as a library OS
“Functors?”

demo
Signatures

mirage_types.ml defines module signatures for “standard” devices: time, clock, network, storage, ...

```ml
(** A network interface that serves Ethernet frames. *)
module type S = sig

  type t
  type buffer = Cstruct.t
  type macaddr = Macaddr.t
  type error = private [>`Unimplemented | `Disconnected]
  val pp_error : error Fmt.t

  val disconnect : t -> unit Lwt.t

  val write : t -> buffer -> (unit, error) result Lwt.t
  (** [write nf buf] outputs [buf] to netfront [nf]. *)

  val writev : t -> buffer list -> (unit, error) result Lwt.t
  (** [writev nf buf's] output a list of buffers to netfront [nf] as a single packet. *)

  val listen : t -> (buffer -> unit Lwt.t) -> (unit, error) result Lwt.t
  (** [listen nf fn] is a blocking operation that calls [fn buf] with every packet that is read from the interface. The function can be stopped by calling [disconnect] in the device layer. *)

  val mac : t -> macaddr
  (** [mac nf] is the MAC address of [nf]. *)

end
```

Mirage_net_lwt.S
Signatures

mirage_types.ml defines module signatures for “standard” devices: time, clock, network, storage, ...

(** An simple interface for read-only key/value stores. *)
module type RO = sig

  type t
  type buffer = Cstruct.t
  type error = private [> 'Unknown_key of string]
  val pp_error : error Fmt.t

  val disconnect : t -> unit Lwt.t

  val read : t -> string -> int64 -> int64 ->
            (Cstruct.t list, error) result Lwt.t
  (** [read t key offset length] reads up to [length] bytes from the
      value associated with [key]. If less data is returned than
      requested, this indicates the end of the value. *)

  val mem : t -> string -> (bool, error) result io
  (** [mem t key] returns [true] if a value is set for [key] in [t],
      and [false] if not so. *)

  val size : t -> string -> (int64, error) result io
  (** Get the value size. *)

end

Mirage_kv_lwt.R0
Implementations

- Module signatures are the backbone of MirageOS
- Every signature has multiple implementations
- Implementations have specialised constructors
- Implementations might define more concrete errors
Implementations

Mirage_net_raw.mli

```ocaml
type error = [Mirage_net.error | Unix of Unix.error]

include Mirage_net_lwt.S with type error := error

val connect : string -> t Lwt.t
(** [connect i] opens the Linux interface [i] (for instance "eth0") in raw mode. *)
```

Mirage_net_xen.mli

```ocaml
(** Xen netfront interface for Ethernet I/O *)
module Netfront : sig
  include Mirage_net_lwt.S
  val connect : int -> t Lwt.t
  (** [connect id] is the device number [id]. To use inside a domU. *)
end

(** Xen netback interface for Ethernet I/O *)
module Netback : sig
  include Mirage_net_lwt.S
  val connect : domain:int -> device:int -> t Lwt.t
  (** [connect ~domain ~device] is the device number [id] for the VM numer [domain]. To use inside dom0. *)
end
```
Implementations

Implementations can be functors with device signature as parameters. Follow same rules as normal implementation regarding errors and constructors.

```ocaml
module Make (N:Mirage_net.S) : sig

  include Mirage_ethif.S with type netif = N.t

  val connect : ?mtu:int -> netif -> t Lwt.t
  (** [connect ?mtu netif] connects an ethernet layer on top of the raw network device [netif]. The Maximum Transfer Unit may be set via the optional [?mtu] parameter, otherwise a default value of 1500 will be used. *)

end
```
Available libraries

<table>
<thead>
<tr>
<th>Domain</th>
<th>Bibliothèques</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interfaces Réseau</td>
<td>tuntap, vmnet, rawlink</td>
</tr>
<tr>
<td>Protocoles Réseau</td>
<td>ethernet, ARP, ICMP, IPv4, IPv6, UCP, TCP, DHCP, DNS, TFTP, HTTP</td>
</tr>
<tr>
<td>Protocoles de Stockage</td>
<td>FAT32, Git, tar, AES-CCM, ramdisk, B-trees</td>
</tr>
<tr>
<td>Sécurité</td>
<td>x509, ASN1, TLS, OTR</td>
</tr>
<tr>
<td>Crypto</td>
<td>MD5, SHA1, SHA224, SHA256, SHA384, SHA512, BLAKE2B, BLAKE2S, RMD160, 3DES, AES, DH, DSA, RSA, Fortuna, ECB/CBC/CCM/GCM modes</td>
</tr>
</tbody>
</table>

**TLS:** “rigorous engineering” e.g. same pure code to generate test oracles, verify oracle against real-world TLS traces and the real implementation

**B-trees:** code extracted from Isabelle/HOL
MirageOS as a compiler
MirageOS compiler

<table>
<thead>
<tr>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration files</td>
</tr>
<tr>
<td>Language Runtime</td>
</tr>
<tr>
<td>Shared Libraries</td>
</tr>
<tr>
<td>Kernel</td>
</tr>
<tr>
<td>Hypervisor</td>
</tr>
<tr>
<td>Firmware</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MirageOS compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Code</td>
</tr>
<tr>
<td>Userspace and Kernel Libraries</td>
</tr>
<tr>
<td>Configuration Files</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sandbox</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
</tr>
<tr>
<td>OCaml Runtime</td>
</tr>
</tbody>
</table>
MirageOS compiler
multi-stage pipeline

config.ml
mirage configure
main.ml
key_gen.ml
unikernel.ml

Makefile
opam

mirage build
image
MirageOS compiler
multi-stage pipeline

config.ml

mirage_configure

main.ml
key_gen.ml

opam

Makefile

unikernel.ml

functor application

image

mirage_build
Application Functor

A MirageOS application is a functor:
- using the standard signatures as parameters
- with a start function

```
open Lwt.Infix

module Hello (Time : Mirage_time_lwt.S) = struct

  let start () =
  let rec loop = function
    | 0 -> Lwt.return_unit
    | n ->
      Logs.info (fun f -> f "hello");
      Time.sleep_ns (Duration.of_sec 1) >>= fun () ->
      loop (n-1)
  in
  loop 4

end
```
Application Functor

The mirage website needs:

- 1 TCP/IPv4 stack
- 3 read-only key/value stores
  - private key
  - raw contents
  - templates
- 1 clock device to get the current time

```
module Make
  (S  : Mirage_stack_lwt.V4)
  (KEYS: Mirage_kv_lwt.RO)
  (FS  : Mirage_kv_lwt.RO)
  (TMPL: Mirage_kv_lwt.RO)
  (Clock: Mirage_clock_lwt.PCLOCK)
  : sig
    val start: S.t -> KEYS.t -> FS.t -> TMPL.t -> Clock.t -> unit Lwt.t
end
```
Application Manifest

- Application manifest is an OCaml file

- Use a eDSL to describe functors composition and configuration keys

- Describe the main application (reflect unikernel.ml)

- Use generic devices which will adapt to the deployment target:
  - network: Static IP vs. DHCP
  - KV/RO: block device, FAT32, crunched files, irmin, ...
  - clock: unix/xen default device

```ocaml
code

open Mirage

let title_key =
    let doc =
        Key.Arg.info ["title"]
        ~doc:"Website title."
        ~env:"WWW_TITLE"
    in
    let a = Key.Arg.(opt (some string) None doc) in
    Key.(create "title" a)

let keys = [Key.abstract title_key]
let packages = [package "cow"; package "cowabloga"]

let http =
    foreign ~packages ~keys "Dispatch.Make"
    (stackv4 @- kv_ro @- kv_ro @- pclock @- job)

let fs_key = Key.(value @@ kv_ro ())
let filesfs = generic_kv_ro ~key:fs_key "../files"
let tmplfs = generic_kv_ro ~key:fs_key "../tmpl"

let () =
    register "www" [
        http
        $ generic_stackv4 default_network
        $ filesfs
        $ tmplfs
        $ default_posix_clock
    ]
```
MirageOS compiler

multi-stage pipeline

config.ml

mirage configure

main.ml

key_gen.ml

Makefile

opam

CLI tool

mirage build

unikernel.ml

image
• **mirage** is the CLI tool to bind everything all together
• **mirage configure**
  • dynlinks config.ml and persists CLI parameters
  • Allow to choose the target layer for runtime environment
    - unix: useful to debug
    - or hypervisor: KVM, Xen, QubesOS, muen
  • generates main.ml to apply intermediate functors and call the main function from unikernel.ml
  • generate key_gen.ml to persist default configuration values and allows to overwrite configuration values at runtime
• generate opam file to install missing opam packages
• **mirage build** calls the OCaml compiler with the right options
MirageOS compiler
multi-stage pipeline

- config.ml
- main.ml
- key_gen.ml
- unikernel.ml
- Makefile
- opam

runtime environment

mirage configure

mirage build

image
MirageOS compiler
“dev mode”

Application Code
Userspace and Kernel Libraries
Configuration Files

MirageOS compiler

Application
OCaml Runtime
kernel
Hypervisor
Firmware
User-space and OS libraries

“dev mode”

mirage configure --target=unix --net=socket

![Image of a diagram showing various library dependencies and connections.]
User-space and OS libraries

“dev mode + MirageOS TCP/IP stack”

mirage configure --target=unix --net=direct
MirageOS compiler

“deployment mode”

<table>
<thead>
<tr>
<th>Application Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Userspace and Kernel Libraries</td>
</tr>
<tr>
<td>Configuration Files</td>
</tr>
</tbody>
</table>

MirageOS compiler

<table>
<thead>
<tr>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCaml Runtime</td>
</tr>
<tr>
<td>Hypervisor</td>
</tr>
<tr>
<td>Firmware</td>
</tr>
</tbody>
</table>

no more OS!
User-space and OS libraries

“deployment mode (xen)”

mirage configure --target=xen --net=direct --dhcp=true
MirageOS
use-cases
BTC Piñata
1.1 MB application which has been running since 2015.

- Used to hold the key to 10 bitcoins
- A successful client or server TLS connection gives the key
- TLS is configured to use a secret certificate key
- If you guess (or steal) the certificate you get the BTC
NSDI 2015 paper

Boot-time of MirageOS application (few ms) is less than TCP RTTs

Can start MirageOS service on-demand, after the first network packet is received, with no perceived latency

Can even start an independent MirageOS service for serving every page of a website
Docker for Mac

MirageOS libraries used by millions of users

- Normally Docker use Linux namespaces and other Linux features

- on Mac: Docker daemon runs in a light Linux VM (using hypervisor.framework)

- on Mac: Docker client is a Mac application

- MirageOS libraries are used to translate semantics differences between platforms:
  - volumes: FUSE format + fsevent/inotify
  - network: Linux ethernet packets to MacOS network syscalls
Conclusion
Summary

- MirageOS is a modular operating system written in OCaml.

- End-goal is to allow individual libraries to be extracted/verified/certified individually and composed together.

- The only way to build end-to-end “high-insurance” services (application + full runtime environment)?
Future(s)

‣ More backends: IBM’s solo5 future looks great. Possibly some kind of “bare-metal” MirageOS targets?

‣ More trust:
  • more “rigorous engineering” approach (like ocaml-tls)
  • code extraction from theorem provers (like ocaml-btree)

‣ More libraries: today MirageOS is mainly focused on internet infrastructure, new distributed storage and security.

‣ New surprises (please!): ocaml-tls, solo5, … what’s next?
MirageOS in 2018

- research on MirageOS continues in Cambridge: IoT, embedded software, privacy-preserving systems, data-science, etc
  contact: Anil Madhavapeddy

- A non-profit organisation created in Berlin to work on secure infrastructure (internet services) using MirageOS
  contact: Hannes Mehnert

- A company created in Paris to commercially support MirageOS and to promote its use in the industry
  contact: Thomas Gazagnaire
Join the community!

mirageos-devel@lists.xenproject.org
https://mirage.io/
https://discuss.ocaml.org/tags/mirageos

6th MirageOS hack retreat

We invite you to participate in the sixth MirageOS hack retreat! The goal is to sync various MirageOS subprojects, start new ones, and fix bugs.

- When? 3rd October (arrival) – 10th October (departure) 2018
- Where? Marrakech, Morocco at Priscilla, Queen of the Medina.
- How much? likely 350 EUR*, accommodation and food (full board) included.
- How do I register? Register by sending a mail to retreat2018@nqsb.io by August 15th, 2018 including:
  - OCaml and MirageOS experience;
  - Project(s) you’re interested to work on; and
  - Dietary restrictions
- Who should participate? Everybody interested in advancing MirageOS.
- How big? We have only limited space (30 people). Selection will be done by various diversity criteria.
- How should I behave while there? Be kind and empathetic to others; do not harass or threaten anyone. If you make others unsafe, you may be asked to leave.

*: If you cannot afford this, please contact us directly (at retreat2018@nqsb.io).
Thank you!

Any questions?