FlexOS: Towards Flexible OS Isolation

Hugo Lefeuvre¹, Vlad-Andrei Bădoiu², Alexander Jung³,⁴, Stefan Teodorescu²,
Sebastian Rauch⁵, Felipe Huici⁶,⁴, Costin Raiciu²,⁷, Pierre Olivier¹

¹The University of Manchester, ²Politehnica Bucharest, ³Lancaster University, ⁴Unikraft.io,
⁵Karlsruhe Institute of Technology, ⁶NEC Labs Europe, ⁷Correct Networks

ASPLOS'22, 28th February - 4th March 2022
Current OS Designs

OS security/isolation strategies are **fixed** at design time!
Isolation granularity, underlying mechanisms, data sharing strategies (copy/share)
Current OS Designs

OS security/isolation strategies are **fixed** at design time!
Isolation granularity, underlying mechanisms, data sharing strategies (copy/share)

- **Micro/Separation kernels**
- **Monolithic kernels**
- **Single Address Space / Dataplane OSes**

- Removing user-kernel separation?
- Using hardware capabilities?
- Memory Protection Keys?
Decouple security/isolation decisions \textbf{from the OS design}

- Achieve a range of trade-offs instead of a single point in the design space.
- Support a range of isolation mechanisms and granularities.

Diagram:
- Security: Less secure to More secure.
- Performance: Slower to Faster.
- FlexOS trade-off area.
- Micro/Separation kernels.
- Monolithic kernels.
- Single Address Space / Dataplane OSes.
Other Use-Cases for Flexible Isolation

Deployment to heterogeneous hardware
Make optimal use of each machine/architecture's safety mechanisms with the same code

Quickly isolate vulnerable libraries
React easily and quickly to newly published vulnerabilities while waiting for a full patch

Incremental verification of code-bases
Mix and match verified and non-verified code-bases while preserving guarantees
FlexOS 101: Approach in 4 points

1. Focus on **single-purpose appliances** such as cloud microservices

   ...the more applications run together, the least specialization you can achieve
FlexOS 101: Approach in 4 points

1. Focus on **single-purpose appliances** such as cloud microservices

   **Full-system (OS+app) understanding of compartmentalization**

2. Not "only application" or "only kernel": consider everything and **specialize**

   Embrace the **library OS philosophy**: everything is a library... network stack, nginx, libopenssl, sound driver, etc.
FlexOS 101: Approach in 4 points

1. Focus on **single-purpose appliances** such as cloud microservices

   **Full-system** (*OS+app*) understanding of compartmentalization

2. **Abstract away** the technical details of isolation mechanisms

   Page table, MPK, CHERI, TEEs? Not the same guarantees, but a **similar interface can be achieved**.
FlexOS 101: Approach in 4 points

1. Focus on **single-purpose appliances** such as cloud microservices

2. **Full-system** (OS+app) understanding of compartmentalization

3. **Abstract away** the technical details of isolation mechanisms

4. Flexibility must not get into the way of performance
FlexOS 101: Overview

Happy users config.yaml

① Input config

FlexOS Toolchain

```
config.yaml

compartments:
- comp1:
  mechanism: intel-mpk
  default: True
- comp2:
  mechanism: intel-mpk
  hardening: [cfi, asan]

libraries:
- libredis: comp1
- libopenjpg: comp2
- lwip: comp2
```

"Redis image with two compartments, isolate libopenjpeg and lwip together"
FlexOS 101: Overview

1. Input config

2. Select isolation mechanism ("Backend")

3. Select libraries (kernel and app), rewrite, and statically put in compartments

4. Generate image with appropriate isolation properties

Isolation Backends
- MPK
- EPT
- ...

Core Libraries
- boot
- sched
- mm

Kernel & User Libs
- ramfs
- vfs
- ...

Happy users

Flexible Toolchain

Possible Image 1
- boot
- sched
- mm
- ...
- libopenjpeg

Possible Image 2
- boot
- sched
- mm
- libssl
- ...
- netdev

VMs
- comp1
- comp2
- comp3
FlexOS 101: Mechanism Abstraction

Based on a **highly modular LibOS design** (Unikraft)

Such libOSes are composed of fine-granular, independent libraries

Reuse libraries as finest granularity of compartmentalization

"Pre-compartmentalize" them

Cross-library **calls and shared data** are replaced by an **abstract construct** (gates, data sharing primitives)

Define them as part of the **FlexOS API**

At build time, these abstract constructs are replaced with a particular implementation by the toolchain. These implementations are defined by the **backends**.

---

Core Libraries

- boot
- sched
- mm

Kernel & User Libs

- nginx
- ssl
- jpeg
- ramfs
- vfs
- ...

---

MPK | VMs | TEEs | ...

---
FlexOS 101: Compartmentalization API

```c
int rc, connfd;
char buf[512];
/* ... */
rc = recv(connfd, buf, 512, 0);
```

**Porting**
- Add gate placeholders
- Annotate shared data

```c
int rc, connfd;
char buf[512] __attribute__((flexos_share));
/* ... */
rc = flexos_gate(liblwip, recv, connfd, buf, 512, 0);
```

**Automatic gate instantiation at build time**
- Replace with shared heap allocation
- Replace with MPK gate

```c
int rc, connfd;
char *buf[512] = shared_malloc(512);
/* ... */
rc = mpk_gate(0, 1, recv, connfd, buf, 512, 0);
```
### FlexOS 101: Compartmentalization API

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
</table>
| int rc, connfd;  
char buf[512];  
/* ... */  
rc = recv(connfd, buf, 512, 0); | Receive data from a connection. |
| int rc, connfd;  
char buf[512] __attribute__((flexos_share));  
/* ... */  
rc = flexos_gate(liblwip, recv, connfd, buf, 512, 0); | Annotate shared data. |
| int rc, connfd;  
char buf[512];  
/* ... */  
rc = recv(connfd, buf, 512, 0); | Porting and automatic gate instantiation at build time. |
| Porting | Add gate placeholders |
| Coccinelle | lwip + app |

**Porting:**
- **Annotate shared data**
- **Add gate placeholders**

**Automatic gate instantiation at build time:**
- Replace with shared heap allocation
- Replace with MPK gate
- Replace with normal stack allocation
- Replace with function call
Prototype

Implementation on top of Unikraft

Backend implementations for Intel MPK and VMs (EPT)

Port of libraries: network stack, scheduler, filesystem, time subsystem

Port of applications: Redis, Nginx, SQLite, iPerf server

This talk: focus on demonstrating flexibility and performance

more results in our paper 😊
Flexibility

Runtime performance with Redis in requests/s

One configuration and its associated performance (80 configurations in total)

FlexOS libraries used in the Redis image (only a subset for readability):
- Redis application
- C standard library (newlib)
- FlexOS scheduler (uksched)
- Network stack (lwip)

The color of boxes indicates the compartment:
- Compartment 1
- Compartment 2
- Compartment 3

The dot whether hardening (ASan, Safestack, etc.) is enabled:
- Hardening on
- Hardening off
Flexibility

① Large safety / performance space! (4x)

② Smooth slope, performance degrades gracefully

Hardening on  □  Hardening off  □  Compartment 1  □  Compartment 2  □  Compartment 3
Flexibility

Performance-wise:

```
 Compartment 1
 lwip

 Compartment 2
 ...

 Compartment 3
 uksched
```

2 crossings

```
 =
```

2 crossings

```
 Compartment 1
 lwip

 Compartment 2
 ...

 Compartment 3
 uksched
```

You can get some safety for free by exploring intelligently.

③ Similar performance, very different properties! need to reason about communication patterns, fast paths.

Hardening on  Hardening off  Compartment 1  Compartment 2  Compartment 3
Performance

Time to perform 10K SQLite INSERT queries in seconds

- QEMU/KVM
- Linux
- Process

Number of compartments and mechanism (e.g., PT2 = 2 compartments with the page table)

VMM/environment
Performance

No overhead when disabling isolation – you only pay for what you get
2. The MPK backend compares very positively to competing solutions.

Tricky comparison with CubicleOS - they're usinglinuxu, a Linux userland debug platform of Unikraft.
The EPT backend too compares positively to competing solutions.
Exploring the Design Space

Now, we've a nice framework!

We can leverage FlexOS to get the most secure image for a given performance budget!

Problem: some configurations are not comparable

How can we reason about security/performance trade-offs?
What we propose: consider configurations as a partially ordered set (poset)

Two configurations that do not share a path are simply not comparable.
Exploring the Design Space

We can then label each node with performance characteristics (in practice no need to label everything)
Exploring the Design Space

Based on this ordering and labeling we can choose the last node of each path that satisfies the performance constraints.

Curated list of optimal configurations

Let the user do the final choice

No need to evaluate everything!
Applying POSets on Redis

Reduction of 80 configurations to 5 candidates
In a Nutshell

There is a need for isolation flexibility
- OS Specialization, hardware heterogeneity
- or quickly react to vulnerabilities!

Current approaches: one isolation approach at design time

Decouple isolation from the OS design:
- Make isolation decisions at build time
- Explore performance v.s. security trade-offs
Interested?

Get in touch!

Webpage: https://project-flexos.github.io/
Pre-print of our ASPLOS'22 paper: https://arxiv.org/abs/2112.06566
By e-mail: hugo.lefeuvre@manchester.ac.uk

License: 3-Clause BSD License 😊