FlexOS: Easy Specialization of OS Safety Properties

FlexOS: Easy Specialization of OS Safety Properties

Hugo Lefeuvre
The University of Manchester
hugo.lefeuvre@manchester.ac.uk

Abstract

Modern operating systems are tightly coupled to a specific isolation approach and safety mechanism. At design time, the isolation strategy is set in stone and rarely revisited later, due to prohibitive costs. This lack of flexibility hurts specialization, makes it hard to leverage new software/hardware isolation technologies, and makes the OS less resilient to attacks targeting the isolation mechanism. To address these issues we have developed FlexOS, a novel libOS approach that decouples isolation properties from the OS design. Depending on the configuration, the same FlexOS code can mimic a microkernel with multiple address-spaces, a single-address-space OS with Intel MPK compartments, or many other OS isolation approaches.

In this paper, we summarize the current state of FlexOS and present two main research avenues that we aim to explore next: automated porting to make OS safety property specialization really easy, and support for CHERI hardware capabilities to better showcase FlexOS' potential.

CCS Concepts

• **Software and its engineering → Operating systems**; • **Security and privacy → Operating systems security**.

Keywords

operating systems, compartmentalization, operating systems security

ACM Reference Format:
Hugo Lefeuvre. 2021. FlexOS: Easy Specialization of OS Safety Properties. In *22nd International Middleware Conference Doctoral Symposium (Middleware '21 Doctoral Symposium), December 6–10, 2021, Virtual Event, Canada.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3491087.3493683

Middleware '21 Doctoral Symposium, December 6–10, 2021, Virtual Event, Canada

ACM ISBN 978-1-4503-9155-9/21/12…$15.00
https://doi.org/10.1145/3491087.3493683

1 Background and Motivation

Modern Operating Systems (OSes) are tightly coupled to one or a few specific isolation approaches and safety mechanisms. At design time, the OS developers settle for a particular set of isolation primitives and build OS services and interfaces around it. Because this decision is so deeply entangled with the structure of the OS, changing it at a later point is a very costly task.

The current OS landscape broadly consists of monolithic OSes [4] which rely on user/kernel separation using the page table, microkernels [12, 13] which rely on server-based isolation with page tables, and single-address-space OSes which rely on intra-address-space isolation [5, 11, 17], or ditch all form of isolation to maximize performance [14, 20, 23]. Clearly, in these models, changing the isolation approach or the safety mechanism after design represents a major refactoring and potentially maintenance effort. For example, the ongoing Rust for Linux patch already changes over 30K LoC [22] with tens of contributors. Kernel Mode Linux, which removes the user/kernel separation in Linux, touches over 20 subsystems, changes about 4K LoC [21], and breaks numerous kernel and user functionalities.

This lack of flexibility with regards to the isolation approach and safety mechanisms poses a number of problems. First, it prevents or limits specialization. Clearly, there is no one-size-fits-all approach that can offer optimal performance *and* safety characteristics for all applications and use-cases; each OS model effectively represents a single point in the design space that offers a specific trade-off with regards to safety and performance. Second, the lack of flexibility regarding safety mechanisms makes it hard to leverage the many new isolation technologies that come out both in academia and in the industry: leveraging memory protection keys for microkernels is everything but trivial [9], and so does using CHERI [25] hardware capabilities to isolate a microkernel [7] or a monolithic BSD system [6]. Even worse, this extensive redesign work has to be done again for every technology. Third, when primitives break (e.g., Meltdown [18] for user/kernel isolation), the lack of flexibility makes it harder to find a timely and efficient answer to the vulnerability. This problem has been noticed to some extent in the industry and in the literature; recently, a few OSes such as Genode [8] and CubicleOS [24] attempt to bring a certain degree of flexibility to the OS design space. Unfortunately, they remain bound to a specific isolation technology (e.g., page table or Intel Memory Protection Keys – MPK [1]).

In order to address these issues, we designed and implemented FlexOS [16], an OS approach that enables specialization of OS isolation properties, both in terms of granularity (what is isolated from what) as well as technology (how the isolation is provided). Unlike previous approaches, FlexOS does not represent a single point in the design space, instead, it can be flexibly specialized *at build-time* to adopt a wide range of isolation approaches by simply changing the build configuration. In this paper, we briefly introduce FlexOS and present two future research avenues that we envision to explore: (1) automated porting to improve on FlexOS' current porting cost and make it fit for more widespread use, and (2) CHERI hardware capabilities support to fully showcase FlexOS' flexibility in the granularity and technology dimensions.

2 FlexOS: Flexible OS Isolation

To enable the build-time customization of the isolation granularity and mechanism, FlexOS is based on a highly modular library OS (libOS), Unikraft [14]. Similarly to a component-based OS, it is composed of fine-granular, arbitrarily small and independent libraries that communicate via well-defined interfaces. Figure 1 summarizes the architecture of FlexOS. FlexOS defines *isolation backends* that implement isolation support for a particular technology (e.g. MPK [1], EPT, etc., Ⓐ in Figure 1), *core libraries* that implement core OS features that constitute the trusted computing base (such as the boot code or the scheduler, Ⓑ), and regular kernel and user libraries (such as the filesystem, or libnginx, Ⓒ).

Unlike Unikraft, where all libraries must share a single protection domain, FlexOS libraries are *isolation-agnostic*: the source code does not make any assumption about the isolation strategy in the final image. The ability to write code that is independent from the isolation profile is made possible by FlexOS' compartmentalization API. In FlexOS, the communication between libraries happens conceptually through *abstract gates* instead of usual function calls, and data shared across libraries must be allocated via *abstract shared data primitives*. At build time, and based on user configuration input, an isolation backend is selected (①), and these abstract primitives are instantiated by the compiler with the backend's implementation (②) to generate an image to reflects precisely the isolation technology and granularity chosen (③).

Since library implementations do not make any assumption about isolation, depending on the granularity and chosen technology, libraries can be isolated in their own compartment, or merged together with other libraries in a same compartment. The implementation of gates can also vary widely depending on the underlying technology. For instance, in the case of isolation with MPKs, gates will mainly switch the call stack and the protection key register, but in the case of microkernel-like VM-based isolation, gates will place arguments in shared memory and emit a remote procedure call (RPC).

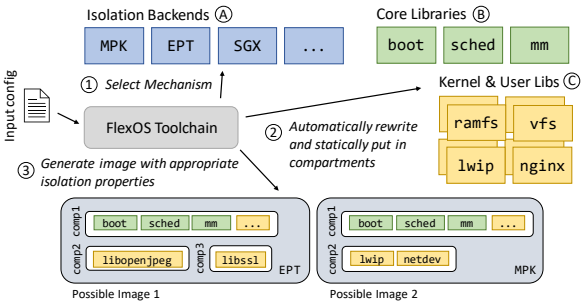

Figure 1: FlexOS overview.

We have implemented and evaluated a full-featured prototype of FlexOS with backend support for Intel MPK and EPT (VMs) — an early version of which was presented at *HotOS'21* [16]. The rest of this paper focuses on possible evolutions of this work.

3 Future Research

This section presents the two main research avenues that we aim to explore next: automated porting to make FlexOS more accessible to the wider community, and support for CHERI hardware capabilities to better showcase FlexOS' flexibility.

3.1 Automated Porting

As discussed in the previous section, FlexOS libraries are developed in an isolation-agnostic manner using a compartmentalization API. In practice, this means that libraries have to be ported to FlexOS, including not only kernel libraries but also userland libraries and applications. This manual porting cost is moderate — a few days for reasonably complex libraries. Although it is acceptable for a research prototype, this porting effort is still likely to hurt the adoption of FlexOS in the industry and leave the system prone to human errors. An important evolution of our work is therefore be to automate as much as possible this porting effort.

At the time of this writing, the porting of effort of FlexOS is twofold. First, *gate placeholders* have to be inserted. Concretely, this consists in replacing inter-library procedure calls by abstract gates, which will be later automatically replaced by the toolchain with a particular implementation (see ② in Figure 1). An example of gate placeholder is given below:

```
sd = socket(AF_INET, SOCK_STREAM, 0); // before porting
flexos_gate_r(fd, liblwip, socket, AF_INET,
              SOCK_STREAM, 0); // after porting
```

Doing such transformations is straightforward given the function call graph for direct calls, but significantly more complex for indirect calls (function pointers).

A second part of the porting effort consists in marking the allocation of data shared between multiple libraries with abstract shared data primitives. This means manually analyzing inter-library data flow and determine allocation sites

that must be annotated. Both are hard problems in the general case, as they can reduced to *points-to* analysis, which is undecidable [15]. It is also a general, open problem shared by all compartmentalization frameworks. In the literature, fully automated solutions have been proposed using static analysis such as PtrSplit and Cali [3, 19]. Other works also present semi-automated approaches using static and dynamic analysis, such as SOAAP [10].

Automated v.s. Semi-automated Approach Fully automated solutions have the advantage of offering seamless compartmentalization and significantly ease the adoption. On the other hand, they have to be conservative due to the fundamental imprecision of static analysis [3, 15]. Moreover, software that has not been developed with isolation in mind does not typically make a good candidate for seamless compartmentalization: if a large part of the state of the library or application is exposed in the form of shared data, then the security benefits of compartmentalization are limited and the performance impact might not be justified. Existing solutions are not able to automatically address such issues — and even if they could, expecting the developer to redesign the application to address them would be at odds with full automation. Such approaches are therefore incompatible with our goals of specialization and optimal performance. Hence, we envision an approach that features a feedback loop similar to SOAAP. In our case, however, this feedback loop would be optional.

In our approach, a static analysis pass similar to that of Cali would be applied to the libraries. The first aim would be full automation, potentially at the cost of precision, performance, and/or resource usage. However, unlike previous approaches, the static analysis would provide rich feedback to the user. Based on it, expert users could then take manual actions to increase the quality of the automatic compartmentalization by manually annotating the code or refactoring it. The porting could therefore be entirely automated, without closing the door to a feedback loop. Clearly, the goal is to get the best of both worlds: the automation of a static analysis tool, and the precision and bespoke character of a manual, expert approach.

Impact on the API The introduction of an automated compartmentalization framework would lead to the partial removal of the explicit API for an implicit one, and potentially to the introduction of additional compiler annotations to guide the compartmentalization and increase the quality of the isolation. Instead of having explicit abstract gates, cross-library function calls would implicitly translate to a gate at compile time. In the spirit of the semi-automated approach, the explicit shared data API would still be present but rather act as a guide for the static analysis approach. If data marked as shared is detected as library-local by the tool, then a warning is emitted, and the same the other way around.

3.2 Support for Hardware Capabilities

With MPK and EPT, we demonstrated that a unified, flexible system can seamlessly marry very heterogeneous technologies. Nevertheless this small number of backends has its limits from a research perspective as it (1) restrains the trade-off space that we can reach (performance, safety guarantees, number of domains), and (2) insufficiently demonstrates and challenges the flexibility of FlexOS' design. We identified CHERI [25] hardware capabilities as a good backend candidate to address these issues.

Motivation for CHERI support CHERI is an ISA extension that offers architectural support for capabilities. It enables for significantly smaller granularity of compartmentalization than MPK/EPT (byte level memory protection), and a theoretically unlimited number of protection domains. Adding CHERI backend support for FlexOS is interesting for this thesis for a number of reasons:

- CHERI capabilities would extend FlexOS' trade-off space with the ability to address confused-deputy situations, reduce data sharing, and larger numbers of domains, something that is currently impossible for architectural (MPK) and performance reasons (EPT).
- With the exception of CheriOS [7], little work has been done on single-address-space OSes on capability hardware. CheriOS proposes a single-address space microkernel utilizing CHERI for process and kernel isolation. Unlike FlexOS, CheriOS takes a general-purpose approach to single-address space OSes and targets exclusively CHERI, utilizing a custom security focused hypervisor to achieve a minimal TCB. It is an open question whether the concepts developed by CheriOS can be applied directly to FlexOS' architecture.
- The flexibility of FlexOS allows it to perform fair comparison of CHERI with other hardware and software-based mechanisms, something that has not been performed in the literature at the time of this writing.

The choice of CHERI as a FlexOS backend is timely; the CHERI ISA extension has recently been made available for ARMv8-A (an ISA that FlexOS supports) and Morello, a prototype board, will be released in January 2022 [2].

Engineering Approach Ultimately the goal would be to port FlexOS to be able to run in full capability mode, i.e. with every pointer in the code backed by a capability used to check boundaries and access rights upon dereferencing. However, a number of issues might arise due to the limited compatibility with certain C idioms commonly used in low level systems software such as OSes. Hence, we envision a transitional goal through the use of CHERI's hybrid capability mode, in which only select pointers are protected with capabilities. We plan to leverage CHERI's Default Code Capability (DCC) and Default Data Capability (DDC), defining the range of virtual address

space accessible by the currently running thread, to enforce isolation between compartments. Data will be shared safely and efficiently between communicating compartments by protecting the corresponding pointers with capabilities.

In addition to this, we expect that additional changes will be required in core subsystems such as scheduling and memory management. These should eased by FlexOS' *hook API* that lets isolation backends perform relevant operations upon events such as thread creation or context switches. In general, the characteristics of the CHERI ISA seem to fit well with FlexOS' API.

4 Summary

We have briefly introduced FlexOS, an OS approach that enables specialization of OS isolation properties. Based on this we presented future research avenues to extend it: automatic porting to make FlexOS it for more widespread use, and CHERI hardware capabilities support to better showcase its flexibility. In addition to these, we also envision to better explore new practical use-cases enabled by FlexOS.

Acknowledgments

I would like to thank my supervisor Pierre Olivier and my mentors Felipe Huici and Costin Raiciu, and Vlad Bădoiu, one of the main contributors of the broader FlexOS effort. I would also like to thank the anonymous reviewers for their insights. This work was partly funded by a studentship from NEC Laboratories Europe, EU H2020 grant agreements 825377 (UNICORE), 871793 (ACCORDION) and 758815 (CORNET), as well as the UK's EPSRC grant EP/V012134/1.

References

[1] [n. d.]. Intel® 64 and IA-32 Architectures Software Developer's Manual. Volume 3A, Section 4.6.2.

[2] 2021. ARM Morello Platform Open Source Landing Page. https://www.morello-project.org/ Online; accessed Sep, 29 2021.

[3] Markus Bauer and Christian Rossow. 2021. Cali: Compiler Assisted Library Isolation. Association for Computing Machinery.

[4] Daniel P. Bovet and Marco Cesati. 2005. *Understanding the Linux Kernel, 3rd Edition* (3rd ed.). O'Reilly Media, Inc.

[5] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. 1994. Sharing and Protection in a Single-Address-Space Operating System. *ACM Trans. Comput. Syst.* 12, 4 (Nov. 1994), 271–307.

[6] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, James Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, and Jonathan Woodruff. 2019. CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-Time Environment *(ASPLOS '19)*. 379–393.

[7] Lawrence G. Esswood. 2021. *CheriOS: designing an untrusted single-address-space capability operating system utilising capability hardware and a minimal hypervisor.* Ph. D. Dissertation. University of Cambridge. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-961.pdf

[8] Norman Feske. 2021. Genode Foundations. https://genode.org/documentation/genode-foundations-21-05.pdf.

[9] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. 2020. Harmonizing Performance and Isolation in Microkernels with Efficient Intra-kernel Isolation and Communication *(USENIX ATC'20)*. USENIX Association, 401–417.

[10] Khilan Gudka, Robert N.M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. 2015. Clean Application Compartmentalization with SOAAP *(CCS '15)*. 1016–1031.

[11] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. 1999. The Mungi Single-Address-Space Operating System. *Software: Practice and Experience* 28, 9 (July 1999), 901–928.

[12] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. 2006. MINIX 3: A Highly Reliable, Self-Repairing Operating System. *SIGOPS Oper. Syst. Rev.* 40, 3 (July 2006), 80–89.

[13] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. SeL4: Formal Verification of an OS Kernel *(SOSP '09)*. 207–220.

[14] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. 2021. Unikraft: Fast, Specialized Unikernels the Easy Way *(EuroSys '21)*. 376–394.

[15] William Landi. 1992. Undecidability of Static Analysis. *ACM Letters on Programming Languages and Systems* 1, 4 (Dec. 1992), 323–337.

[16] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Stefan Teodorescu, Pierre Olivier, Tiberiu Mosnoi, Răzvan Deaconescu, Felipe Huici, and Costin Raiciu. 2021. FlexOS: Making OS Isolation Flexible *(HotOS '21)*.

[17] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. 1996. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications* 14, 7 (1996), 1280–1297.

[18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. 2018. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security '18)*. 973–990.

[19] Shen Liu, Gang Tan, and Trent Jaeger. 2017. PtrSplit: Supporting General Pointers in Automatic Program Partitioning *(CCS '17)*. New York, NY, USA, 2359–2371.

[20] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud *(ASPLOS '13)*. Association for Computing Machinery, 461–472.

[21] Toshiyuki Maeda. 2015. Kernel Mode Linux : Execute user processes in kernel mode. http://web.yl.is.s.u-tokyo.ac.jp/~tosh/kml/ Online; accessed Sep, 28 2021.

[22] Miguel Ojeda. 2021. Rust support. The Linux Kernel Mailing List archive. https://lkml.org/lkml/2021/7/4/171 Online; accessed Sep, 28 2021.

[23] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. 2019. A Binary-Compatible Unikernel *(VEE 2019)*. New York, NY, USA, 59–73.

[24] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. 2021. CubicleOS: A Library OS with Software Componentisation for Practical Isolation *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 546–558.

[25] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. 2015. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 20–37.