FlexOS: Making OS Isolation Flexible

Hugo Lefeuvre
The University of Manchester

Vlad-Andrei Bădoiu
University Politehnica of Bucharest

Ștefan Teodorescu
University Politehnica of Bucharest

Pierre Olivier
The University of Manchester

Tiberiu Mosnoi
University Politehnica of Bucharest

Răzvan Deaconescu
University Politehnica of Bucharest

Felipe Huici
NEC Laboratories Europe GmbH

Costin Raiciu
University Politehnica of Bucharest

Abstract

OS design is traditionally heavily intertwined with protection mechanisms. OSes statically commit to one or a combination of (1) hardware isolation, (2) runtime checking, and (3) software verification early at design time. Changes after deployment require major refactoring; as such, they are rare and costly. In this paper, we argue that this strategy is at odds with recent hardware and software trends: protections break (Meltdown), hardware becomes heterogeneous (Memory Protection Keys, CHERI), and multiple mechanisms can now be used for the same task (SFI, verification, HW isolation, etc). In short, the choice of isolation strategy and primitives should be postponed to deployment time.

We present FlexOS, a novel, modular OS design whose compartmentalization and protection profile can seamlessly be tailored towards a specific application or use-case at build time. FlexOS offers a language to describe components' security needs/behavior, and to automatically derive from it a compartmentalization strategy. We implement an early prototype of FlexOS that can automatically generate a large array of different OSes implementing different security strategies.

CCS Concepts

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

ACM Reference Format:
Hugo Lefeuvre, Vlad-Andrei Bădoiu, Ștefan Teodorescu, Pierre Olivier, Tiberiu Mosnoi, Răzvan Deaconescu, Felipe Huici, and Costin Raiciu. 2021. FlexOS: Making OS Isolation Flexible. In *Workshop on Hot Topics in Operating Systems (HotOS '21), May 31–June 2, 2021, Ann Arbor, MI, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3458336.3465292

HotOS '21, May 31–June 2, 2021, Ann Arbor, MI, USA

ACM ISBN 978-1-4503-8438-4/21/05…$15.00
https://doi.org/10.1145/3458336.3465292

1 Introduction

To create secure and fast software, programmers can use three main approaches offering various trade-offs between human effort, safety guarantees and runtime performance: software verification, runtime checking and hardware isolation. Today's software statically commits to one or a combination of these approaches. At design time, systems are built around the protection ensuing from these mechanisms; changing them after deployment is rare and costly.

In operating systems, the current landscape (illustrated on Figure 1) broadly consists of micro-kernels [24, 29], which favor hardware protection and verification over performance, monolithic kernels [8], which choose privilege separation and address spaces to isolate apps, but assume all kernel code is trusted, and single-address-space OSes (SASOSes), which attempt to bring isolation within the address space [10, 23, 32], or dump all protection for maximum performance [30, 36, 43].

OS implementations are heavily interlinked with the protection mechanisms they rely upon, making changes to them difficult to implement. For instance, removing user/kernel separation [37] requires a lot of engineering effort, as does breaking down a process into multiple address spaces for isolation purposes [27]. This is the case despite the fact that alternative approaches can be used to provide the same guarantees. First, verification and software fault isolation (SFI) can help ensure memory isolation between separate components even if they run in the same address space, thus avoiding the need for hardware isolation [19, 52]. Second, hardware isolation, SFI and runtime property checking can be used to check that certain correctness properties hold (e.g. when specified as pre and post conditions), thus relieving the user from needing to prove code correctness statically against a specification [47]. Third, both software verification and protection domains can be used to ensure (a form of) control-flow integrity between components, guaranteeing that code execution starts only at

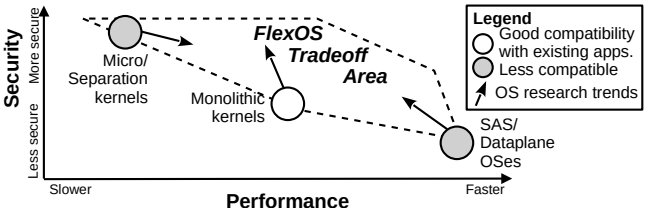

Figure 1: Design space of OS kernels.

well-defined entry points, without needing software runtime checks [53].

The rigid use of safety primitives in modern OSes poses a number of problems. First, when the protection offered by hardware primitives breaks down (e.g. Meltdown), it is difficult to decide how they should be replaced, and with what costs. In cases where multiple mechanisms can be used for the same task (e.g. SFI and verification), choosing the primitive that provides the best performance depends on many factors such as the hardware, the workload, etc., and should ideally be postponed to deployment time, not design time. Locking the design to a certain isolation primitive will result in poor performance in many scenarios.

Second, computer hardware is becoming heterogeneous [54] and certain primitives are hardware-dependent (e.g. Intel Memory Protection Keys – MPK [12]). When running the same software on different hardware, how can we minimize the porting effort while preserving safety?

Software modularization should, in principle, provide better robustness and security. Most software, including OSes, integrates modules from different sources, with various levels of trust. Unfortunately, the isolation primitives assumed by the module designers affect the way in which a module can be used, limiting its usefulness. Take, for instance, a formally verified OS subsystem: how does one go about embedding it into a larger project while still maintaining its safety properties? Clearly, if one embeds this component alongside untrusted C code, its verified properties may not hold in practice.

This leads us to the following research problem: *How can we enable users to easily and safely switch between different isolation and protection primitives at deployment time, avoiding the lock-in that characterizes the status-quo?*

Our answer is *FlexOS*, a novel, modular OS design whose compartmentalization and protection profile can easily *and cost-efficiently* be tailored towards a specific application or use-case at build time, as opposed to design time as it is the case today. To that aim, we extend the Library OS model (LibOS) and augment its capacity to be specialized towards a given use case, historically done for performance [18, 26], towards the *security* dimension.

With FlexOS, the user can decide at build time which of the fine-grained OS components should be compartmentalized, as well as how to instantiate isolation and protection primitives for each compartment. FlexOS allows developers to easily explore the trade-offs than can be achieved with different isolation technologies and granularities, and to select the best security/performance profile for their use case. Concretely, our research contributions are:

- The design and implementation of *FlexOS*, a novel framework for effortlessly investigating performance vs. security trade-offs in operating systems.
- The identification of fundamental primitives that are needed in order to provide isolation and protection via a wide range of software and hardware-based mechanisms.
- A preliminary evaluation showing how FlexOS can be used to explore a wide array of security/performance profiles for two apps: iperf and Redis.

2 Design Overview

The goal of FlexOS is to allow developers and OS researchers to easily inspect and select different points in the security vs. performance trade-off space. Exploring such a space is far from trivial, and our aim is also to automate such exploration. Here, various strategies could be followed:

- Given a performance target and a set of predefined compartments (e.g. isolate the application and the network stack from everything else), find the combination of isolation primitives that maximizes security within a certain performance budget.
- Given a set of safety requirements (e.g. no buffer overflows), find a compliant instantiation that yields the best performance or that can run on the largest number of devices (based on the availability of hardware-based mechanisms).

Both objectives above have in common the need to describe the security attained by each mechanism, and the implications of running one software component in the same compartment as another one.

We base our design as an extension of the LibOS model [18], since LibOSes are by nature divided into fine-grained components/libraries. Our approach consists in supporting a set of hardware and software isolation mechanisms, and complementing the API of each such library with FlexOS metadata specifying 1) the expected memory access behavior of other components running in the same compartment as the library for its safety properties to hold; 2) the areas of memory this library can access in normal but also adversarial operation (for example if the library's execution flow is hijacked); and 3) API specific information.

Such metadata are created manually for each library by its developer, a one-time and relatively low effort for the library's author. The metadata purpose is to capture the effects upon the overall safety properties of running this library alongside other libraries in the same or in a different compartment.

For instance, here is an example describing FlexOS' formally verified scheduler that we have implemented in Dafny [31]:

```
[Memory access] Read(Own,Shared); Write(Own,Shared)
[Call] alloc::malloc, alloc::free
[API] thread_add (. . . ); thread_rm(. . . ); yield(. . . )
[Requires] *(Read,Own), *(Write,Shared),
            *(Call, thread_add), *. . .
```

The description concisely specifies that (1) the library accesses its own memory and a segment shared with other libraries (e.g. its callers), that (2) it only uses functions provided by the memory allocator, (3) which functions it exposes as its API, and that (4) it expects other libraries to be able to read its own memory (but not write to it) and be able to write in shared memory.

Consider now a component written in an unsafe language, such as C, that is deemed potentially unsafe (perhaps due to variable-length writes to a buffer that cannot be proven safe statically); its description will read:

```
[Memory access] Read(*); Write(*)
[Call] *
```

This specification simply outlines that the control/data flow of this component may be hijacked at runtime, resulting in arbitrary code execution/memory access. Since there is no `Requires` clause, this means other libraries should not be prevented from writing to memory owned by this library.

Given two libraries and their metadata, we now have enough information to automatically decide whether they can run in the same compartment. If both libraries have no `Requires` clause, the answer is yes. If any of the libraries has such clauses, each clause can be automatically checked in the presence of the other library. In our example above, for its verified properties to hold, the scheduler expects others to only read, not write, to its own memory. The C component, on the other hand, could write to all memory it has access to (in its compartment) - thus breaking the expectation: as a result, these two libraries cannot be run in the same compartment.

Armed with information about pair-wise incompatibility, selecting the smallest number of compartments in a FlexOS image can be reduced to the classical graph coloring problem: each library is a vertex, and an edge connects two incompatible libraries. Graph coloring assigns the smallest number of colors to the vertices of a graph such that no two adjacent vertices have the same color. For each color, we will instantiate a separate compartment that holds the libraries that have been painted with that color. In the worst case where all libraries have conflicts, each library will be instantiated in its own compartment.

When to Enable SFI? In certain cases, it is preferable from a performance or deployment point of view to use runtime checks (CFI [2], DFI [3, 9], etc., grouped in the rest of this paper under the *SFI* term) instead of multiple compartments – possibly only for a subset of the system/compartments.

To automate the process of selecting SFI mechanisms, we first create in FlexOS a machine-readable description of the impact each SFI technique has on the safety behavior of a library. This is a transformation that takes as input a library definition and outputs a changed definition describing the safety behavior of the library when the SFI technique is enabled.

For control-flow integrity, the transformation is simple: libraries that previously declared `Call(*)` are transformed into `Call(func. list)` where the list of functions is populated via a standard control-flow analysis of the library. For data-flow integrity, the transformation is similar: if the data flow graph of a library shows that all its writes are to its own data, `Writes(*)` will be transformed to `Writes(Own)`; other SFI techniques are handled similarly. To enumerate feasible deployments with SFI, we proceed as follows: 1) for each library that writes to all memory, enable DFI / ASAN; 2) for each library that can execute arbitrary code, enable CFI.

The result of this step will be a list of libraries that have two versions: one with SFI, and one without. We then iterate through all combinations of such library versions and run the graph coloring algorithm described above. This will result in as many colorings as there are possible combinations of libraries. Consider our example above: the unsafe C library will have two versions now, one with SFI and one without SFI. When put together with the scheduler in the same image, the SFI version will be able to share a compartment with the scheduler, while the original version will require a separate compartment.

Handling pre and post conditions. The approach we took to handling memory access requirements could also be applied to pre-conditions that certain API functions may request to be true when called. For instance, in the case of the scheduler, one of `thread_add`'s preconditions is to not add a thread that has already been added. In such cases, FlexOS could be extended to automatically check whether the pre-conditions always hold on call (based on a static analysis of the call graph); if they don't, runtime checks should be added to ensure they do hold. In our current prototype, we add these checks manually in our scheduler code; in future work we intend to explore ways of deriving this automatically.

FlexOS Architecture. FlexOS is based on a modular LibOS (Unikraft, a unikernel framework [30]) and allows fined-grained OS software modules to be placed in compartments (see Figure 2). Note that the granularity of such modules is much more fine-grained than that of traditional microkernel/multi-server OSes. Compartments in FlexOS are separated via *gates* which are made up of the API each compartment exposes. The gates also implement isolation between compartments, and

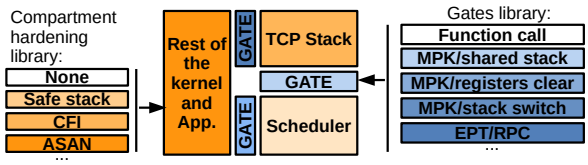

Figure 2: FlexOS architecture. Gates isolate an arbitrary number of compartments using a wide set of software and hardware-based security mechanisms.

can leverage different isolation mechanisms depending on the available hardware (e.g. protection keys [12, 14], capabilities [55]) or software (e.g. CFI or ASAN).

Gates are instantiated at link time based on the requirements provided by the user or automated tools. Implementations vary from cheap function calls all the way to expensive RPC across VM boundaries. Depending on the chosen gates, the compartments will be running in the same protection domain or in different ones. Further, each compartment can be individually hardened by using SFI without code changes.

FlexOS leverages Unikraft's micro-library granularity (e.g. a scheduler, a memory allocator or a message queue are all micro-libs) but replaces each micro-lib's standard function-call based API with call gates. In the porting process, developers replace cross-micro-libs function calls with gate placeholders. Once replaced by a particular implementation in the linking stage, gates take care of executing the function call in the foreign compartment, and of copying the return value back. Programmers do not need knowledge of the internal functioning of gates; all gates are exposed by the same, simple API:

```
rc = listen(sockfd, 5);           // before porting
uk_gate_r(rc, listen, sockfd, 5); // after porting
```

Programmers also annotate data shared with other micro-libs so that they are allocated in shared areas according to the compartmentalization graph.

FlexOS's build system extends Unikraft's to allow specifying how many compartments the resulting image should have, how they should be isolated, and whether SFI techniques should be applied to one or multiple of these. Using this information, FlexOS's builder will generate the required protection domains (one per compartment) and replace the call gate placeholders with the relevant code. For libraries in the same compartment, it will replace the call gates with direct function calls. For inter-compartment crossings, it will use the appropriate gate for switching protection domains: in our example in Figure 2, we have three separate compartments.

Using these basic primitives, a developer will be able to easily experiment with various isolation techniques to find the fastest implementation for a given task. We show, via experiments in §4, how we can create several networking images that do not trust the networking stack, with vastly different performance and security characteristics.

3 Implementation Prototype

To demonstrate its practicality, we implement a prototype of FlexOS on top of Unikraft v0.4 [49] in 1.5K LoC. Gate support is provided by two isolation mechanisms, referred to as isolation *backends*: Intel MPK and VM (EPT) isolation. SFI support is available with CFI, ASAN, etc. We ported a subset of the Unikraft micro-libraries to FlexOS, manually created compartment specifications and identified shared data to showcase the trade-offs FlexOS enables; implementing the automated approach to defining compartments is left as future work. Note that although we focused on virtualized environments for this prototype, nothing fundamentally precludes FlexOS to run as a bare-metal OS.

Intel MPK Backend. Intel MPK is a mechanism providing low-overhead intra-address space memory isolation [1, 5, 46] at the granularity of a page. Our MPK backend places each compartment in its own MPK memory region, including static memory, heap, stack, and TLS. MPK permissions for the thread executing on a core are held in a register named PKRU. Since any compartment can modify its value, the MPK backend has to prevent such unauthorized writes; it can do so via static analysis [50], runtime checks [22] or page-table sealing [36].

In addition, the MPK backend introduces isolation requirements for the scheduler and the Memory Manager (MM): the scheduler holds the value of the PKRU for threads that are not currently running, and so its memory is as critical as the PKRU register itself. The MM's domain includes the page-table holding the mapping between pages and protection domains. This implies that the scheduler and MM have to be trusted when using MPK. In our implementation we use a provably correct scheduler implemented in Dafny, and we can also use SFI to harden schedulers/MMs implemented in C.

Our MPK backend supports two types of gates. In the *shared-stack* gate, heap and static memory are isolated and only shared data is accessible from all compartments in dedicated heap/static memory segments. Thread stacks are located in a domain shared by all compartments. This gate is similar to ERIM's [50]. With the *switched stack* gate, the heap, stacks, and static memory are all isolated. There is one stack per thread per compartment and the stack is switched at domain boundaries. Parameters are copied to the target domain stack, and shared stack data is placed on a shared heap. This gate is similar to HODOR's [22].

VM-based Backend. Many works use virtualization to support isolation within a kernel [33, 40, 41, 56]. VM-based isolation provides strong security guarantees and is widely supported, at the cost of higher overhead.

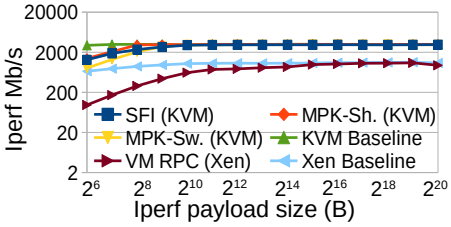

Figure 3: iperf throughput, various configs (*Sh* - shared, *Sw*- switched stacks)

Component *C*	SFI: all but *C*	SFI: *C* only
Scheduler	496Mb/s	2.90 Gb/s
Network stack	631Mb/s	2.76 Gb/s
LibC	1.47Gb/s	1.25 Gb/s
Rest of the system	1.08Gb/s	2.50 Gb/s
Entire system	2.94Gb/s (baseline)	489 Mb/s

Table 1: iperf throughput with SFI on various components.

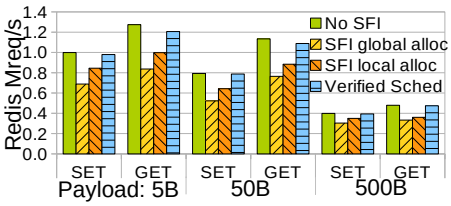

Figure 4: Redis throughput for various SFI configs and our verified scheduler.

Our toolchain generates one VM image per compartment. Images contain the minimum set of micro-libraries necessary to run the VM independently (platform code, memory allocator, scheduler), along with a thin RPC implementation based on inter-VM notifications and a shared area of memory for shared heap/static data. It is mapped in all compartments (VMs) at an identical address so that pointers to/in shared structures remain valid. Compartments do not share a single address space anymore, and run on different vCPUs. Hence, each compartment needs its own memory allocator and scheduler, so these have to be trusted. Our VM-based isolation backend is currently based on Xen, with KVM support underway.

SFI Support. FlexOS's SFI support is modular: we can apply hardening mechanisms per compartment (not system-wide), allowing for fine-grained protection and performance trade-offs. For example it is possible to apply SFI only to components that interact directly with the outside world, such as the network stack. Our implementation supports KASAN, Stack protector and UBSAN on GCC, and CFI and SafeStack under clang. A key requirement for SFI is the ability to have a separate memory allocator per compartment: as many SFI techniques instrument malloc, using a single global allocator would result in the entire system paying the cost of the instrumented allocator. FlexOS can be configured to use separate memory allocators per compartment to avoid such overheads when only a subset of compartments are hardened.

4 Preliminary Results

We studied the performance impact brought by a number of FlexOS' configurations for two apps: an iperf server and Redis. We aim to confirm that FlexOS allows easy exploration of a wide design space of security/performance trade-offs: each configuration is obtained by setting a few options and recompiling the LibOS against the app sources. Both apps were manually ported to the prototype, though most of this process should be easy to automate. Experiments were run on a Xeon Silver 4110 (2.1 GHz), with KVM and Xen.

Safe iperf. In our first test, we created an iperf server where an untrusted network stack is isolated from the rest of the OS image. We test three configs: 1) two compartments with MPK, one for the stack and one for rest of the OS; 2) separate VMs for the two compartments; 3) A single compartment, with SFI applied only to the network stack.

Performance results as measured by an iperf client are shown in Fig. 3. At the server side, we vary the size of the buffer passed to `recv`. With SFI and MPK, for small buffers there is a non negligible slowdown (2x to 3x). However, these solutions catch up quickly to the baseline, yielding similar performance starting at 1KB buffer size. Xen's numbers are lower due to Unikraft not being optimized for this hypervisor; still, we observe that the payload needs to be larger for the VM backend to catch up to the baseline, 32KB, due to increased domain switching costs. These results show that the performance impact of various protection mechanisms depends on the workload, so locking into a protection mechanism at design time is suboptimal.

iperf: Fine-Grained SFI. FlexOS' modular design allows us to enable/disable SFI at micro-library granularity. We ran iperf with a variable number of FlexOS' components running with SFI: the network stack, the scheduler, the standard C library (LibC), and the rest of the system including iperf itself.

Results are in Table 1. The performance impact strongly depends on the component running with SFI: the scheduler brings a 1% overhead while the LibC has a 2.3x slowdown. Interestingly, the slowdown with SFI for the network stack is low (6%). SFI for the entire system has a 6x slowdown, demonstrating the benefits of FlexOS' flexibility, useful in scenarios where components have variable levels of trust and variable performance impact when protected with SFI.

Redis: Isolation Strategies. We ran Redis in various scenarios. We defined 4 compartmentalization models: {`NW stack, rest of the system`} (NW only), {`NW stack, scheduler, rest of the system`} (NW/sched/rest), {`NW stack + scheduler, rest of the system`} (NW and sched/rest), and a baseline with no isolation. These demonstrate FlexOS' capacity to seamlessly manage various trust models. For MPK we ran both the shared and switched stack versions.

The results are in Figure 5. The isolation overhead depends on the number of compartments and how they communicate. Isolating only the network stack brings on average a 17% slowdown, while also isolating the scheduler brings a 1.4x (shared stack) and 2.25x (switched stack) slowdown – an increase due

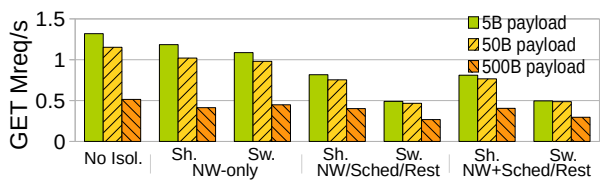

Figure 5: Redis throughput with MPK isolation.

to the stack switch overhead. This points to frequent communication between the scheduler and the network stack, making intensive use of wait queues through semaphores. However, putting the network stack and the scheduler in the same compartment does not increase performance: this is due to semaphores being implemented in another compartment (LibC). This brings the need for further compartmentalization or redesign of the components. Similar to iperf, the isolation overhead drops significantly when the request size increases.

Redis: SFI. We ran Redis enabling SFI for the network stack with 1) a global allocator for the entire system and 2) a dedicated local allocator for the network stack and another for the rest of the system. The results are in Figure 4. With a global allocator, the slowdown from running the network stack with SFI is on average 1.45x. FlexOS' capacity to easily setup a local allocator for the network stack allows us to reduce that overhead to a 1.24x slowdown. Overall, the results for Redis show that FlexOS can manage a wide range of security/performance requirements scenarios.

Verified Scheduler. We developed a verified cooperative scheduler written in Dafny [31]; the scheduler's safety is given by pre- and post-conditions that are statically proven to hold by Dafny. We generate C++ code from the scheduler and integrate it in FlexOS by adding glue code. How can we embed this safely alongside untrusted code? To protect the scheduler's memory from external writes we can either apply SFI to the rest of the unikernel or use MPK. To check that pre-conditions hold on call we integrate the checks in the glue code, and disable interrupts. In future work we will generate glue code automatically.

The context switch latency of our verified scheduler is 218.6ns, 3x slower than the C scheduler (76.6ns). This is fairly high, but Fig. 4 shows that the verified scheduler's overhead over the C one is always below 6% for Redis.

5 Open Questions

Decoupling OSes isolation and safety primitives from their fundamental design brings a number of challenges.

How to minimize porting effort? FlexOS requires porting, not only for kernel-internal libraries, but also for external user-space libraries. This porting process usually boils down to identifying shared data and handling indirect cross-component calls, a common effort among isolation frameworks [21, 38]. While it is a one-time, reasonably inexpensive effort, we recognize that it might hinder the adoption of our approach [43]. Further, manual approaches are not fail proof and can result, for example in over- or under-sharing data. To address these issues, automated porting techniques, mostly explored at the user-space level [6, 35, 48], can be explored.

Another element of the porting process is the writing of per-library metadata. These metadata are used by the design space exploration tool to automatically derive a compartmentalization strategy. The tool is then able to guarantee that properties hold according to the specified characteristics of each component. The resulting kernel is guaranteed to be correct as long as the metadata themselves are correct. But who verifies the specification/metadata? The process of writing metadata is error prone, and methods for (semi-)automatically generating them should be explored.

Isolation alone is not enough. Traditional system call APIs are designed from the outset as a trust boundary. Not only are they copy-based and carefully check function arguments, they are also designed as to avoid more subtle privilege escalation vulnerabilities, e.g. confused deputies. For such APIs, swapping the isolation mechanism (e.g., from standard system calls to MPK domain switching) is relatively straightforward. On the other hand, when the API was previously developed without a trust model (as is the case with all kernel internal APIs, but also userland library APIs), introducing isolation is a more complex task; isolation alone is not enough, and in order to provide protection against a wide range of attacks, APIs have to be carefully revisited [11]. Further, in the case of FlexOS, we only want to execute such checks when they are really needed, depending on the instantiated kernel configuration: if component A is together with component B in the same trust domain, then checks are not necessary, but they are when component C (in another domain) calls component B.

A possible approach to tackle this problem is the one that we envision to take for preconditions: by enriching all microlibraries with API metadata, the build system could possess sufficient information to automatically generate wrappers that would include or exclude these checks on-demand.

6 Related Work

Previous work addressed the isolation inefficiencies of monolithic kernels by reducing the TCB through separation [4, 44] and micro-kernels [20, 24]. More recently, OSes providing security through software isolation brought by safe languages [7, 13, 25, 36, 39] have been proposed. In SASOSes, isolation has been provided with traditional page tables [10, 23, 32] and recently through intra-address-space hardware isolation mechanisms [34, 42, 45, 47]. Formal verification offers deterministic security guarantees, but has trouble scaling to modern OSes' large codebases [28, 29]. Low-overhead runtime protection mechanisms are commonly found in production kernels [15, 17]. However, the most security-efficient

ones [16] are only enabled for test runs [51] due to their high performance impact.

In all, each of these approaches represents a single point in the OS design space and lacks the flexibility of FlexOS to automatically configure variable, fine-grained security/performance profiles. LibrettOS [41] does allow a LibOS to switch between SASOS and microkernel modes, but remains limited to a small subset of the security/performance design space. SOAAP [21] proposes a system to explore software's compartmentalization space using static/dynamic analysis; however, this work targets *monolithic user-space* code-bases, as opposed to *modular kernel* code-bases for FlexOS.

7 Conclusion and Future Work

FlexOS provides developers the ability to mix and match isolation primitives, be they hardware or software, which allows creating tailor-made versions of the same app for target workloads, with good performance and improved security, as our experiments have shown for two apps.

This paper is only an initial exploration of the potential benefits of FlexOS. Our future work aims to automate checking the safety of a proposed configuration, and searching for configurations with desired properties automatically. This will in turn enable developers to build robust software by mixing and matching components with various trust levels.

Acknowledgments

We would like to thank the anonymous reviewers for their comments and insights. A special thanks goes to Julia Lawall for her help on Coccinelle. This work was partly funded by EU H2020 grant agreements 825377 (UNICORE), 871793 (ACCORDION) and 758815 (CORNET), VMWare gift funding for UPB, as well as the UK's EPSRC New Investigator Award 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] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (Alexandria, VA, USA) *(CCS '05)*. Association for Computing Machinery, New York, NY, USA, 340–353. https://doi.org/10.1145/1102120.1102165

[3] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. 2008. Preventing Memory Error Exploits with WIT. In *2008 IEEE Symposium on Security and Privacy*. 263–277. https://doi.org/10.1109/SP.2008.30

[4] J. Alves-Foss, P. Oman, C. Taylor, and S. Harrison. 2006. The MILS architecture for high-assurance embedded systems. *Int. J. Embed. Syst.* 2 (2006), 239–247.

[5] Steve Bannister. 2019. Memory Tagging Extension: Enhancing memory safety through architecture. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/enhancing-memory-safety Online; accessed October 27, 2020.

[6] Markus Bauer and Christian Rossow. 2021. Cali: Compiler Assisted Library Isolation. In *Proceedings of the 16th ACM Asia Conference on Computer and Communications Security (ASIA CCS'21)*. Association for Computing Machinery.

[7] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. 2020. Theseus: an Experiment in Operating System Structure and State Management. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, 1–19. https://www.usenix.org/conference/osdi20/presentation/boos

[8] Daniel P Bovet and Marco Cesati. 2005. *Understanding the Linux Kernel: from I/O ports to process management*. O'Reilly Media, Inc.

[9] Miguel Castro, Manuel Costa, and Tim Harris. 2006. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation* (symposium on operating systems design and implementation (osdi) ed.) *(OSDI'06)*. USENIX. https://www.microsoft.com/en-us/research/publication/securing-software-by-enforcing-data-flow-integrity/

[10] 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. https://doi.org/10.1145/195792.195795

[11] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. 2020. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security'20)*. USENIX Association, 1409–1426. https://www.usenix.org/conference/usenixsecurity20/presentation/connor

[12] Jonathan Corbet. 2015. Memory protection keys. *Linux Weekly News* (2015). https://lwn.net/Articles/643797/.

[13] Cody Cutler, M Frans Kaashoek, and Robert T Morris. 2018. The benefits and costs of writing a POSIX kernel in a high-level language. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 89–105.

[14] Leila Delshadtehrani, Sadullah Canakci, Manuel Egele, and Ajay Joshi. 2021. Efficient Sealable Protection Keys for RISC-V. (2021).

[15] Jack Edge. 2013. Kernel address space layout randomization. https://lwn.net/Articles/569635/.

[16] Jack Edge. 2014. The kernel address sanitizer. https://lwn.net/Articles/612153/.

[17] Jack Edge. 2014. "Strong" stack protection for GCC. https://lwn.net/Articles/584225/.

[18] D. R. Engler, M. F. Kaashoek, and J. O'Toole. 1995. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, USA) *(SOSP '95)*. Association for Computing Machinery, New York, NY, USA, 251–266. https://doi.org/10.1145/224056.224076

[19] Matt Fleming. 2017. A thorough introduction to eBPF. https://lwn.net/Articles/740157/.

[20] David B Golub, Daniel P Julin, Richard F Rashid, Richard P Draves, Randall W Dean, Alessandro Forin, Joseph Barrera, Hideyuki Tokuda, Gerald Malan, and David Bohman. 1992. Microkernel operating system architecture and Mach. In *In Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*. 11–30.

[21] 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. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 1016–1031. https://doi.org/10.1145/2810103.2813611

[22] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC'19)*. USENIX Association, Renton, WA, 489–504. https://www.usenix.org/conference/atc19/presentation/hedayati-hodor

[23] 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. https://doi.org/10.1002/(SICI)1097-024X(19980725)28:9%3C901::AID-SPE181%3E3.0.CO;2-7

[24] 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. https://doi.org/10.1145/1151374.1151391

[25] Galen C. Hunt and James R. Larus. 2007. Singularity: Rethinking the Software Stack. *SIGOPS Oper. Syst. Rev.* 41, 2 (April 2007), 37–49.

[26] M Frans Kaashoek, Dawson R Engler, Gregory R Ganger, Héctor M Briceno, Russell Hunt, David Mazieres, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. 1997. Application performance and flexibility on exokernel systems. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*. 52–65.

[27] Douglas Kilpatrick. 2003. Privman: A Library for Partitioning Applications.. In *USENIX Annual Technical Conference, FREENIX Track*. 273–284.

[28] Gerwin Klein. 2009. Operating system verification—an overview. *Sadhana* 34, 1 (2009), 27–69.

[29] 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. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[30] 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. In *Proceedings of the 16th European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 376–394. https://doi.org/10.1145/3447786.3456248

[31] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning* (Dakar, Senegal) *(LPAR'10)*. Springer-Verlag, Berlin, Heidelberg, 348–370.

[32] 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. https://doi.org/10.1109/49.536480

[33] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. 2004. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)*. 17–30.

[34] Guanyu Li, Dong Du, and Yubin Xia. 2020. Iso-UniK: lightweight multi-process unikernel through memory protection keys. *Cybersecurity* 3, 1 (May 2020), 11.

[35] Shen Liu, Gang Tan, and Trent Jaeger. 2017. PtrSplit: Supporting General Pointers in Automatic Program Partitioning. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2359–2371. https://doi.org/10.1145/3133956.3134066

[36] 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. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. Association for Computing Machinery, 461–472.

[37] Toshiyuki Maeda and Akinori Yonezawa. 2003. Kernel Mode Linux: Toward an operating system protected by a type theory. In *Annual Asian Computing Science Conference*. Springer, 3–17.

[38] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security'20)*. USENIX Association, 699–716. https://www.usenix.org/conference/usenixsecurity20/presentation/narayan

[39] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. 2020. RedLeaf: Isolation and Communication in a Safe Operating System. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association. https://www.usenix.org/conference/osdi20/presentation/narayanan-vikram

[40] Ruslan Nikolaev and Godmar Back. 2013. VirtuOS: An Operating System with Kernel Virtualization. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 116–132. https://doi.org/10.1145/2517349.2522719

[41] Ruslan Nikolaev, Mincheol Sung, and Binoy Ravindran. 2020. LibrettOS: A Dynamically Adaptable Multiserver-Library OS. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Lausanne, Switzerland) *(VEE '20)*. Association for Computing Machinery, New York, NY, USA, 114–128. https://doi.org/10.1145/3381052.3381316

[42] Pierre Olivier, Antonio Barbalace, and Binoy Ravindran. 2020. The Case for Intra-Unikernel Isolation. *Proceedings of the 10th Workshop on Systems for Post-Moore Architectures* (April 2020).

[43] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. 2019. A Binary-Compatible Unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Providence, RI, USA) *(VEE 2019)*. Association for Computing Machinery, New York, NY, USA, 59–73. https://doi.org/10.1145/3313808.3313817

[44] J. M. Rushby. 1981. Design and Verification of Secure Systems. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles* (Pacific Grove, California, USA) *(SOSP '81)*. Association for Computing Machinery, New York, NY, USA, 12–21. https://doi.org/10.1145/800216.806586

[45] Vasily A. Sartakov, Lluis Vilanova, and Peter Pietzuch. 2021. CubicleOS: A Library OS with Software Componentisation for Practical Isolation (extended abstract). In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[46] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. 2020. Donky: Domain Keys – Efficient In-Process Isolation for RISC-V and x86. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security'20)*. USENIX Association, 1677–1694. https://www.usenix.org/conference/usenixsecurity20/presentation/schrammel

[47] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. 2020. Intra-Unikernel Isolation with Intel Memory Protection Keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Lausanne, Switzerland) *(VEE '20)*. Association for Computing Machinery, New York, NY, USA, 143–156. https://doi.org/10.1145/3381052.3381326

[48] Stylianos Tsampas, Akram El-Korashy, Marco Patrignani, Dominique Devriese, Deepak Garg, and Frank Piessens. 2017. Towards automatic compartmentalization of C programs on capability machines. In *Workshop on Foundations of Computer Security 2017*. 1–14.

[49] Unikraft Contributors. 2020. Unikraft release 0.4. https://github.com/unikraft/unikraft/tree/RELEASE-0.4.

[50] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *Proceedings of the 28th USENIX Security Symposium (USENIX Security'19)*. USENIX Association, Santa Clara, CA, 1221–1238. https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner

[51] Dmitry Vyukov. 2020. Syzkaller: Adventures in continuous coverage-guided kernel fuzzing. BlueHat IL.

[52] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles* (Asheville, North Carolina, USA) *(SOSP '93)*. Association for Computing Machinery, New York, NY, USA, 203–216. https://doi.org/10.1145/168619.168635

[53] Jiong Wang. 2018. Initial Control Flow Support for eBPF Verifier. https://lwn.net/Articles/753724/.

[54] Xiaoguang Wang, SengMing Yeoh, Robert Lyerly, Pierre Olivier, Sang-Hoon Kim, and Binoy Ravindran. 2020. A Framework for Software Diversification with ISA Heterogeneity. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. 427–442.

[55] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceedings of the 41st International Symposium on Computer Architecture*. 457–468.

[56] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfen Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. 2018. KylinX: A Dynamic Library Operating System for Simplified and Efficient Cloud Virtualization. In *2018 USENIX Annual Technical Conference (USENIX ATC'18)*. USENIX Association, 173–186.