FlexOS: Towards Flexible OS Isolation

FlexOS: Towards Flexible OS Isolation

Hugo Lefeuvre
The University of Manchester
Manchester, UK

Vlad-Andrei Bădoiu
University Politehnica of Bucharest
Bucharest, Romania

Alexander Jung
Lancaster University / *Unikraft.io*
Lancaster, UK

Stefan Lucian Teodorescu
University Politehnica of Bucharest
Bucharest, Romania

Sebastian Rauch
Karlsruhe Institute of Technology
Karlsruhe, Germany

Felipe Huici
NEC Labs Europe / *Unikraft.io*
Heidelberg, Germany

Costin Raiciu*
UPB / *Correct Networks*
Bucharest, Romania

Pierre Olivier
The University of Manchester
Manchester, UK

ABSTRACT

At design time, modern operating systems are locked in a specific safety and isolation strategy that mixes one or more hardware/software protection mechanisms (e.g. user/kernel separation); revisiting these choices after deployment requires a major refactoring effort. This rigid approach shows its limits given the wide variety of modern applications' safety/performance requirements, when new hardware isolation mechanisms are rolled out, or when existing ones break.

We present FlexOS, a novel OS allowing users to easily specialize the safety and isolation strategy of an OS at compilation/deployment time instead of design time. This modular LibOS is composed of fine-grained components that can be isolated via a range of hardware protection mechanisms with various data sharing strategies and additional software hardening. The OS ships with an exploration technique helping the user navigate the vast safety/performance design space it unlocks. We implement a prototype of the system and demonstrate, for several applications (Redis/Nginx/SQLite), FlexOS' vast configuration space as well as the efficiency of the exploration technique: we evaluate 80 FlexOS configurations for Redis and show how that space can be probabilistically subset to the 5 safest ones under a given performance budget. We also show that, under equivalent configurations, FlexOS performs similarly or better than existing solutions which use fixed safety configurations.

CCS CONCEPTS

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

KEYWORDS

Operating Systems, Security, Isolation

*UPB: University Politehnica of Bucharest

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02…$15.00
https://doi.org/10.1145/3503222.3507759

ACM Reference Format:
Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. 2022. FlexOS: Towards Flexible OS Isolation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3503222.3507759

1 INTRODUCTION

Modern OS architectures are heavily interlinked with the protection mechanisms they rely upon. OSes rigidly commit at design time to various high-level safety decisions, such as the use of software verification, hardware isolation, runtime checking, etc. Changing these after deployment is rare and costly.

The current OS design landscape (depicted in Figure 1) broadly consists of micro-kernels [34, 45], which favor hardware protection and verification over performance, monolithic kernels [8], which choose privilege separation and multiple address spaces (ASes) to isolate applications, but assume all kernel code is trusted, and single-address-space OSes (SASOSes), which attempt to bring isolation within the address space [12, 33, 52], or ditch all protection for maximum performance [47, 58, 70]. Making post-design changes to these high-level safety decisions is very difficult to implement. For instance, removing the user/kernel separation [59] requires a lot of engineering effort, as does breaking down a process into multiple address spaces for isolation [42]. Recently, the potential safety benefits hinted by the proposal to introduce Rust components in Linux [23] are questioned by the fact that the bulk of the kernel code will remain written in a memory-unsafe language [22].

The rigid use of safety primitives in modern OSes poses a number of problems. First, it precludes per-application OS specialization [24, 39, 60, 62] at a time when modern applications exhibit a wide range of safety and performance requirements. Prematurely locking the design into any combination of safety primitives is likely to result in suboptimal performance/safety in many scenarios. Effortless specialization for safety is further motivated by the fact that today's applications are made up of multiple components showing different degrees of trust and criticality, and as such requiring various levels of isolation. Furthermore, new isolation mechanisms [3, 4, 15, 18, 75, 82], with the ability to complement or replace traditional ones, are regularly being proposed by CPU manufacturers. When multiple mechanisms can be used for the same task, choosing the most

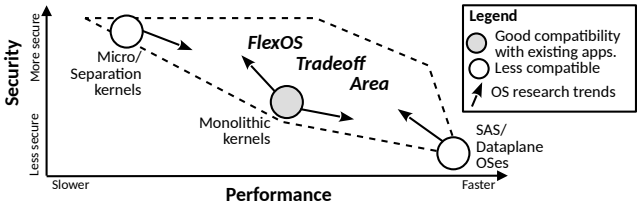

Figure 1: Design space of OS kernels.

suitable primitive depends on many factors, and should ideally be postponed to deployment time. Finally, when the protection offered by a hardware primitive breaks down (e.g. Meltdown [56]), it is difficult to decide how it should be replaced, and generally costly to do so.

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 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. To that aim, we extend the Library OS (LibOS) model and augment its capacity to be specialized towards a given use case, historically done for performance reasons [24, 39, 60, 62], towards the *safety* dimension.

With FlexOS, the user can decide, at *build time*, which of the fine-grained OS components should be placed in which compartment (e.g. the scheduler, TCP/IP stack, etc.), how to instantiate isolation and protection primitives for each compartment, what data sharing strategies to use for communication between compartments, as well as what software hardening mechanisms should be applied on which compartments. To that aim, we abstract the common operations required when compartmentalizing arbitrary software behind a generic API that is used to retrofit an existing LibOS into FlexOS. This API limits the manual porting effort of kernel and application legacy components to the marking of shared data using annotations. These annotations, alongside other abstract source-level constructs, are replaced at build time by a code transformation step that instantiate a given FlexOS safety configuration.

The design space enabled by the system, illustrated on Figure 1, is very large and difficult for a non-expert user to explore manually. This leads to the second research question we explore: *how to guide the user navigating the vast design space unlocked by FlexOS?* To answer this, we propose a semi-automated exploration technique named *partial safety ordering*, using partially ordered sets to describe the probabilistic security degrees of FlexOS' configurations and identify the safest ones under a given performance budget.

We have implemented a prototype of FlexOS with support for Intel MPK and VM/EPT-based isolation, as well as a wide range of hardening mechanisms (CFI [1], ASAN [79], etc.). Our evaluation using four popular applications demonstrates the wide safety versus performance tradeoff space unlocked by FlexOS: we evaluate over 160 configurations for Redis and Nginx. We also show the ease of exploring different points in that space: our semi-automated exploration technique can probabilistically subset the 80 Redis configurations to the 5 safest ones under a given performance budget. Finally, we demonstrate that under equivalent configurations, FlexOS performs better or similarly to baselines/competitors: a monolithic kernel, a SASOS, a microkernel, and a compartmentalized LibOS.

2 FLEXIBLE OS ISOLATION: PRINCIPLES, CHALLENGES

FlexOS seeks to enable users to easily and safely switch between different isolation and protection primitives at deployment time. This section formalizes the fundamental design principles required to achieve this, the challenges that arise from them, and how we address them.

2.1 Principles

(P1) ***The isolation granularity of FlexOS' components should be configurable.*** The compartmentalization strategy, i.e. the number of compartments and which components are merged/split into compartments, has a major impact on safety and performance, thus it should be configurable.
(P2) ***The hardware isolation mechanisms used should be configurable.*** There is a wide range of isolation mechanisms with various safety and performance implications. These should be configurable by the user. For the OS developer, supporting a new mechanism should not involve any rewrite/redesign and be as simple as implementing a well-defined API.
(P3) ***Software hardening and isolation mechanisms should be configurable.*** Software hardening techniques such as CFI, or Software Fault Isolation (SFI), as well as memory safe languages such as Rust, bring different levels safety at a variable performance cost. They should be selectively applicable on the components they are the most meaningful for in a given use case.
(P4) ***Flexibility should not come at the cost of performance.*** The OS runtime performance should be similar to what would be achieved with any particular safety configuration without the flexibility approach.
(P5) ***Compatibility with existing software should not come at a high porting cost***, to maximize adoption.
(P6) ***The user should be guided in the vast design space enabled by FlexOS.*** Given its very large configuration space, the system should come with tools helping the user identify suitable safety/performance configurations for a given use case.

2.2 Challenges and Approach

P1 and P4 raise the question of ***how to offer variable isolation granularities, and how to do so without compromising performance?*** Genericity is typically paid at the price of performance [47, 61, 62], and interface design may not be easily decoupled from the isolation granularity without performance loss [27]. In order to tackle this issue, we propose to rely on a LibOS design that is *already finely modularized while providing state of the art performance*, Unikraft [47]. The main idea is to consider Unikraft's level of modularization (micro-library) as a minimal granularity, using pre-existing interfaces as compartment boundaries. Then, in order to maximize performance and safety for a given use case,

less granular configurations can be composed by merging select components into compartments. At build time when an isolation mechanism is selected, FlexOS uses code transformations to inline function-call-like cross-domain gates, avoiding the overhead of a runtime abstraction interface [26].

P2 and P5 bring the challenge of ***how to design an OS in which 1) isolation can be enforced by many hardware mechanisms and 2) the engineering cost of introducing a new mechanism is low?*** Technology agnosticism is already difficult in userland software, but core kernel facilities (interrupt handling, memory management, scheduling) introduce additional complexity that should be handled very differently depending on the underlying isolation technology. For example, some technologies share a single address space between protection domains (e.g. MPK [15]) while other use disjoint address spaces (e.g. TEEs [3], EPT). The main idea of FlexOS is to abstract existing isolation technologies and identify kernel facilities that require different handling depending on the technology, and design these subsystems so as to minimize the changes needed when implementing a new technology.

P5 asks ***how to limit the engineering costs of porting new applications/libraries?*** To allow compatibility with existing software, FlexOS extends an OS that offers a POSIX interface. That OS is compartmentalized by marking cross-component calls and shared data using an abstract API and, in its basic form, porting a new application requires the developer to use the same API to mark shared data (i.e. data passed to other components) with source-level annotations. This avoids the need to change the application design or major code rewriting. Such an approach is common among state-of-the art compartmentalization frameworks [32, 65, 75, 80].

Finally, P1-P3 and P6 raise the question of ***how to help the user navigate the vast design space enabled by FlexOS?*** The introduction of safety flexibility increases the potential for safety/performance specialization, but selecting suitable configurations may be hard for a non-expert. For example, it can be difficult to reason about the safety implications of increasing the degree of compartmentalization vs. increasing the level of software hardening for a given configuration. To tackle that issue, we propose a method named *partial safety ordering*, using partial order relationships to probabilistically rank FlexOS configurations by safety and identify the safest ones for a given application under a performance budget.

Section 3 presents an OS design that satisfies P1-P5, and Section 4 gives key implementation points of a prototype we developed. Section 5 shows an approach to tackle P6. Finally, Section 6 presents an evaluation of our prototype.

3 DESIGNING AN OS WITH FLEXIBLE ISOLATION

We now provide an overview of FlexOS' main elements, starting with an overview of its design, compartmentalization API, the backend API, and finally the trusted computing base.

FlexOS is based on a modular LibOS, Unikraft [47] composed of a set of independent, fine grained libraries. In FlexOS, each library can be placed in a given compartment (an isolation domain), and it can be hardened via techniques such as Control-Flow Integrity (CFI), address sanitization and so forth. This safety configuration is

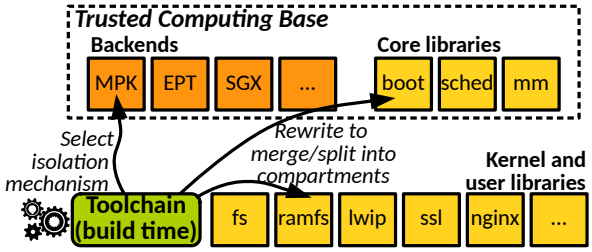

Figure 2: OS overview. The TCB includes backends and core libraries. Backends are used by the toolchain to rewrite the libraries at build time.

provided at build time, in a configuration file provided by the developer, and FlexOS' toolchain produces an OS image with the desired safety characteristics. Below is an example of such a configuration file that isolates libopenjpg and lwip in a separate compartment with CFI and ASan enabled.

```
compartments:
  - comp1:
    mechanism: intel-mpk
    default: True
  - comp2:
    mechanism: intel-mpk
    hardening: [cfi, asan]
libraries:
  - libredis: comp1
  - libopenjpg: comp2
  - lwip: comp2
```

In contrast to Unikraft where all libraries are in the same protection domain and any library can directly call a function from another library, in FlexOS' source code libraries call external functions via *abstract gates*, and may share data with external libraries at the granularity of a byte using abstract code annotations. Gates and annotations form an API used to compartmentalize Unikraft into FlexOS, and represent metadata which is automatically replaced by our toolchain with a particular implementation at build time. Different implementations can leverage different isolation technologies, or flavors of a same technology. We refer to the API implementation for a given technology (MPK, EPT, etc.) together with its runtime library as *isolation backend*. This subsection gives a short overview of FlexOS' main design elements, which are then elaborated in the following subsections. Figure 2 depicts the components described in this subsection.

LibOS Basis. Achieving flexible isolation at a fine granularity implies a high degree of modularity. In practice, this modularity is not offered by typical monolithic general-purpose OSes [47]. A flexible isolation approach on the basis of Linux would require a first non-trivial "modularization" step [55] that may take years of engineering and careful redesign. Library OSes [47] and component-based OSes [10, 72] are a better starting point for flexible OS isolation because they often provide highly modular code bases with good application compatibility and high performance. Flexible isolation also suits well the specialization spirit of LibOSes, where the OS can be tailored for a given application/use-case. This was historically done for performance [24], and FlexOS enables specialization towards safety.

API and Build-time Instantiation. Unlike a typical LibOS, we design FlexOS in an *isolation-agnostic* manner. Cross compartment calls are made through abstract call gates that are instantiated at build time (arrows in Figure 2). Shared data is marked using compiler annotations, used at build time to instantiate a given data sharing strategy. Unlike linker-based approaches [74], FlexOS performs replacements using source to source transformations using Coccinelle [48, 71]. This has the advantage of allowing all compiler optimizations and gives FlexOS a clear performance advantage compared to historical approaches that relied on heavyweight runtime abstraction interfaces such as `COM` for Flux OSKit [26]. It also makes FlexOS' isolation approach easy to debug and understand by anyone who knows C: transformations can be visually inspected in a high-level language with usual file comparison tools.

3.1 Compartmentalization API and Transformations

Most isolation mechanisms (memory protection keys [15], TEEs like SGX [18], or hardware capabilities [82]) restrict data access according to a set of current privileges, and provide a means to switch privileges and share data across compartments. Ensuring safety is equivalent to controlling privilege transitions, making sure that the system only ever enters "legal" couplings of executing code and data privileges. Other isolation approaches such as ARM TrustZone [3] or EPT/VMs consider compartments as entirely different systems (or "worlds"), enforcing a 1:1 system/compartment mapping. With this approach, systems never switch privileges, instead they communicate with other compartments via remote procedure calls (RPCs) and shared memory. We design FlexOS' call gates and data sharing primitives to cater for both approaches. In FlexOS, the only requirement for an isolation mechanism is to (1) implement the concept of protection domains and provide a domain switching mechanism, and (2) support some form of shared memory for cross-domains communication. To the best of our knowledge, this applies to the vast majority of industry and research isolation mechanisms. This subsection gives an overview of FlexOS' compartmentalization approach, first focusing on the API with call gates and shared data, and then on build-time source transformations.

Call Gates. In FlexOS, cross-library calls are represented in the source code by *abstract call gates*. At build time, as part of the transformation phase, abstract call gates are replaced with a specific implementation. For instance, when the caller and callee are configured to be in the same compartment, call gates implement a classical function call. When they are in different compartments, isolated for example by MPK, the call gate performs a protection domain switch before finally executing the `call` instruction. In a setting where libraries are isolated using VMs, the call gate performs a remote procedure call (RPC). From the perspective of the compiler, caller, and the callee, call gates are entirely transparent as they implement the System V ABI calling convention. Unlike typical System V function calls however, call gates guarantee isolation of the register set and therefore save and zero out all registers not used by parameters. Figure 3 presents an example of gates from the porting (step ②) to the replacement by the toolchain (③ and ③').

The part of the process of porting existing user/kernel code to FlexOS consisting in marking call gates is automated: knowing the control-flow graph of the system, static analysis determines whether a procedure call crosses library boundaries, and if so, performs a syntactic replacement of the function call with a call gate instead. A corner case requiring programming effort is when a component calls another component through a function pointer. The callee cannot be determined statically, thus the programmer must annotate the possible pointed functions with the list of possible components they can be called from. The toolchain will then generate wrappers enclosing the implementations of the functions in question in the appropriate call gates. Our prototype implementation uses Cscope [14] and Coccinelle [71].

FlexOS call gates are not trampolines. Instead, they replace System V function calls entirely and are always inlined at the call site. An advantage of such approach is that call gates naturally provide an inexpensive (albeit incomplete) form of CFI, guaranteeing that libraries can only be entered through well defined entry points, known and enforced at compile time.

Data Ownership Approach. FlexOS takes a code-centered [30] isolation approach. Each library is present only once and maps to a specific set of privileges. There is a slight tweak for backends that rely on several systems (TrustZone, VMs): for them, the trusted computing base (§3.3) is duplicated; one for each system, as each compartment must possess a self-contained kernel (§4.2).

FlexOS considers all static and dynamic data allocated by a library as private by default. Individual variables can then be annotated as "shared" with a specific group of libraries into *whitelists*, similarly to access control lists. In practice, the maximum number of isolated data sharing "zones" is limited by the underlying technology. Annotations are made with the keyword `__shared` as illustrated in Figure 3 step ②.

Compiler annotations are identical for all types of variables. However, under the hood, FlexOS differentiates between statically allocated variables, dynamically allocated heap variables, and dynamically allocated stack variables.

FlexOS' compartmentalization API itself does not dictate *how* variables have to be shared. Different mechanisms can require very different sharing approaches: while certain mechanisms such as MPK require shared data to be located in shared memory regions, others such as CHERI's hybrid capabilities [83] require compiler annotations that can be automatically generated in place of the FlexOS placeholder. Section 4 describes the implementation of the API for the two supported backends (MPK/EPT), and sketches implementations for an additional one (CHERI).

Identifying shared data represents the vast majority of the porting effort. It is necessary for both kernel libraries, user libraries, and applications. On the kernel side, this problem is simplified (but not eliminated) by the modularity of Unikraft's code base. This issue is not specific to FlexOS and is widely explored in the literature. State of the art approaches (1) rely on manual code annotations [65], (2) perform static analysis at compile time to identify shared data automatically [6], or (3) perform a mix of static, dynamic, and manual analysis [30]. There is no silver bullet: manual code annotation can be non-trivial, but typically produces precise results that not only take into account what *is* accessed across modules, but also what *should be* shared from a security perspective. Static-analysis

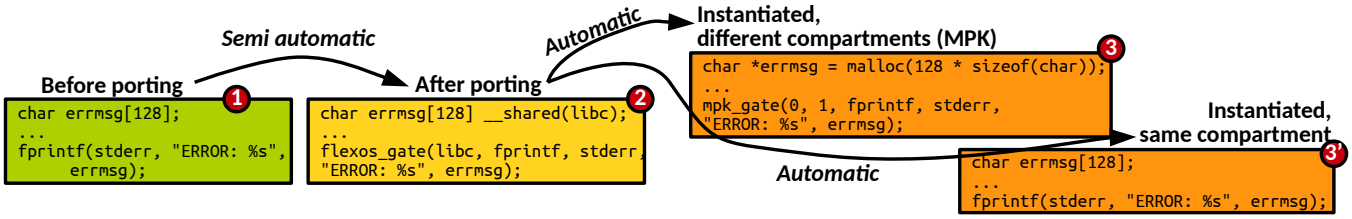

Figure 3: FlexOS code transformations. First, developers manually annotate shared data, and gate placeholders are automatically inserted. At build time, API primitives are automatically replaced with the chosen mechanism. In the MPK case, shared data can for example be allocated on a shared heap. If the two libraries are in the same compartment, the result is similar to the code prior porting, resulting in zero overhead.

based approaches, on the other hand, are automatic, but conservative. These methods would be applicable to FlexOS, however automated shared data identification is not the main focus of this paper. The current prototype relies on manual annotations, and Section 4 details the porting effort for a number of applications and libraries.

Build-time Source Transformations. Before compilation, FlexOS' toolchain performs source transformations to (1) instantiate abstract gates, (2) instantiate data sharing code, (3) generate linker scripts, and (4) generate additional code in core libraries according to backend-provided recipes. The amount code generated in considerable. As an example, the toolchain modifies about 1 KLoC for a simple Redis configuration. Figure 3 steps ③ and ③' presents an example of the porting-transformation process.

3.2 Kernel Backend API

Most isolation mechanisms require changes to a specific set of components in the kernel. The kernel facilities that can require special handling depending on the technology exclusively correspond to the core libraries (see Figure 2). In order to make such changes scalable, we designed core components to expose a *hook API* to isolation backends, allowing the core libraries to be easily extended with backend specific functionalities. For example, the MPK backend leverages the thread creation hook offered by the scheduler to switch a newly created thread to the right protection domain. These hooks come at no cost: since the instantiation is done at build time, the compiler is able to aggressively inline such calls.

Porting FlexOS to use a new isolation mechanism does not require redesign. In general, it is equivalent to (1) implementing gates for the particular mechanism, (2) implementing hooks for core components (see previous paragraph), (3) implementing linker script generation in the toolchain, (4) implementing Coccinelle code transformations, and (5) registering the newly created backend into the toolchain. In practice, developers can heavily reuse existing transformations for new backends.

3.3 Trusted Computing Base

Regardless of the isolation mechanism, certain components are so deeply involved in the OS' functioning that they will cause the entire system to violate its safety guarantees when compromised. These components are (1) the early boot code, (2) the memory manager, (3) the scheduler, (4) the first-level interrupt handler's context switch primitives, and (5) the isolation backend. We refer to these components as FlexOS' trusted computing base (TCB), illustrated in Figure 2. Clearly, malfunctioning or malicious early boot code can setup the system in an unsafe manner, the memory manager can manipulate page table mappings in order to freely access any compartment's memory, the scheduler can manipulate sleeping thread's register states, and the backend provide incomplete isolation, etc. This is the case even when considering architectural hardware capabilities such as CHERI [21]. It comes as no surprise: this "core" set of libraries is historically the set of services that microkernel OSes provide [78]. FlexOS' TCB is small: around 3000 LoC in the case of Intel MPK, and even less for VM/EPT.

Trust Model. The whole point of flexible isolation is to be able to achieve a wide range of trust models where different components (such as the network stack, parser libraries, etc.) can be considered untrusted and potentially compromised. Thus there is no single trust model for FlexOS. In general, however, we assume that the TCB (see previous paragraph) is safe and error free. This is not an unreasonable assumption given the small size and the potential for formal verification (we have formally verified a version of our scheduler [50] using Dafny [51]). The hardware and the compiler are also part of the TCB. Note that the rest of the toolchain (Coccinelle included) is *not* part of the TCB as the code includes compile time checks that are able to detect invalid transformations. Finally we must also assume that interfaces correctly check arguments and are free of confused deputy/Iago [13] situations. This is not an unreasonable assumption within the core FlexOS codebase. Further, confused deputy and Iago attacks are probabilistically made more complex to execute in FlexOS due to the variability of the interface size; the system call API, for example, is divided into a variable number of sub-interfaces depending on the chosen configuration, and several compartments may need to be subverted for an attack to be successful.

4 PROTOTYPE

We present a prototype of FlexOS on top of Unikraft [47] v0.5, with Intel MPK and EPT backends. Modification to the Unikraft kernel represent about 3250 LoC: 1400 for the MPK backend, 1000 for EPT, and 850 for core libraries. In user space, changes to Unikraft's toolchain represents 2300 LoC. We port user codebases (Redis, Nginx, iperf, and SQLite) as well as most kernel components (the

TCP/IP stack, scheduler, filesystem, etc.) to run as isolated components. This section presents the MPK and EPT backends, sketches a CHERI backend, and concludes with the porting effort.

4.1 Intel MPK Isolation Backend

MPK is a mechanism present in Intel CPUs offering low-overhead intra-AS memory isolation [5, 37, 75]. MPK leverages unused bits in the page table entries to store a *memory protection key*, enabling up to 16 protection domains. The PKRU register then stores the protection key permissions for the current thread. On each memory access, the MMU compares the key of the target page with the PKRU and triggers a page-fault in case of insufficient permissions. FlexOS associates each compartment with a protection key and reserves one key for a shared domain used for communications. If the image features less than 15 compartments, FlexOS uses remaining keys for additional shared domains between restricted groups of compartments. Any compartment can modify the value of the PKRU, thus the MPK backend has to prevent unauthorized writes. This has previously been done via runtime checks [32] and static analysis [80]. In FlexOS, no code is loaded after compilation, hence static binary analysis coupled with strict W⊕X is sufficient.

MPK Gates. For flexibility, FlexOS offers two different implementations of the MPK gate. The main one provides full spatial safety, similarly to HODOR [32]. The gate protects the register set and uses one call stack per thread per compartment. Each compartment has a stack registry that maps threads to their local compartment stack, making it fast and safe to switch the call stack. Upon domain transition, the gate (1) saves the current domain's registers set, (2) clears registers, and (3) loads function arguments. It then (4) saves the current stack pointer, (5) switches thread permissions, (6) switches the stack, and finally (7) executes the `call` instruction. Once the function has returned, operations are executed in reverse.

The second gate implementation shares the stack and the register set across compartments, similarly to ERIM [80]. It is conceptually very simple, switching the content of the PKRU before performing a normal function call. This lightweight implementation offers lesser guarantees but presents a lighter overhead, close to the raw cost of `wrpkru` instructions.

Data Ownership. FlexOS' MPK images feature one data, read-only data, and bss section per compartment to store private compartment static data. At boot time, the boot code protects these sections with the compartment's protection key.

Each compartment has a private heap, and a shared one is used for communications. Our prototype uses a single shared heap for all shared allocations, but this is not a fundamental restriction. Stack allocations are slightly more complex. Existing works convert shared stack allocations to shared heap allocations [6, 32, 44]. This approach is costly from a performance perspective: an allocation+free on the fast path for a modern allocator typically takes 30-60 cycles, and up to thousands of cycles on the slow path [40]. This is as expensive as entire domain transitions, and that for a single shared stack variable. While FlexOS supports stack-to-heap conversions, we propose another approach that addresses this issue, the *data shadow stack*.

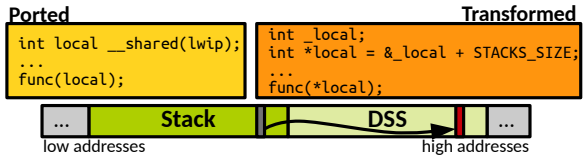

Figure 4: Data Shadow Stacks.

Data Shadow Stacks. Stack allocations are much faster than heap allocations because the compiler is able to perform bookkeeping *at compile time*. At runtime, a single push instruction is needed, resulting in constant low cost. Data Shadow Stacks (DSS), illustrated in Figure 4, leverage this bookkeeping work for shared stack allocations.

When using the DSS, the usual stack size of threads is doubled. The upper part corresponds to the DSS and is put in the shared domain. The lower part is the traditional stack and remains in the compartment's private domain. For each shared variable `x`, we define the *shadow* of `x` as `&x + STACK_SIZE`. Thus, allocating space for a shared variable on the stack transparently allocates a shadow variable in the DSS. Before compilation, the toolchain replaces every reference to a shared stack variable with its shadow `*(&var + STACK_SIZE)` in the shared domain.

Allocations on the DSS are much faster than on a shared heap, since the DSS' bookkeeping overhead is null (stack speed), and the locality of reference high. The cost is a relatively small increase in memory usage (stacks are twice as large). The DSS mechanism is applicable to any isolation mechanism that supports shared memory, and is compatible with common stack protection mechanisms.

Control Flow Integrity. Intel MPK does not provide protection from execution. As such, if a compartment is compromised and the attacker ROPs into another compartment, a fault will not directly happen. The MPK backend is able to provide a certain form of CFI, ensuring that compartments can only be entered at well-defined points. This ability is the consequence of the hardcoding of gates as described in 3.1. If the control-flow of one compartment is compromised and the attacker ROPs directly into another compartment c, then the system is guaranteed to crash if any data local to c is accessed.

4.2 EPT/VM Backend

Virtualization has been used in many works to support isolation within a kernel [53, 67, 68, 84]. Hardware-assisted virtualization is widely supported and provides strong safety guarantees compared to MPK, at the cost of higher overheads. The EPT backend is an extreme case; compartments do not share ASes and run on different vCPUs. It shows that FlexOS is able to cater very different mechanisms under a common API.

FlexOS' EPT backend generates one VM image per compartment, each containing the TCB (boot code, scheduler, memory manager, backend runtime) and the compartment's libraries. Communications use a shared memory-based RPC implementation. Our prototype runs on QEMU/KVM patched to support lightweight inter-VM shared memory (less than 90 LoC).

EPT Gates. Upon domain transition, the caller places a function pointer and arguments in a predefined shared area of memory. All other VMs busy wait until they notice an RPC request, check that the function is a legal API entry point, execute the function and place the return value in a predefined area of the shared memory. In order to support multithreaded loads, each RPC server maintains a pool of threads that are used to service RPCs. Using function pointers instead of abstract routine identifiers simplifies the RPC server's unmarshalling operation and does not prevent the RPC server from checking the pointer to ensure that it is a legal entry point. This optimization is possible since all compartments are built at the same time, hence all addresses are known.

Busy-waiting allows the EPT backend to minimize gate latency as opposed to VM notifications, but a similar implementation with `MONITOR/MWAIT` instructions would also be possible to minimize power consumption if calls are sparse. Overall, any of these tweaks can be implemented as gate variant in order to offer as much freedom as possible to the user.

Data Ownership. The EPT backend relies on shared memory areas to share data (static and dynamic) across VMs. Areas are always mapped at the same address in the different compartments so that pointers to/in shared structures remain valid. Each VM manages its own portion of the shared memory area to avoid the need for complex multithreaded bookkeeping.

Control Flow Integrity. The EPT backend is able to provide a form of CFI stronger than that of the MPK backend, ensuring that compartments can only be *left and entered* at well defined points. Indeed, the RPC server can control at entry that the executed function is legal, and compartments are not able to execute other compartments' code without RPC calls.

4.3 Supporting More Isolation Mechanisms

To check whether FlexOS can support other isolation backends, we discuss how we can leverage the CHERI hardware capabilities [82], an emerging isolation hardware mechanism. The CHERI ISA extension is available for ARMv8-A, which is supported by FlexOS. Among others, CHERI capabilities would extend FlexOS' trade-off space with the ability to address confused-deputy situations, reduce data sharing, and allow for a larger number of domains, something that is currently impossible for architectural (MPK) and performance (EPT) reasons. The backend would use boot-time hooks to initialize CHERI support, and scheduler hooks to perform capability-aware context-switching and thread initialization. Similarly to other backends, CHERI gates would save caller context, clear the relevant traditional and capability registers, install the callee context, and rely on the domain crossing instruction `CInvoke` and sentry capabilities [81] to perform protection domain jumps. As a first step, FlexOS should rely on the hybrid pointer model to maximize compatibility. Our API's shared data annotations would transform to `__capability` at build time to treat shared variables as a capabilities for efficient communications.

4.4 Porting Effort

The porting process consists of two phases: call gate insertion (automated), and shared data annotation (manual). The typical workflow, once gates have been inserted, is to run the program with a representative test case (e.g., a benchmark or test suite) until it crashes due to memory access violations. Crash reports point to the symbol that triggered the crash, at which point the developer can annotate it for sharing. In some cases, the crash can be a genuine violation; e.g., a library exposes internal state to external libraries, in which case the developer can decide to rework the library's API to address the privacy issue. This case is much less frequent and left at the developer's discretion. An example is `ramfs`, which is so deeply entangled with `vfscore` that blindly isolating it without redesign would impair performance with little additional security benefits, as a critical portion of the component's state would be shared. However, coupled with `vfscore`, both components can perfectly well be isolated from the rest of the system. This highlights a limitation of automated tools that blindly isolate this component [74]. Overall, the porting process is greatly simplified by common debugging tools: GDB and all usual debugging toolchains are supported. The debugging experience in FlexOS is not significantly different from Unikraft and most mainstream OSes, and we expect it to remain intuitive for anyone familiar with OS development. Depending on the amount of data shared with the outside world, the porting process ranges from 10 minutes (time subsystem, no data shared), to 2-5 days (filesystem, network stack). This porting cost is similar that of other compartmentalization frameworks [65]. Table 1 illustrates the porting effort with concrete numbers.

Table 1: Porting effort: size of the patch (including automatic gate replacements), number of shared variables.

Libs/Apps	Patch size	Shared vars
TCP/IP stack (LwIP)	+542 / -275	23
scheduler (`uksched`)	+48 / -8	5
filesystem (`ramfs`, `vfscore`)	+148 / -37	12
time subsystem (`uktime`)	+10 / -9	0
Redis	+279 / -90	16
Nginx	+470 / -85	36
SQLite	+199 / -145	24
iPerf	+15 / -14	4

4.5 Software Hardening

The flexible isolation provided by FlexOS allows to enable/disable software hardening (SH) such as CFI, etc., on a per-component basis: isolating components without SH from components with it allows the latter to maintain the guarantees offered by SH. Moreover, many SH schemes work by instrumenting the memory allocator, and we use FlexOS' capacity to have an allocator per-compartment to enable flexible SH. This flexibility allows for example to alleviate the performance impact of SH by enabling it only for a subset of the system. Our prototype currently uses address sanitization (KASan), undefined behavior sanitization (UBSan), CFI, and stack protector.

5 EXPLORATION WITH PARTIAL SAFETY ORDERING

In this section we present a design space exploration technique, *partial safety ordering*, that aims to guide a user towards suitable configurations for a given use case by subsetting the vast design

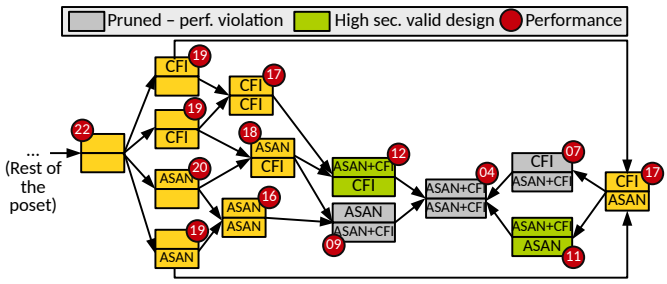

Figure 5: Partial view of the configuration poset for a fixed compartmentalization (2 compartments), varying per-compartment software hardening (CFI/ASAN).

space enabled by FlexOS according to safety and performance requirements.

Given a performance budget, partial safety ordering attempts to find the most secure configurations among those enabled by FlexOS. Quantifying safety is challenging: it is impossible to give each configuration an absolute safety score that would allow to completely order them; for instance, is the safety of a configuration with 3 compartments, MPK isolation and no hardening better or worse than another one with 2 compartments, EPT isolation and CFI hardening?

Nevertheless, the safety of *some* configurations is programmatically comparable. Consider 3 configurations, *C1* with no isolation and no software hardening; *C2* with two compartments protected by a given mechanism with a given data sharing strategy and no hardening; and *C3* adding CFI for each compartment on top of *C2*. In terms of (probabilistic) safety, we have the following relationship: $C1 \leq C2 \leq C3$. With that in mind, it is thus possible to organize all configurations into a *partially ordered set* (poset), that can be viewed as a Directed Acyclic Graph (DAG) for which each node represents a configuration, and a directed edge between nodes *n1* and *n2* indicates that the level of safety of *n1* is probabilistically superior to that of *n2*. The safety of nodes on the same path is comparable, while that of nodes on different paths is not.

Figure 5 presents a subset of the configuration poset corresponding to fixed choices for a compartmentalization strategy with 2 compartments, an isolation mechanism, and a strategy of data sharing. This subset of the poset represents the variation of the last feature, the software hardening, for which we assume only CFI and ASAN for the sake of simplicity. Each configuration is depicted by a node indicating, for each of the two compartments, which hardening mechanism is applied: none, CFI, ASAN, and CFI+ASAN. We construct the poset partially depicted on Figure 5, ordering safety with the assumption that safety probabilistically increases with 1) the number of compartments; 2) data isolation (isolated vs. shared stacks, dedicated shared memory areas per pair of communicating compartments vs. shared areas accessible from everywhere, etc.); 3) stackable software hardening; and 4) the strength of the isolation mechanism.

Given such a poset, we can label each node with its performance characteristics (circles in the figure denote fictional performance numbers), and prune those that don't meet minimum requirements (gray nodes), ultimately yielding a set of configurations that offer the best guarantees for a given performance budget. This set corresponds to the *maximal elements* of the poset, i.e. sinks of the DAG (green nodes in the figure).

Partial Safety Ordering in Practice. In practice, users provide the toolchain with a test script (e.g., `wrk` for Nginx) and a performance budget (e.g., at least 500k req./s). Users are free to define performance as they may deem suitable depending on their needs: application throughput, tail latency, runtime, etc. Any metric is suitable as long as it remains comparable across configurations and runs. With this in hand, the toolchain generates the unlabeled poset. Then, it labels it by automatically measuring the performance of each configuration. The toolchain does not have to run all configurations: assuming monotonically decreasing performance, it can safely stop evaluating a path as soon as a threshold is reached. In practice, we observe that this significantly limits combinatorial explosion. The result is a set of the most secure configurations for the given budget, which the user can use to choose the most suitable one for a given use case. Ultimately, we expect this process to significantly trim the design space and allow the user to make an informed and relatively effortless choice.

This approach assumes that the user is able to get representative feedback on the application's performance, and users will not be able to use FlexOS' exploration facilities if they are not able to properly benchmark their application. However, we expect this situation to be quite rare: in the vast majority of cases, users will be able to at least minimally test their applications. These results can be used to exclude configurations that are too costly and test the best candidates in production using lightweight performance measurement systems, e.g., blue-green deployments.

Skipping Exploration. Some developers might already come with a particular isolation strategy in mind. In that case the developer can skip this exploration phase by providing a configuration file as shown in Section 3. In this case, the developer leverages FlexOS' flexibility and not its exploration facilities. We note, however, that this "expert" approach has its limits: applications evolve over time and a compartmentalization approach that is deemed optimal at a given time may not be suitable in the future [30]. In this case, an exploration system such as FlexOS' can be of use for the expert to easily reconsider their approach in light of changing software.

6 EVALUATION

We aim to demonstrate the vast performance/safety design space enabled by FlexOS, assess the efficiency of the partial safety ordering exploration technique, and compare FlexOS' performance with the literature. To this end, we present an overview of the performance obtained with numerous safety configurations on three popular cloud applications (Redis, Nginx, and SQLite), as well as iPerf, a standard network stack benchmark. We demonstrate our design-space exploration technique with Redis and Nginx. Then, we compare selected SQLite configurations with Linux, CubicleOS [74], a (non-flexible) compartmentalized LibOS, the SeL4 [45]/Genode [25] microkernel, as well as Unikraft [47]. Finally, we study raw isolation overheads in FlexOS: DSS efficiency and cross-compartments call gate latencies.

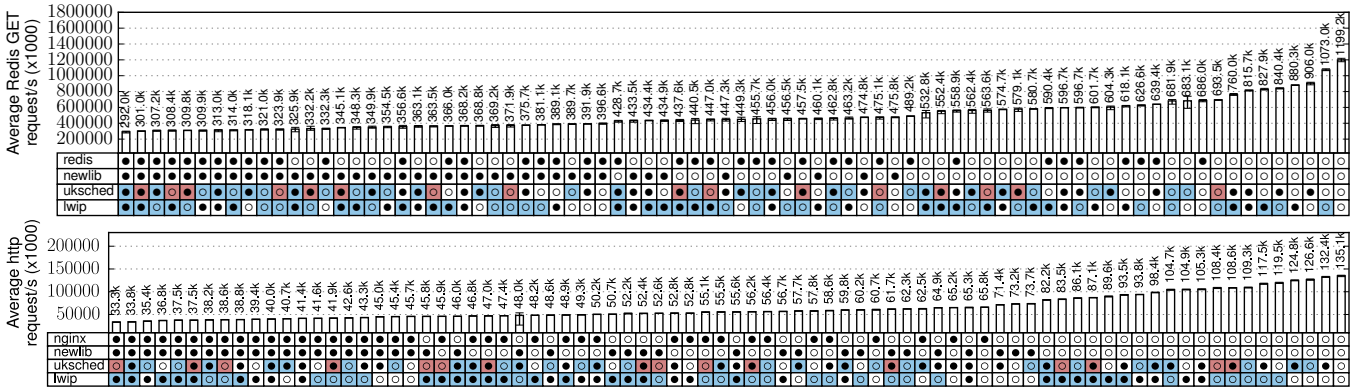

Figure 6: Redis (top) and Nginx (bottom) performance for a range of configurations. Components are on the left. Software hardening can be enabled [•] or disabled [○] for each component. The white/blue/red color indicates the compartment the component is placed into. Isolation is achieved with MPK and DSS.

We run experiments on an Intel Xeon Silver 4114 @2.2 GHz. For each experiment we use 4 cores from the same socket, isolated with *isolcpu*: 2 cores for the client (iPerf client/`redis-benchmark`, `wrk`) on the host, 1 core for the QEMU process, and 1 core per FlexOS' vCPU. Hyperthreading is disabled.

6.1 Design Space Exploration: Redis, Nginx

We automatically generate and run a large set of configurations for Redis and Nginx using the Wayfinder [38] benchmarking platform. We fix the isolation mechanism to MPK with DSS and vary: the number of compartments (1-3), compartmentalized components (TCP/IP stack, libc, scheduler, application), as well as per-compartment software hardening (stack protector, UBSan and KASan), for a total of 2x80 configurations.

Redis. The results are on Figure 6 (top), plotting for each configuration Redis' GET throughput. Overall we observe that FlexOS enables for a very wide range of safety configurations with significant performance variation: there is one order of magnitude of difference between the configuration yielding the lowest throughput (292K req/s) vs. the highest one (1.2M req/s).

Unsurprisingly, the configuration that disables isolation and hardening gives the highest throughput. Conversely, configurations with many compartments/hardening perform worst. Still, in between these two extremes, creating more compartments and enabling hardening has a variable impact on performance. For example, with two compartments and no hardening, isolating LwIP from the rest of the system leads to an 11% performance hit, while that number reaches more than 43% when the scheduler is the isolated component — indicating extensive communication between user code and the scheduler. The same is true for hardening: with a single compartment, enabling hardening on the scheduler has a 24% performance cost, while that cost is 42% when hardening the Redis application code.

The complexity of maximizing safety and performance becomes more clear when isolating several components: isolating LwIP from the scheduler from the rest only differs from a few percentage points from isolating LwIP together with the scheduler from the rest. Such "isolation for free" effects are caused by communication patterns; LwIP does not directly communicate with the scheduler, hence the "cut" is not on a hot path, and merging them in a same compartment brings little performance benefits. Thus, the performance does not entirely depend on the number of compartments or the number of components with hardening enabled, but rather *what* particular components are isolated/hardened, and their communication patterns. Such effects can be leveraged to maximize safety and performance.

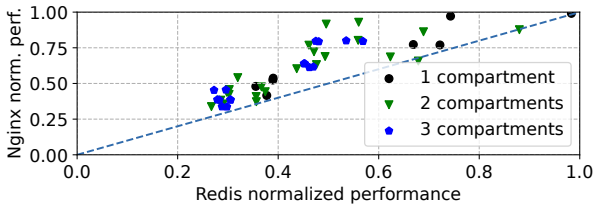

Figure 7: Nginx versus Redis normalized performance.

Nginx. The results are on Figure 6 (bottom), plotting for each configuration Nginx' HTTP throughput. Overall we observe that results span over the same range of overhead as Redis (0-4.1x). However, overheads do not follow the same distribution: 9 configurations have less than 20% overhead in the Nginx case, but only 2 for Redis. Similarly, 32 configurations have less than 45% of overhead, only 20 for Redis. This can be explained by looking more closely at individual configurations. Compared to Redis, isolating the scheduler is much less expensive (6% versus 43% for Redis), and the same goes for hardening (2% versus 24% for Redis). The costs, however, become similar as more hardening and isolation boundaries are added because of bottleneck effects.

This different distribution of costs is made more clear by Figure 7 which compares the relative performance of configurations for Nginx and Redis (same dataset as Figure 6). These differences show that isolating and hardening the *same components* on two networked

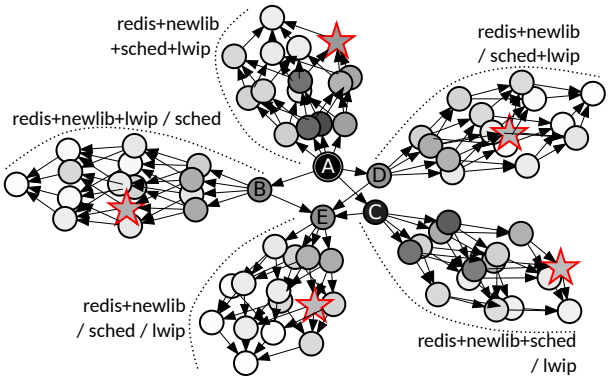

Figure 8: Configurations poset for the Redis numbers (Figure 6). Stars are the most secure configurations with performance >= 500k requests/s.

applications results in uneven, difficult to predict slow-down. Existing approaches assume a one-size fits all safety configuration are therefore suboptimal; in contrast, FlexOS enables users to easily navigate the safety / performance trade-off inherent in their application.

6.2 Partial Safety Ordering

We applied this technique on the Redis numbers from Figure 6. We construct the poset presented in Figure 8, where each node is a Redis configuration, i.e. a column from Figure 6. The node's color intensity indicates the configuration's performance, black being the fastest (1.2M req/s) and slower configurations becoming gradually white (pure white representing 292K req/s). The fastest configuration is the one with no isolation and no hardening (Ⓐ on Figure 8). Other nodes in the center of the plot represent compartments addition, still with no hardening: separating from the rest of the system either the scheduler Ⓑ, lwip Ⓒ, or Redis+newlib Ⓓ, and a 3 compartments scenario Ⓔ. From these 5 basic compartmentalization strategies come out 5 "branches". The nodes in each branch represent various combinations of per-component software hardening. The nodes' color evolution indicate the variable performance impact of creating new compartments and stacking software hardening on components.

We set a minimum required performance of 500K req/s, and let partial safety ordering identify the safest configurations satisfying that constraint, indicated with stars on Figure 8. In this case, the technique can prune the configuration space from 80 to 5 configurations, helping the user easily pick the most appropriate one.

6.3 Batching Effects: Network Stack Throughput

We port a simple iPerf server to FlexOS and use it to measure the network performance of our system. We fix the compartmentalization to the following scenario: the iPerf application code is placed within a compartment, and the rest of the system (including the network stack) is placed in a second compartment. We apply no software hardening, and configure the iPerf server to pass buffers of varying sizes when calling `recv` on the socket. We measure the

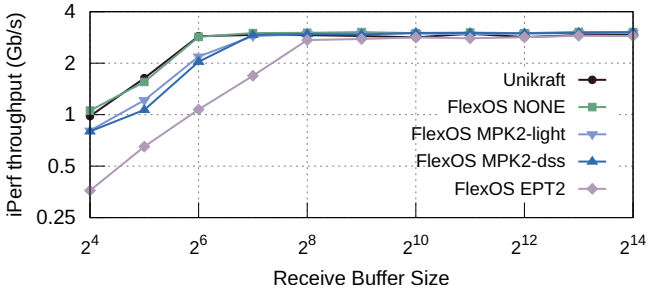

Figure 9: Network stack throughput (iPerf) with Unikraft (baseline), FlexOS without isolation, with two compartments backed by MPK (*-light* = shared call stacks, *-dss* = protected and DSS), and with two compartments backed by EPT.

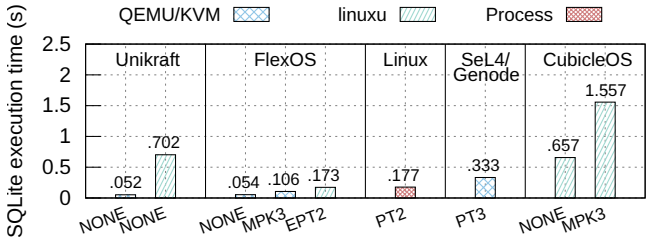

Figure 10: Time to perform 5000 INSERT queries with SQLite on Unikraft, FlexOS, Linux, SeL4 (with the Genode system), and CubicleOS. The isolation profile is shown on the x axis (NONE: no isolation, MPK3: MPK with three compartments, EPT2: two compartments with EPT, PT2/3: two/three compartments with page-table-based isolation).

achieved throughput using an iPerf client for FlexOS without isolation, with MPK (sharing or protecting the call stack), as well as EPT. We run vanilla Unikraft as baseline.

The results are on Figure 9. FlexOS without isolation performs similarly to Unikraft, confirming that users "only pay for what they get". FlexOS' isolation slowdown manifests for small payload sizes, for which the domain crossing latency is an important bottleneck in the request processing time. Depending on the buffer size, EPT isolation is 1.1-2.2x slower than MPK with DSS, which is itself 0-1.5x slower than the baseline without isolation. MPK with shared stacks bears a 0-1.3x slowdown. Although MPK with DSS pays the price of a stack switch (see Table 11b), it is more secure than fully sharing the stack and still faster than fully isolating it while moving shared data to the heap (see Figure 11a). Batching effects clearly manifest as the payload size increases: MPK's performance quickly becomes similar to to baseline starting from 128 B. EPT's isolation being more costly, the payload size needs to be 256 B or above so that its performance to reach about 90% of the baseline's. These results illustrate that, depending on the size of the payload and the frequency of domain crossings, all backends can constitute a valid solution to a given problem.

6.4 Filesystem Intensive Workloads: SQLite

We evaluate the performance of FlexOS with filesystem intensive workloads and compare it to vanilla Unikraft, Linux, SeL4 [45] with the Genode [25] system, and CubicleOS [74]. Although both FlexOS and CubicleOS extend Unikraft, the former runs in a standard Qemu/KVM VM while the latter is implemented on top of *linuxu*, Unikraft's Linux userland debug platform. The Unikraft baseline number thus cover both cases. We evaluate two scenarios: one with two components (EPT2, PT2), where the filesystem is isolated from the application, and one with three components (MPK3, PT3), where the filesystem is isolated from the time subsystem from the rest of the system. This benchmark performs 5000 INSERTs queries sequentially. To increase pressure on the filesystem, each query is in a separate transaction. The results are shown in Figure 10.

Compared to the baseline, FlexOS without isolation adds no overhead, and MPK3 adds an overhead of 2x. This is still significantly faster than the userland Linux version which performs a large number of system calls, highlighting the benefits of the LibOS basis. Somewhat surprisingly, FlexOS with EPT2 performs almost identically to Linux. This is because the syscall latency is almost identical to the EPT2 gate latency on this system (see Figure 11b). Compared to SeL4, FlexOS is 3.1x faster with MPK3, and 2x faster with EPT2.

Compared to CubicleOS, FlexOS is an order of magnitude faster. This is due to (1) CubicleOS relying on *linuxu*, i.e. running in Ring 3 and performing Linux system calls for privileged operations, (2) CubicleOS not implementing MPK support and relying on Linux `pkey_mprotect` system calls (making domain transitions orders of magnitude more expensive and the TCB thousands of times larger), and (3) CubicleOS' *trap-and-map* approach (that FlexOS avoids with shared data annotations). Even compared to its baseline without isolation, CubicleOS with MPK3 adds an overhead of 2.4x, about 30% more than FlexOS. CubicleOS without isolation is faster than the Unikraft linuxu baseline; this is because it uses the Lea [49] memory allocator which behaves better than Unikraft's TLSF [63] allocator in this benchmark.

6.5 Overheads: Stack Allocations, Gate Latencies

In FlexOS, stack data can be shared via heap allocations, using the DSS (trading space for performance), or sharing the stack entirely (trading safety for performance). To illustrate the benefits of the DSS, we measure, for each of the mechanisms, the execution time of a function that allocates 1 to 3 shared stack variables (size 1 Byte) and returns immediately.

The results are on Figure 11a. Heap-based stack allocations are one to two orders of magnitude (100-300+ cycles) slower than typical stack allocations (constant 2 cycles). This is not surprising, since general-purpose allocators typically feature unbounded execution time. This cost increases with the number of variables, since each variable triggers a separate call to malloc. The DSS matches the shared stack in performance, confirming that it combines the safety of isolation with the performance of traditional stack allocations. The memory footprint increase due to the DSS is modest as FlexOS uses small stacks (8 pages). For example, an instance with Redis (8 threads), has a space overhead of 288 KB. The DSS is a data sharing strategy and does not remove the need to perform stack switches.

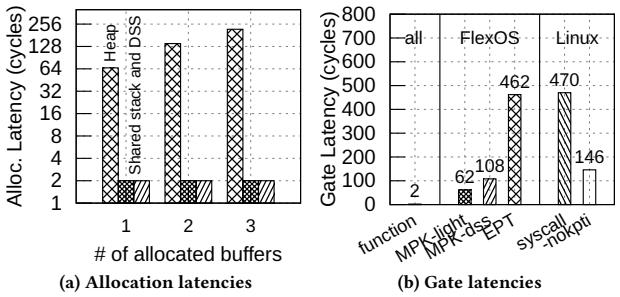

Figure 11: FlexOS latency microbenchmarks.

Another source of compartmentalization overhead is gate latency. To illustrate the raw performance of FlexOS' gates we measure the gate latency of MPK stack-sharing gates (*-light*), normal MPK gates, and EPT gates. We compare them with the latency of a function call, and of a Linux system call (with and without KPTI, *-nopkti*). The results are shown in Figure 11b. MPK light gates are 80% faster than normal MPK gates, and 7.6x faster than EPT gates, as they correspond to the cost of raw `wrprku` instructions. EPT latencies are similar to syscall latencies without KPTI, illustrating the practicability of the EPT backend.

7 USE CASES FOR ISOLATION FLEXIBILITY

FlexOS enables developers to seamlessly experiment with various safety configurations for their OS. An obvious use-case we presented throughout this paper is the specialization of the OS' safety strategy for a given application: manually or semi-automatically selecting, among the vast design space unlocked by FlexOS, the most suitable configuration for a particular use case with given safety/performance constraints. Still, there are many other ways in which this flexibility can be used; we detail some of them next.

Quickly Isolate Exploitable Libraries. Consider the period between the full disclosure of a vulnerability and the release of its fix, or the embargo period when vulnerabilities are disclosed only to affected vendors, but not to the general public; these periods can last for weeks up to years during which vulnerable software runs in the wild. With FlexOS, it takes seconds to create a new binary that isolates a vulnerable library into its own compartment (e.g. EPT + hardening) to at least mitigate the effects of exploits; an automated system could be created to respond to known vulnerabilities by recompiling production software to isolate certain libraries, similar to Self-Certifying Alerts [17]. Such flexibility improves over the state of the art by avoiding a loss of functionality (e.g. compared to Senx [35]), and providing excellent resistance to polymorphic variations of vulnerabilities (e.g. compared to filters [16]).

Quickly React to Hardware Protections Breaking Down. Recent hardware vulnerabilities [46, 56] showed that hardware-backed isolation mechanisms are not foolproof. The corresponding fixes may require significant engineering and redesign efforts (e.g. KPTI for Meltdown), leading to long vulnerability windows. FlexOS is

not immune to hardware vulnerabilities by design. In this case, however, its ability to easily switch between protection techniques comes handy: by supporting a wide range of isolation primitives relying on a range of different hardware, switching the isolation mechanism from a vulnerable to a non-vulnerable one is just a matter of rebuilding the LibOS with a different configuration (snippet in §3), i.e. the engineering cost is nil.

As Secure as You can Afford. Consider a service provider who wishes to offer the best possible security as long as its server can keep up with the client load. A natural approach would be to run the safest combination that copes with peak load, as we suggested in our Redis evaluation; this means that in periods of low load the system has idle compute power.

With its capacity to quickly switch safety configurations, FlexOS enables another approach: to run, at any time, the safest configuration that can sustain the actual load. This makes attacks much harder as long as the system is under-loaded, but gracefully switches off defenses as load increases to respect SLA. Another approach is to couple this with software load balancers to triage users into likely benign or malicious, sending them to machines running faster or safer software, accordingly.

Dealing with Crashed Software. Vulnerabilities are a fact of life, and the standard approach is to quickly restart crashed software and to examine the faults in the background. When such a crash is detected (e.g. memory error), with FlexOS it is wiser to start a safer configuration of the same software, to ensure that any vulnerability is not turned into an exploit.

Incremental Verification. Individual components of FlexOS can be verified and isolated from the rest of the system. In this way, one can obtain strong guarantees on pre-conditions and ensure that verified properties hold even when mixed with unverified components, something that isn't possible with monolithic operations systems [55]. Over time, the entire system could be verified, gradually increasing the guarantees of the system.

Deployment to Heterogeneous Hardware. The flexibility of FlexOS mechanisms can also come in very handy when deploying on heterogeneous hardware. Some servers might offer MPK support for example, others CHERI, others only the classical MMU. In every case, Chrysalis is able to get the best from the available hardware without major rewrite, and without requiring insider knowledge from application developers.

8 RELATED WORK

Improving OS safety. Previous work proposed to address the safety issues of monolithic OSes by reducing the TCB through separation [2, 73], micro-kernels [28, 34], and safe languages [7, 19, 36, 58, 66]. In SASOSes, internal isolation may be traded off for performance [41, 43, 47, 70], provided with traditional page tables [12, 33, 52, 68], or intra-AS hardware isolation mechanisms [54, 69, 74, 76]. Other research efforts strive to speedup IPC in microkernels [29, 64], or redesign monolithic OSes entirely [9, 11, 20, 31, 53, 67, 77].

Overall, each of these approaches is a single or a few point(s) in the OS safety/performance design space and lacks the flexibility of FlexOS to automatically specialize for safety or performance. LibrettOS [68] allows a LibOS to switch between SASOS and microkernel modes, but remains limited to a small subset of the safety/performance design space.

Compartmentalization Frameworks. Several compartmentalization frameworks have been proposed recently [6, 30, 32, 57, 65, 74, 75, 80]. Contrary to FlexOS, none focuses on flexible isolation. Regarding application porting, most [32, 65, 75, 80] rely on code annotations. A few studies provide various degrees of porting automation through data flow analysis [6, 30, 57], but are typically bound to numerous limitations due to the complexity of breaking down monolithic code bases. Nevertheless, some of their principles can be applied to increase the degree of automation of FlexOS' porting process – something we scope out as future works. CubicleOS [74] proposes a *trap and map* mechanism to limit the porting effort, but this comes at a high cost, is specific to MPK, and is not entirely automated. Further, as shown in our evaluation, CubicleOS' reliance on Unikraft's *linuxu* leads to suboptimal performance.

9 CONCLUSION

The isolation strategy of today's OSes is mostly fixed at design time. This lack of flexibility is problematic in many scenarios. We propose FlexOS, an OS whose isolation strategy is decoupled from its design. We augment the historical capacity of the LibOS to specialize towards performance with the ability to specialize for safety: fundamental decisions such as the compartmentalization granularity and which isolation mechanism to use are deferred to build time. FlexOS ships with a semi-automated exploration strategy helping the user navigate the vast configuration space the system unlocks. FlexOS is available online at https://project-flexos.github.io under an open source license.

In our future work, we intend to add more isolation backend implementations to FlexOS including CHERI and SGX, as well as support for more software hardening techniques. Another direction of future work is to create a formal basis to help users navigate the safety configuration space. This would enable, among others, embedding formally verified components in FlexOS configurations while preserving their proven properties.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers, and our shepherd, Gerd Zellweger, for their comments and insights. A similar thanks goes to our colleague Marc Rittinghaus for his insights, and to Julia Lawall for her invaluable help on Coccinelle. We are immensely grateful to the Unikraft OSS community for their past and ongoing contributions. This work was funded by a studentship from NEC Labs Europe, EU H2020 grants 825377 (UNICORE), 871793 (ACCORDION) and 758815 (CORNET), as well as the UK's EPSRC grants EP/V012134/1 (UniFaaS) and EP/V000225/1 (SCorCH). UPB authors were partly supported by VMWare gift funding.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact contains the source code of FlexOS, the proof-of-concept of our flexible isolation approach, along with all scripts necessary to reproduce the paper's measurements and plots. The goal of this artifact is to allow readers to reproduce the paper's results, and build new research on top of FlexOS.

A.2 Artifact Check-List (Meta-Information)

- **Program:** the FlexOS library OS, benchmarked with standard application benchmarks (`wrk` and `redis-benchmark`), a custom SQLite benchmark, and custom microbenchmarks.
- **Binary:** automatically built from source.
- **Run-time environment:** GNU/Linux Debian 11 (Bullseye), with KVM and Docker. Other dependencies are automatically installed.
- **Hardware:** Intel® Xeon® Silver 4114 @ 2.20 GHz, or any machine with more than 8 cores that supports Intel MPK, typically Intel® Xeon® Scalable Processors starting with the Skylake generation. At least 128.0 GB of RAM.
- **Metrics:** requests/s, Gb/s, queries/s, execution time, gate latencies.
- **Output:** performance data, FlexOS images.
- **Experiments:** Figures 6, 7, 9, 10, 11a and 11b are reproducible automatically. Figure 8 is reproducible manually (it is only a graph). Table 1 is also reproducible manually.
- **How much disk space required (approximately)?:** 100.0 GB
- **How much time is needed to prepare workflow (approximately)?:** 6-12 Hours (*automated*).
- **How much time is needed to complete experiments (approximately)?:** 4-5 Hours (*automated*), and up to 1.5 Hours (*manual*).
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** BSD-3-clause.
- **Workflow framework used?:** Wayfinder [38], Docker, scripts.
- **Archived (provide DOI)?:** `10.5281/zenodo.5748505`

A.3 Description

A.3.1 How to Access. The latest version of the artifact can found on GitHub[1]. Alternatively, individual releases can be downloaded from our Zenodo archive[2]. Note that the artifact evaluation (AE) GitHub repository only contains part of the artifact, namely scripts to reproduce this paper's experiments. The core of FlexOS, libraries, and applications, are all available in the `project-flexos` organization, as documented in the AE repository.

In order to precisely reproduce this paper's measurements, we gave ASPLOS'22 reviewers access to our server, an Intel® Xeon® Silver 4114 with 128.0 GB RAM, Debian 11.1, and Linux version `5.10.70-1`. Nonetheless, access to this particular setup is not required to run this artifact; hardware and software dependencies are detailed further below.

A.3.2 Hardware Dependencies. An Intel® Xeon® Silver 4114 @ 2.20 GHz, or any machine that supports Intel MPK, typically any Intel® Xeon® Scalable Processor starting with the Skylake generation. The processor must have more than 8 cores. 128.0 GB of RAM are necessary to run the experiments corresponding to Figure 6, as all images are built and stored in RAM by our tool in order to achieve reasonable preparation times. Note that this amount of cores/RAM is required to reproduce this paper's results, *not* to run FlexOS.

A.3.3 Software Dependencies. This artifact has been tested with Debian GNU/Linux 11 with Linux kernel version `5.10.70-1` (KVM enabled), Docker version `20.10.10` (or any recent version). All other dependencies are automatically installed by the artifact's scripts.

A.3.4 Data Sets. All data sets and benchmarks are included in the artifact, generated automatically, or downloaded automatically by the run scripts.

A.4 Installation

Before running any experiment, prepare your host with the recommendations detailed above in A.3.3. Note that all commands below assume superuser permissions. Once the system is set up, clone our AE repository:

```
$ git clone https://github.com/ukflexos/asplos22-ae.git
```

Then, generate a GitHub personal access token with the permissions "`public_repo`" and set it in the Makefiles. You can do it for the entire system by exporting an environment variable:

```
$ export KRAFT_TOKEN="<your token>"
```

Alternatively, you can also set it individually in every Makefile by editing the `KRAFT_TOKEN` variable:

```
...
#
# Parameters
#
KRAFT_TOKEN ?= <your token>
...
```

Note that if `KRAFT_TOKEN` is set system-wide, definitions in Makefiles will not override it. After this, install dependencies on the host:

```
$ make dependencies
```

A.5 Experiment Workflow

All experiments should be prepared first. The prepare step installs necessary tools and downloads additional resources before they can run. This can be done for a single experiment or for all experiments, for example:

```
$ make prepare-fig-07 # prepare experiment 7
$ make prepare # prepare all experiments
```

The automated preparation of all experiments takes on average 6-12 hours on our setup. This very long preparation time is due to the generation of all images. Once one or many experiments have been prepared they can be run, again using a similar syntax as above:

```
$ make run-fig-07 # run experiment 7
$ make run # run all experiments
```

[1] https://github.com/project-flexos/asplos22-ae
[2] https://zenodo.org/record/5748505

Running all automated experiments takes on average 4-5 hours on our setup. The plot for Figure 8 is not automated, and neither is the measurement of LoC changes for Table 1. We estimate that the combination of the two manual items may take up to 1.5 hours of manual work. Automated experiments will generate experimental results within the results folder of the specific experiment. To plot one or many experiment figures, use, for example:

```
$ make plot-fig-07 # plot experiment 7
$ make plot # plot all experiments
```

You can clean, or "properclean" to completely reset any preparation with `make clean` or `make properclean` for individual or all experiments, for example:

```
$ make clean-fig-07
$ make properclean-fig-07
$ make clean
$ make properclean
```

The clean rule removes results and plots, the properclean rule additionally deletes containers.

A.6 Evaluation and Expected Results

Reproducing experiments on the same machine should produce the same results as in the paper. On other machines, we expect different absolute numbers but similar ordering. On recent processors that benefit from hardware mitigations for transient executions attacks we expect EPT, Linux, and SeL4 measurements to improve comparatively to the MPK baseline.

A.7 Experiment Customization

Reviewers may use the base FlexOS Docker container to access a clean FlexOS development environment, port their own application, and build custom images. Instructions to build the base FlexOS Docker image, port applications, and build custom images are available in the `README.md` file of our main AE repository[3].

A.8 Notes

Some experiments have a slightly different workflow compared to the one described in A.5. Figure 6 requires you to set `HOST_CORES` with a set of cores to be used for the experiment. Figure 7 is only a plot and requires some manual steps. Figure 11b requires a reboot of the machine with different kernel parameters. Table 1 is manual. In all of these cases, the local `README.md` provides appropriate explanations. In general, the top-level and individual `README.md` files of our artifact contains more precise information on experiment timings, repository structure, setup requirements, and potential issues and solutions. We strongly recommend a careful read of these instructions before starting to reproduce experiments.

A.9 Methodology

Submission, reviewing and badging methodology:

- https://acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

[3] https://github.com/project-flexos/asplos22-ae/blob/main/README.md

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Trans. Inf. Syst. Secur.* 13, 1, Article 4 (2009). https://doi.org/10.1145/1609956.1609960

[2] 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). https://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.76.6810

[3] ARM Ltd. 2009. Building a Secure System using TrustZone Technology. https://developer.arm.com/documentation/genc009492/c. Online; accessed Jan 24, 2021.

[4] ARM Ltd. 2019. ARM Morello Program. https://developer.arm.com/architectures/cpu-architecture/a-profile/morello. Online; accessed June 25, 2020.

[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. https://dl.acm.org/doi/10.1145/3433210.3453111

[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. 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] Silas Boyd-Wickizer and Nickolai Zeldovich. 2010. Tolerating Malicious Device Drivers in Linux. In *2010 USENIX Annual Technical Conference (ATC'10)*. USENIX Association. https://dl.acm.org/doi/abs/10.5555/1855840.1855849

[10] John Bruno, José Brustoloni, Eran Gabber, Avi Silberschatz, and Christopher Small. 1999. Pebble: A Component-Based Operating System for Embedded Applications. In *Proceedings of the Embedded Systems Workshop (WOES'99)*. USENIX Association.

[11] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. 2009. Fast Byte-Granularity Software Fault Isolation. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)*. Association for Computing Machinery. https://doi.org/10.1145/1629575.1629581

[12] 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 (1994). https://doi.org/10.1145/195792.195795

[13] Stephen Checkoway and Hovav Shacham. 2013. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. Association for Computing Machinery. https://doi.org/10.1145/2451116.2451145

[14] The CScope contributors. [n. d.]. CScope: developer's tool for browsing source code. http://cscope.sourceforge.net/. Online; accessed December 22, 2021.

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

[16] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. 2007. Bouncer: Securing Software by Blocking Bad Input. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*. Association for Computing Machinery. https://doi.org/10.1145/1294261.1294274

[17] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. 2005. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*. Association for Computing Machinery. https://doi.org/10.1145/1095810.1095824

[18] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptol. ePrint Arch.* 2016, 86 (2016). https://eprint.iacr.org/2016/086.pdf

[19] 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 *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association. https://dl.acm.org/doi/10.5555/3291168.3291176

[20] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. 2015. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. Association for Computing Machinery. https://doi.org/10.1145/2694344.2694386

[21] 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. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*. Association for Computing Machinery. https://doi.org/10.1145/3297858.3304042

[22] Jack Edge. 2021. Rust for Linux redux. *Linux Weekly News* (2021). https://lwn.net/Articles/862018/.

[23] Jack Edge. 2021. Rust heads into the kernel? *Linux Weekly News* (2021). https://lwn.net/Articles/853423/.

[24] 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 (SOSP'95)*. Association for Computing Machinery. https://doi.org/10.1145/224056.224076

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

[26] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. 1997. The Flux OSKit: A Substrate for Kernel and Language Research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'97)*. Association for Computing Machinery. https://doi.org/10.1145/268998.266642

[27] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin J. Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. 2000. The SawMill Multiserver Approach. In *Proceedings of the 9th ACM SIGOPS European Workshop (EW 9)*. Association for Computing Machinery. https://doi.org/10.1145/566726.566751

[28] 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*.

[29] 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. In *2020 USENIX Annual Technical Conference (ATC'20)*. USENIX Association. https://www.usenix.org/conference/atc20/presentation/gu

[30] 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 (CCS'15)*. Association for Computing Machinery. https://doi.org/10.1145/2810103.2813611

[31] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. 1997. The Performance of μ-Kernel-Based Systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'97)*. Association for Computing Machinery. https://doi.org/10.1145/268998.266660

[32] 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 (ATC'19)*. USENIX Association. https://www.usenix.org/conference/atc19/presentation/hedayati-hodor

[33] 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 (1999). https://doi.org/10.1002/(SICI)1097-024X(19980725)28:9%3C901::AID-SPE181%3E3.0.CO;2-7

[34] 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 (2006). https://doi.org/10.1145/1151374.1151391

[35] Zhen Huang, David Lie, Gang Tan, and Trent Jaeger. 2019. Using Safety Properties to Generate Vulnerability Patches. In *2019 IEEE Symposium on Security and Privacy (S&P'19)*. https://doi.org/10.1109/SP.2019.00071

[36] Galen C. Hunt and James R. Larus. 2007. Singularity: Rethinking the Software Stack. *SIGOPS Oper. Syst. Rev.* 41, 2 (2007). https://dl.acm.org/doi/10.1145/1243418.1243424

[37] Intel Corporation. 2021. Intel® 64 and IA-32 Architectures Software Developer's Manual. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. Volume 3A, Section 4.6.2.

[38] Alexander Jung, Hugo Lefeuvre, Charalampos Rotsos, Pierre Olivier, Daniel Oñoro-Rubio, Mathias Niepert, and Felipe Huici. 2021. Wayfinder: Towards Automatically Deriving Optimal OS Configurations. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys'21)*. https://doi.org/10.1145/3476886.3477506

[39] 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 16th ACM symposium on Operating systems principles*. https://dl.acm.org/doi/10.1145/268998.266644

[40] Svilen Kanev, Sam Likun Xi, Gu-Yeon Wei, and David Brooks. 2017. Mallacc: Accelerating Memory Allocation. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*. Association for Computing Machinery. https://doi.org/10.1145/3037697.3037736

[41] Antti Kantee. 2012. Flexible Operating System Internals: The Design and Implementation of the Anykernel and Rump Kernels. http://urn.fi/URN:ISBN:978-952-60-4917-5.

[42] Douglas Kilpatrick. 2003. Privman: A Library for Partitioning Applications. In *USENIX Annual Technical Conference, FREENIX Track (ATC'03)*. https://www.usenix.org/legacy/events/usenix03/tech/freenix03/kilpatrick.html

[43] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. 2014. OSv Optimizing the Operating System for Virtual Machines. In *2014 USENIX Annual Technical Conference (ATC'14)*. USENIX Association. https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity

[44] Chris Kjellqvist, Mohammad Hedayati, and Michael L. Scott. 2020. Safe, Fast Sharing of Memcached as a Protected Library. In *Proceedings of the 49th International Conference on Parallel Processing (ICPP'20)*. Association for Computing Machinery, Article 6. https://doi.org/10.1145/3404397.3404443

[45] 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 (SOSP'09)*. Association for Computing Machinery. https://doi.org/10.1145/1629575.1629596

[46] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2019. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. https://doi.org/10.1109/SP.2019.00002

[47] 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 (EuroSys'21)*. Association for Computing Machinery. https://doi.org/10.1145/3447786.3456248

[48] Julia Lawall and Gilles Muller. 2018. Coccinelle: 10 years of automated evolution in the Linux kernel. In *2018 USENIX Annual Technical Conference (ATC'18)*. https://dl.acm.org/doi/10.5555/3277355.3277413

[49] Doug Lea. 1996. A Memory Allocator. http://gee.cs.oswego.edu/dl/html/malloc.html.

[50] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Stefan Teodorescu, Pierre Olivier, Tiberiu Mosnoi, Răzvan Deaconescu, Felipe Huici, and Costin Raiciu. 2021. FlexOS: Making OS Isolation Flexible. In *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS'21)*. https://sigops.org/s/conferences/hotos/2021/

[51] K Leino and Rustan M. 2010. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer. https://link.springer.com/chapter/10.1007%2F978-3-642-17511-4_20

[52] 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). https://doi.org/10.1109/49.536480

[53] 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)*. USENIX Association. https://dl.acm.org/doi/10.5555/1251254.1251256

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

[55] Jialin Li, Samantha Miller, Danyang Zhuo, Ang Chen, Jon Howell, and Thomas Anderson. 2021. An Incremental Path towards a Safer OS Kernel. In *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS'21)*. Association for Computing Machinery. https://doi.org/10.1145/3458336.3465277

[56] 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 *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)*. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[57] 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 (CCS'17)*. Association for Computing Machinery. https://doi.org/10.1145/3133956.3134066

[58] 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. https://dl.acm.org/doi/10.1145/2451116.2451167

[59] 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. https://rd.springer.com/chapter/10.1007/978-3-540-40965-6_2

[60] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. Association for Computing Machinery. https://dl.acm.org/doi/abs/10.1145/3132747.3132763

[61] Ilias Marinos, Robert N.M. Watson, and Mark Handley. 2014. Network Stack Specialization for Performance. In *Proceedings of the ACM SIGCOMM 2014 Conference (SIGCOMM'14)*. Association for Computing Machinery. https://doi.org/10.1145/2619239.2626311

[62] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the Art of Network Function Virtualization. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*. USENIX Association. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/martins

[63] M. Masmano, I. Ripoll, A. Crespo, and J. Real. 2004. TLSF: a new dynamic memory allocator for real-time systems. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS)*. https://doi.org/10.1109/EMRTS.2004.1311009

[64] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. 2019. SkyBridge: Fast and Secure Inter-Process Communication for Microkernels. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys'19)*. Association for Computing Machinery, Article 9. https://doi.org/10.1145/3302424.3303946

[65] 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. https://www.usenix.org/conference/usenixsecurity20/presentation/narayan

[66] 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

[67] Ruslan Nikolaev and Godmar Back. 2013. VirtuOS: An Operating System with Kernel Virtualization. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. Association for Computing Machinery. https://doi.org/10.1145/2517349.2522719

[68] 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 (VEE'20)*. Association for Computing Machinery. https://doi.org/10.1145/3381052.3381316

[69] 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* (2020). https://www.ssrg.ece.vt.edu/papers/spma20.pdf

[70] 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 (VEE 2019)*. Association for Computing Machinery. https://doi.org/10.1145/3313808.3313817

[71] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. 2008. Documenting and Automating Collateral Evolutions in Linux Device Drivers. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (Eurosys'08)*. Association for Computing Machinery. https://doi.org/10.1145/1352592.1352618

[72] Gabriel Parmer and Richard West. 2007. Mutable Protection Domains: Towards a Component-Based System for Dependable and Predictable Computing. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS'07)*. https://doi.org/10.1109/RTSS.2007.27

[73] J. M. Rushby. 1981. Design and Verification of Secure Systems. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP'81)*. Association for Computing Machinery. https://doi.org/10.1145/800216.806586

[74] Vasily A. Sartakov, Lluis Vilanova, and Peter Pietzuch. 2021. CubicleOS: A Library OS with Software Componentisation for Practical Isolation. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*. Association for Computing Machinery. https://dl.acm.org/doi/10.1145/3445814.3446731

[75] 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. https://www.usenix.org/conference/usenixsecurity20/presentation/schrammel

[76] 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 (VEE'20)*. Association for Computing Machinery. https://doi.org/10.1145/3381052.3381326

[77] Michael M. Swift, Steven Martin, Henry M. Levy, and Susan J. Eggers. 2002. Nooks: An Architecture for Reliable Device Drivers. In *Proceedings of the 10th ACM SIGOPS European Workshop (EW 10)*. Association for Computing Machinery. https://doi.org/10.1145/1133373.1133393

[78] A. S. Tanenbaum, J. N. Herder, and H. Bos. 2006. Can we make operating systems reliable and secure? *Computer* 39, 5 (2006). https://doi.org/10.1109/MC.2006.156

[79] The Linux Kernel Development Community. 2020. The Kernel Address Sanitizer (KASAN). https://www.kernel.org/doc/html/v5.10/dev-tools/kasan.html. Online; accessed Jan, 25 2021.

[80] 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. https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner

[81] Robert NM Watson, Peter G Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alex Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. 2021. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8)*. Technical Report. University of Cambridge. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-951.pdf

[82] 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. https://doi.org/10.1109/SP.2015.9

[83] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Simon W. Moore, Steven J. Murdoch, and Michael Roe. 2014. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-set architecture*. Technical Report UCAM-CL-TR-864. University of Cambridge, Computer Laboratory. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-864.pdf

[84] 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 (ATC'18)*. USENIX Association. https://www.usenix.org/conference/atc18/presentation/zhang-yiming