CubicleOS: A Library OS with Software Componentisation for Practical Isolation

Vasily A. Sartakov
v.sartakov@imperial.ac.uk
Imperial College London
United Kingdom

Lluís Vilanova
vilanova@imperial.ac.uk
Imperial College London
United Kingdom

Peter Pietzuch
prp@imperial.ac.uk
Imperial College London
United Kingdom

ABSTRACT

Library OSs have been proposed to deploy applications isolated inside containers, VMs, or trusted execution environments. They often follow a highly modular design in which third-party components are combined to offer the OS functionality needed by an application, and they are customised at compilation and deployment time to fit application requirements. Yet their monolithic design lacks isolation across components: when applications and OS components contain security-sensitive data (e.g., cryptographic keys or user data), the lack of isolation renders library OSs open to security breaches via malicious or vulnerable third-party components.

We describe CubicleOS, a library OS that isolates components in the system while maintaining the simple, monolithic development approach of library composition. CubicleOS allows isolated components, called *cubicles*, to share data dynamically with other components. It provides spatial memory isolation at the granularity of function calls by using Intel MPK at user-level to isolate components. At the same time, it supports zero-copy data access across cubicles with feature-rich OS functionality. Our evaluation shows that CubicleOS introduces moderate end-to-end performance overheads in complex applications: 2× for the I/O-intensive NGINX web server with 8 partitions, and 1.7–8× for the SQLite database engine with 7 partitions.

CCS CONCEPTS

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

KEYWORDS

compartments, isolation, inter-process communication, Intel MPK

ACM Reference Format:
Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. 2021. CubicleOS: A Library OS with Software Componentisation for Practical Isolation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3445814.3446731

ASPLOS '21, April 19–23, 2021, Virtual, USA

ACM ISBN 978-1-4503-8317-2/21/04…$15.00
https://doi.org/10.1145/3445814.3446731

1 INTRODUCTION

In cloud environments, library OSs gain popularity when users want to make deployed applications self-contained in terms of OS functionality. They are used to deploy lightweight unikernels [29, 31, 37], make containers more efficient [47], and run shielded applications inside of trusted execution environments (TEEs) [7, 44].

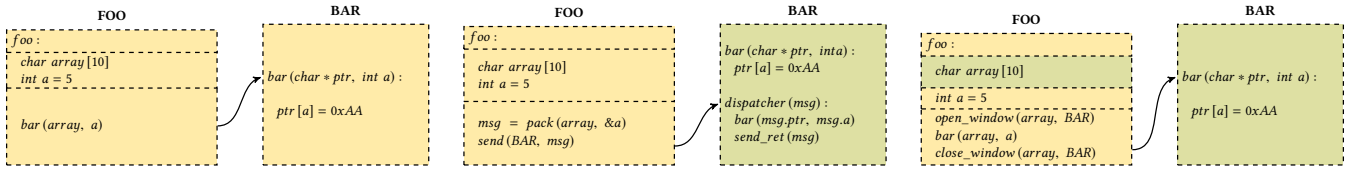
Library OSs such as Graphene [53], IncludeOS [5] and Unikraft [31] are typically assembled from various independent library components (e.g., file system libraries, network stacks, and low-level drivers). To minimise the image size when linked with an application, the components required for a given application are selected at compile-time. All components and the application then execute as part of a single, unprotected address space.

The lack of compartmentalisation between library OS components raises security, robustness and reliability concerns; these are well-known deficiencies of monolithic designs, especially given the complexity of library OS components and when applications are exposed over the network [39]. For example, a vulnerability in a file system implementation may be exploited to compromise the whole library OS and application, and then disclose, e.g., encrypted keys from the TLS implementation [12].

We argue that current library OS designs therefore risk being a step backwards in terms of security, robustness and reliability. In contrast to the well-known monolithic design, microkernel designs [22, 30, 34, 51] impose standard interfaces between kernel components (e.g., based on message passing or RPC-like calls), which can be used to enforce protection boundaries between components. We observe that such designs have seen limited uptake in library OSs, which typically strive for full POSIX compatibility, e.g., to execute current Linux applications. In addition, the extra data copies imposed by microkernel interfaces adds further to the overhead of library OS designs.

The research question that we explore in this paper is whether it is possible to design a modular and compartmentalized library OS that consists of existing, third-party components, while enforcing practical isolation between these components. Our goal is to retain the flexibility of arbitrary in-kernel interfaces between components, as found in monolithic kernel designs, while enforcing spatial and temporal memory isolation with an acceptable performance overhead. To be practical, this must be achieved without invasive source code changes to the application or the library OS, and retain compatibility with feature-rich (Linux) applications.

We describe *CubicleOS*, a new library OS with three core contributions: (1) it mutually isolates existing, third-party components to provide data integrity and privacy; (2) it imposes minimal code changes to express isolation policies for components; and (3) it



(a) Direct function call **(b) Message-based interface** **(c) Windows in CubicleOS**

Figure 1: Different designs for interface isolation (Colours represent different memory protection domains.)

provides an efficient implementation based on existing hardware with trivial modifications.

CubicleOS offers three core abstractions that together enforce the memory isolation policies defined by each component: cubicles, windows, and cross-cubicle calls. *Cubicles* provide transparent spatial memory isolation for each library OS and application component, such that any attempt to read or write from/to memory of another cubicle leads to a memory protection fault. *Windows* provide user-managed temporal memory isolation, such that cubicles can temporarily share data with each other without any copying (e.g., as function arguments). Finally, *cross-cubicle calls* provide control-flow integrity (CFI) when calling functions across cubicle boundaries, ensuring only public entry points are used while enforcing memory isolation policies.

We prototype CubicleOS on top of Unikraft [31], an existing library OS. Developers simply need to manage CubicleOS' windows to grant memory accesses across cubicles. CubicleOS' build system automatically identifies the public entry points of each component and generates trusted cross-cubicle call trampolines for each. CubicleOS enforces memory isolation with low overheads using Intel's *Memory Protection Keys* (MPK) extensions [25]; it maps each cubicle into a separate MPK tag and dynamically manages the access permissions for windows, requiring only a minor modification to MPK hardware to ensure the integrity of cross-cubicle calls.

When a cross-cubicle call accesses arguments passed by another cubicle via a window, CubicleOS reassigns that page's tag to the accessing cubicle. Note that this differs from a typical use of MPK in which separate tags are set up for shared communication buffers [21, 54]. This would require changing component interfaces to copy memory to/from the shared buffers, and result in indirect overheads due to data copies and exhaustion of MPK tags. CubicleOS avoids these issues by dynamically retagging pages, and judiciously doing so only when necessary.

We evaluate CubicleOS both in terms of development effort for third-party library OS and application developers, and performance. Our experimental results shows that CubicleOS is between 3× and 5× faster than a state-of-the-art microkernel for the SQLite database engine with an isolated file system stack, with a small developer effort that does not impact the organisation or interfaces of existing third-party OS and application code (SQLite: 600 SLOC; NGINX: 400 SLOC). CubicleOS with SQLite is 1.7× to 8× slower compared to a non-isolated version and, for the I/O-intensive NGINX web server, it is 2× slower.

2 ISOLATION IN LIBRARY OSS

Here we discuss the trade-offs between different approaches to isolate interacting components in a library OS, and then look at recent hardware support that makes new approaches possible.

2.1 Interfaces between Components

We consider three mechanisms for communication between isolated components, as shown in Figure 1: (a) direct functions calls between components; (b) message-based interfaces, as employed by microkernels; and (c) the *window*-based approach used by CubicleOS. The figures consider an example with two components, `FOO` and `BAR`, that interact with each other. Each component has one function: `foo()` is located inside `FOO`, and `bar()` is located inside `BAR`. Function `foo()` has two stack variables, an array of 10 bytes, `array[10]`, and an integer, `a=5`. Function `foo()` invokes function `bar()` and passes the pointer to the array and the integer.

Figure 1a shows a regular function call within a single address space. Function `bar()` is called with two arguments, a pointer to `foo`'s stack and a scalar value. Function `bar()` has access to the `foo`'s stack, and therefore can directly change the array (`ptr[a]=0xAA`). In a library OS, trampolines into the kernel have similar semantics because the kernel has privileged access to the user program. Such calls are fast, but cannot be used for the interaction between *isolated* components. For example, if component `BAR` is in a separate protection domain, any attempt to access `FOO`'s stack would cause a memory protection fault.

In contrast, Figure 1b shows a general approach for a message-based interface, which is commonly used in microkernel designs. Here, the function call is realised by sending messages. Function `foo` prepares a message (`pack()`) and requests the kernel to send it to the callee. At the callee's side, there is a message dispatcher, which retrieves arguments from a message register (i.e., a thread control block) and calls the actual function with the passed arguments.

Such an approach incurs an overhead due to data marshalling, switching between the caller, the kernel, and the callee, and sending the result back. In addition, from a developer's point-of-view, the use of message-based interfaces may require these interfaces to be defined in interface description languages (e.g., MIG [14]) or the use of particular serialisation mechanisms (e.g., Genode [17]).

We want to explore an interface approach that combines the efficiency and flexibility of direct function calls with the isolation properties of message-based implementations. The idea is to have

a new mechanism that is suitable for communication between isolated components but that does not require the use of messages or an interface description language.

Figure 1c shows the idea of *window-based* interaction between components, as employed by CubicleOS. Calls have the same semantics as direct function calls: e.g., the caller can pass a pointer and a scalar value to the callee, and the callee, in turn, directly accesses the passed values. This becomes possible because individual memory pages are assigned to protection domains (indicated by different colours). Before the invocation of function `bar()`, component `FOO` makes the memory pages with the array accessible to `BAR`. After the call, the caller revokes the access permissions to the stack variable, and the components are again fully isolated.

This window-based approach trades the need for designing interfaces around the more restrictive message passing style (Figure 1b) for maintaining existing software interfaces, while adding explicit memory grant management operations (Figure 1c). In the rest of the paper, we explore the feasibility of this for a real-world library OS.

2.2 Memory Protection with Intel MPK

The window-based interaction between components requires efficient fine-grained control over page permissions. We propose to use Intel's Memory Protection Keys (MPK) [25], an ISA extension that manages access permissions on groups of pages. It was first introduced in Intel's Skylake architecture, and it is similar to protection keys in Itanium [24] and access identifiers in PA-RISC [23].

MPK assigns a 4-bit key to each virtual page by extending the page table structures, and adds a new 64-bit `pkru` register that defines the access permissions to all pages on a key. It supports 16 different keys, each with a 2-bit access permission field, which encodes if a key is granted read and/or write access to all pages with that key.

Each process gets its own set of 16 keys, and each thread can have different access permissions to each key. Assigning keys to pages is a protected operation that takes more than 1,100 cycles in Linux [43] (using the `pkey_mprotect` system call), but the `pkru` register is available to user-level code, and applications can change its value in around 20 cycles [43] (using the `wrpkru` instruction).

MPK is suited for window-based interfaces because of its ability to quickly change the access permissions on entire groups of pages. Yet, there are three challenges that CubicleOS must overcome before it can use MPK for window-based interfaces (see Section 5): (i) MPK is limited to 16 keys; (ii) MPK does not control access to the `wrpkru` instruction (i.e., it can be invoked by untrusted user code); and (iii) MPK does not control the ability to execute pages, which is only defined on a per-page basis in the page table.

2.3 Threat Model

Our goal is to protect the confidentiality and integrity of library OS and application components via compartmentalisation, as defined by their respective developers. We assume a software attacker who controls the source code of one or more components outside the TCB; the adversarial source code includes third-party OS as well as application components. We also assume an attacker who controls external inputs to the application (e.g., via network packets).

We trust the developer who defines the isolation policy, and compiles and deploys the system. Our TCB also includes a *component builder* tool, which identifies the public symbols of each component after compilation, and a *memory monitor* and *cubicle loader*, which enforce memory access and control flow policies across components (as will be described in Sections 3 to 5). We also trust the hardware infrastructure and underlying virtualisation environment (since library OSs are typically deployed in VMs or containers). We do not target the validity of arguments or function call sequences used across isolation compartments (cubicles), but prevent components from undermining any invariant that is maintained by such components. We also do not target malicious operators, side-channel attacks, or full protection against external input attacks, which can be handled by complementary work [1, 2, 9, 11, 15, 27].

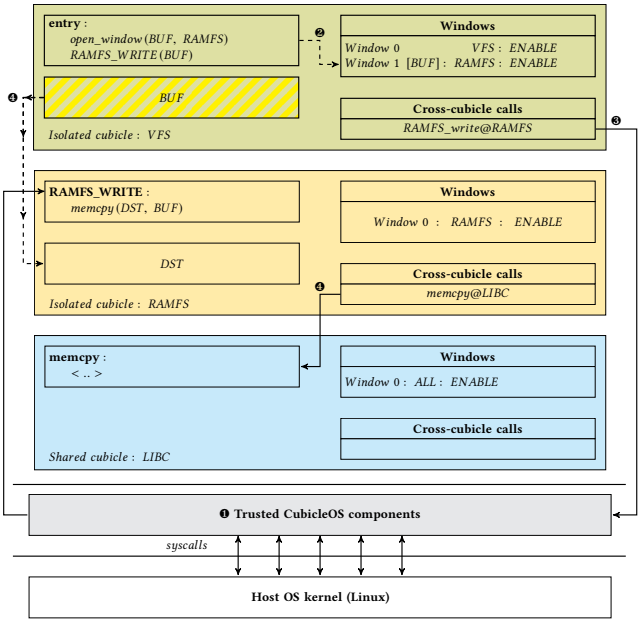


Figure 2: Function calls and memory accesses in CubicleOS using *cubicles*, *windows*, and *cross-cubicle calls*

3 CUBICLEOS OVERVIEW

CubicleOS is a library OS that protects against unauthorised interference between its components, including OS services and applications. Protection is provided by three core elements: *cubicles*, *windows*, and *cross-cubicle calls*, which provide spatial memory isolation, temporal memory isolation, and control flow integrity, respectively. By compartmentalizing OS and application components, we can build a system with improved security, robustness and reliability.

CubicleOS runs on top of an existing host OS such as Linux and fully isolates each component in a separate cubicle, which CubicleOS translates into per-cubicle MPK tags. Each component in CubicleOS is compiled as a dynamic library, and the build process automatically generates a trusted code thunk for each public function in a component to switch execution across cubicles. These code thunks, known as cross-cubicle calls, ensure that untrusted components only interact via their intended interfaces. This allows

CubicleOS to enforce that they cannot escape their isolation boundaries by directly manipulating page or MPK permissions. Finally, a component developer must modify their code to manage CubicleOS windows, dynamically granting access to a cubicle's data through cross-cubicle calls.

The core argument that we make with CubicleOS is that windows allow library OS developers to use existing component interfaces with little development effort, while providing strong isolation and system performance.

Figure 2 shows an example, later used in the evaluation, with three components: VFS, RAMFS and LIBC. The `VFS_write` function in VFS (called by the application) writes the contents of `BUF` into the file system backend, which in turn uses LIBC's `memcpy` to copy the data across components (and, therefore, cubicles).

❶ CubicleOS starts by loading each component into separate, **isolated cubicles**, which contain their respective code, data, heap, and stack memory pages. Cubicles provide *spatial memory isolation*: each is considered to *"own"* its memory, and cannot access the memory of other cubicles unless it has been explicitly authorised to do so (see below). Existing library OS implementations are typically componentised along semantic and API lines to allow compile-time configurability [31], and CubicleOS exploits this to delineate isolation boundaries that match existing code.

❷ VFS then *"opens"* a **window** to BUF for RAMFS (window 1 in VFS), allowing RAMFS to access the memory buffer that VFS wants to write into the file system. Windows define the policies for *temporal memory isolation* of a cubicle's own memory, and are enforced by CubicleOS's TCB. Each window contains a set of memory ranges in a cubicle, and the set of other cubicles that can access them at any point in time, allowing cubicles to exchange data that they own via regular function call arguments (i.e., pointers). As illustrated in Figure 2, each cubicle has an implicit *window 0* that gives it access to all pages that it owns.

❸ VFS then uses a **cross-cubicle call** to invoke the `RAMFS_write` function, effectively switching execution across cubicles. Cross-cubicle calls ensure control flow integrity (CFI) because each must go through CubicleOS's TCB to switch cubicle permissions. After the call, CubicleOS gives this thread access to all open windows for the RAMFS cubicle – in this case, the implicit window 0 in RAMFS, as well as window 1 in VFS –, switch across per-cubicle stacks, and start executing the target `RAMFS_write` function (function returns across cubicles are handled in a similar way).

❹ Finally, RAMFS calls `memcpy` in the **shared cubicle** LIBC to perform the actual memory copy from VFS' `BUF` into RAMFS' `DST` buffers. Shared cubicles such as LIBC are used in cases in which components contain little state and are frequently used by other components. Static data in shared cubicles, e.g., pre-allocated global buffers and variables, is shared among all cubicles. Calls to a shared cubicle never involve CubicleOS' runtime TCB, effectively executing with the privileges, stack and heap of their calling cubicle. Therefore, the call to `memcpy` will execute with the privileges of RAMFS, with access to both the source and destination buffers on the VFS and RAMFS cubicles, respectively.

Together, cubicles and windows ensure spatial and temporal isolation, respectively, across components in the system, providing the equivalent of dynamic, cubicle-centric memory access control with minimal developer involvement. In addition, cross-cubicle calls ensure CFI by unequivocally tracking the currently-executing cubicle and enforcing that only known, public functions are used to switch memory access permissions across cubicles.

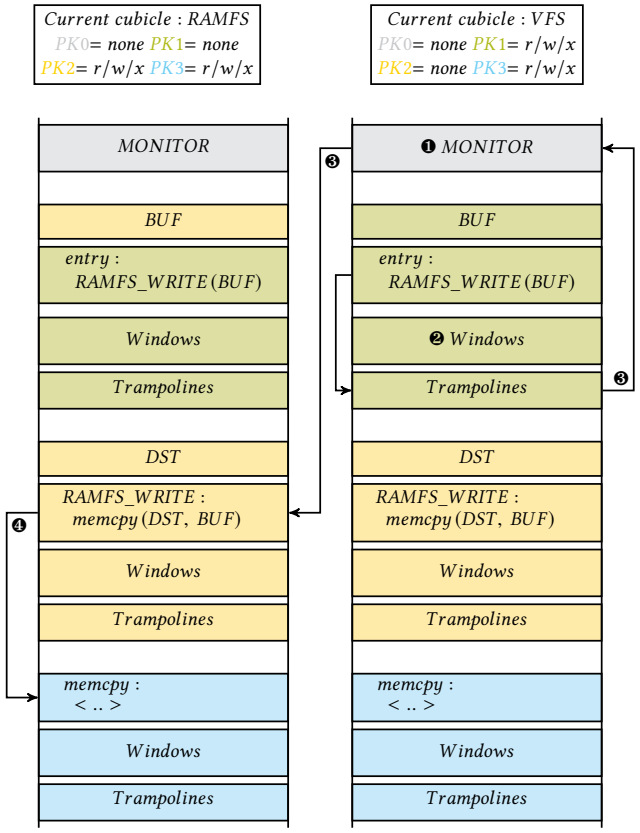


Figure 3: Mapping cubicles and windows into memory protection keys (MPKs) in CubicleOS

4 CUBICLEOS DESIGN AND API

To enforce the desired isolation policies, CubicleOS has four trusted components: (i) the *component builder*, (ii) the *cross-cubicle call trampolines*, (iii) the *memory monitor*, and (iv) the *cubicle loader*.

The **component builder** identifies each component in the system as well as their public functions, and generates for each function a *cross-cubicle call trampoline*. The trampoline is an auto-generated, trusted code thunk that provides cross-cubicle calls, and has three tasks performed at call and return: switching the memory access permissions across cubicles via the trusted memory monitor (step ❸ in Figure 2), as well as switching stack pointers and copying in-stack arguments when the caller and callee cubicles use different stacks.

The **memory monitor** is a trusted cubicle with an interface to manage permissions and window ownership. It offers stack-/heap allocation primitives that assign pages to the calling cubicle (each isolated cubicle has its own memory sub-allocator), as well as the CubicleOS-specific API in Table 1. A cubicle creates a window using `cubicle_window_create`, and adds/removes owned memory via `cubicle_window_add` and `cubicle_window_remove`. It can use

Table 1: CubicleOS-specific API available to untrusted code

API function	Description
`wid_type` **`cubicle_window_init`**`()`	Initialise an empty window
`cubicle_window_add``(wid_type wid, void *ptr, size_t size)`	Associate memory range *(*`ptr`*,* `ptr+size`*)* to window `wid`
`cubicle_window_remove``(wid_type wid, void *ptr)`	Remove a memory range previously associated to window `wid`
`cubicle_window_open``(wid_type wid, cid_type cid)`	Allow cubicle `cid` to access contents of window `wid`
`cubicle_window_close``(wid_type wid, cid_type cid)`	Disallow cubicle `cid` to access contents of window `wid`
`cubicle_window_close_all``(wid_type wid)`	Disallow all accesses to `wid` from other cubicles
`cubicle_window_destroy``(wid_type wid)`	Destroy window `wid`

`cubicle_window_open` to grant another cubicle access to that window's contents (step ❷ in Figure 2); `cubicle_window_close` does the opposite. Note that windows are assigned to the calling cubicle, and can only be managed by it.

To ensure the integrity of the isolation mechanisms, code can only be loaded by the **cubicle loader**. The loader takes a set of pages owned by a cubicle, containing the code and data of a component to load into the system, and switches their ownership into a newly-created cubicle. This is similar to how `dlopen` loads program and library images into a system, with two additional rules: (1) pages identified as code are given execute-only permissions, data pages are given read or read-write permissions (as specified by the binary), and CubicleOS does not allow cubicles to change the execution permissions of any page; and (2) the loader scans the code pages to ensure that they do not contain any instructions that would affect the integrity of the isolation mechanisms, i.e., cubicles cannot directly execute system calls nor MPK-related operations. This is similar to other prior work [21, 54] (see Section 5 for further details).

5 IMPLEMENTATION OF CUBICLEOS

We implement CubicleOS on top of the Unikraft framework [31], using its modular architecture as the basis for automatically identifying and isolating cubicles. We also use Intel MPK [25] to enforce the memory access policies expressed by cubicles and windows, dynamically assigning memory keys to open windows. MPK allows us to implement cross-component calls efficiently by switching the set of access permissions for each memory key in a per-thread basis, ensuring that only CubicleOS's TCB can assign memory keys and change their access permissions.

We first give an overview of how we adapt Unikraft to build CubicleOS, and then describe the implementation of CubicleOS' trusted components and how they use Intel MPK to enforce isolation.

It is important to note that MPK was intended as a user-level mechanism for hardening the security of applications, and thus has limitations for CubicleOS that we address as follows:

- We assume that the host OS, CubicleOS and compilation toolchains are bug free, and that the developer building the system is trusted (even if the components are not).
- We ensure that untrusted components cannot perform MPK-related operations by parsing the binaries at load time; this includes the `wrpkru` system call instructions (the latter could otherwise be used to instruct the host OS to modify the per-page MPK tags).
- We assume that the `wrpkru` instruction will revoke the execution attribute in the future, and describe a trivial hardware modification to do so.

5.1 Unikraft Architecture

Unikraft [31] is a framework for building unikernels. It has a modular architecture in which each component implements a single OS function (the virtual file system, file system backends, memory allocation, network stack, etc.). Components are selected at compile-time and linked into a monolithic image together with the application.

Components in Unikraft interact by referencing each other's function and data symbols (resolved at dynamic link time), either by directly using a symbol of another component, or by using a callback table that is filled-in by component at initialisation time (e.g., the RAMFS component initialises a callback table defined by the VFS component to export file system backend-specific functions; similar cases arise with network device drivers or memory allocators).

5.2 Builder: Piggy-Backing on Unikraft Components

The builder in CubicleOS is a tool that extends Unikraft's build logic to produce the information necessary for CubicleOS. It performs the following tasks:

(1) It compiles each component in Unikraft as a separate dynamic library. During this process, the developer specifies for each component whether it is an isolated or a shared cubicle (in practice, a file is generated that enforces this when the system is deployed).
(2) It identifies functions used across cubicles. As part of Unikraft, each symbol exported by a component has an entry in an `exportsyms.uk` file. In the case of callback tables, we modify the source code of a component to ensure that the pointer on each callback is resolved as a dynamic symbol at load time. This way CubicleOS' loader can interpose a cross-cubicle call trampoline.
(3) It generates a cross-cubicle call trampoline for each such function (see Section 5.5). The builder takes the symbols in `exportsyms.uk`, parses the corresponding function definition to extract its signature, and generates a cross-cubicle call trampoline for that symbol. The generated trampoline is security-sensitive because it can copy data across per-cubicle stacks; therefore, it must be generated and signed by the trusted builder.

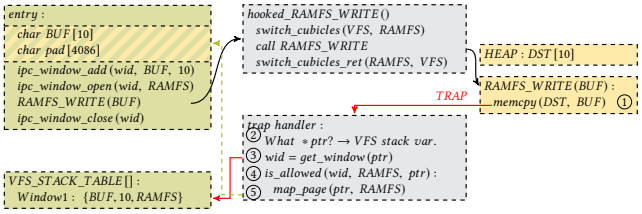


Figure 4: Trap-and-map scheme

Note that an application does not differ from other components in Unikraft: it has a public `main` function and uses functions from other components in the system.

5.3 Monitor: Memory Access Authorisation

The monitor in CubicleOS is responsible for bootstrapping the system and enforcing the isolation of cubicles and the access permissions of windows. Window manipulation is performance critical, because it can occur before and after each cross-cubicle call. We thus choose to operate windows as user-managed, discretionary access control lists (ACLs) for memory, and to lazily grant access to pages across cubicles.

Each cubicle has three arrays with **window descriptors** for global, stack, and heap data. The descriptors are stored in a simple array, created by the CubicleOS monitor, and contain the address and size of a single memory range and a bitmask of cubicles for which this window is open. The size of each bitmask is fixed at deployment time, as the number of cubicles in the system is known by CubicleOS' builder at link time. Furthermore, if a window descriptor array runs out of free entries, the user code asks the monitor to extend it.

CubicleOS authorises memory accesses across cubicles via a lazy *trap-and-map* approach. We assign one MPK tag for each cubicle, and change the tag of window-assigned pages as they are used. Figure 4 shows how CubicleOS' trap-and-map approach works with the example in Figures 2 and 3. The VFS isolated cubicle opens a window to `BUF` for RAMFS, and the `memcpy` function in the LIBC shared cubicle executes with the permissions of the RAMFS isolated cubicle. Note that the CubicleOS monitor is a trusted cubicle that executes with access to all cubicles (MPK tags) on the system, granting it access to all the per-cubicle window descriptor arrays:

❶ When `memcpy` tries to access `BUF`, that page is not accessible and a page fault exception is raised, which is captured by the monitor.

❷ The monitor then locates the window descriptor array for `BUF`'s page, which is owned by VFS. CubicleOS keeps a page metadata map that identifies the window descriptor array corresponding to that page, together with its owner and type (code, global data, stack or heap). Code and global data pages are known at deployment time, and CubicleOS builds a page metadata map at deployment time where pages can be located with O(1) time. CubicleOS updates a separate memory map for stack and heap at runtime, where pages can also be located with O(1) time. Pages are strictly assigned an owner and type at allocation time to ensure the safety of this operation, similar to L4Sec [28].

❸ The monitor then does a linear search in the window descriptor array for `BUF`'s page, which corresponds to *"Window 1"* in VFS. Since cross-cubicle calls usually grant access to few memory ranges, this is sufficient – in our evaluation, all but one cubicle have less than ten windows at any point in time.

❹ Finally, the monitor indexes the cubicle bitmask of the window using the cubicle ID of the failing instruction, RAMFS. This is a O(1) operation because all cubicles and their IDs are known at link time.

❺ After establishing that *"Window 1"* in VFS contains `BUF` and that the window is open for RAMFS, the monitor assigns the MPK tag for cubicle RAMFS to the page, granting the code access to it.

Note that windows work at page granularity, and a component developer must thus align structures to prevent unintended sharing via windows (see Section 6 for a description of developer effort).

5.4 Loader: Verifiable Isolation

The loader enforces two properties on untrusted code that ensure the integrity of the memory isolation mechanisms:

(1) No access to system calls, as they can change the MPK tags and access permissions on pages via the host OS.
(2) No access to the `wrpkru` instruction, as it can be used to change access permissions to the per-cubicle MPK tags.

Cubicles can only load new code via the loader, which sets them on a new cubicle. To enforce these integrity properties, the loader scans code pages for binary sequences containing system call or `wrpkru` instructions before making the pages executable, and refuses to load code if any such sequence is found [21].

When loading a binary, the loader also populates the per-cubicle page metadata maps. Finally, the loader performs dynamic symbol resolution such that cross-cubicle calls will go through the appropriate trampolines (described next), and allocates the necessary per-cubicle stacks for the current thread.

5.5 Cross-Cubicle Call Trampolines: CFI

CubicleOS uses the cross-cubicle call trampolines (generated by the trusted CubicleOS builder; see Section 5.2) to enforce control flow integrity (CFI), i.e., calls and returns across cubicles can only happen through the intended entry points. Note that trusted CubicleOS primitives such as core memory allocation are implemented on trusted cubicles and therefore are also accessed through cross-cubicle calls.

The code thunk of a trampoline is generated and signed by CubicleOS builder and thus trusted by the loader. It is in charge of performing the actual permission and context switch between cubicles. It uses `wrpkru` to switch access permissions between MPK tags assigned to the caller's and callee's cubicles (Figure 4), and performs a context switch between the per-cubicle stacks by copying necessary data across them (the function's binary interface is known by CubicleOS' builder when generating the trampoline's code thunk).

Hardware support. To ensure CFI in cross-cubicle calls, CubicleOS must ensure a caller can only enter a trampoline, and a callee resume execution of the trampoline, via the intended addresses. To that end, CubicleOS forbids direct execution of trampoline code

thunks and, instead, provides access to an intermediate caller or callee trampoline guard page that enforces control flow integrity.

This requires a simple change in the MPK implementation: whenever read and write access is disabled, execution is too, which is allowed otherwise. The loader places trampoline code thunks in the monitor's cubicle and places the guard pages on the corresponding caller and callee cubicles for that trampoline. Each guard page simply contains a `wrpkru` instruction to allow execution of the trampoline in the monitor's cubicle, and a jump to that trampoline, followed by a series of no-ops so that starting execution of a guard page anywhere but in its first instruction will result in a fault.

5.6 Discussion

The trap-and-map approach implemented in CubicleOS has several important features.

Trap-and-map model. Previous work [21, 54] on compartmentalisation has used MPK to share pages that are used for communication across compartments. This implies that each compartment needs one additional tag for every other compartment that it communicates with, and must copy data to/from these shared buffers. Such an approach requires changes to component interfaces to perform these memory copies, but also adds overhead due to the extra copies. It also requires a larger number of used MPK tags, which must be virtualised when exhausted [43]. Instead, CubicleOS's trap-and-map model maintains existing interfaces and avoids memory copies by dynamically changing a page's tag only when necessary, which also uses fewer tags.

Causal tag consistency. Closing a window does not immediately revoke access to its contents, i.e., page tags are not reassigned to the window's owning cubicle. Instead, CubicleOS keeps pages with their current tag, and lazily reassigns tags using trap-and-map only when a page is accessed by a cubicle with access to that window (which includes the owning cubicle). This is an optimisation to decrease the overhead of cubicle switches, and maintains causal consistency with respect to window operations. Since a callee cubicle could have accessed a page from a window before closing, it is thus correct to let it access it anytime before another cubicle accesses such a page.

Nested calls. A cubicle cannot open a new window on data shared by another cubicle (via a window) because a cubicle must be the owner of a window to modify its permissions through its window table. In the case of a nested call, i.e., when data is shared with more than one other cubicle, the window must be opened by its owner for all cubicles ahead of time. Alternatively, nested calls could copy shared data and open windows for intermediate buffers, or the monitor could be modified to keep track of "sub-windows", but we have not encountered such a case for the evaluated applications.

6 EVALUATION

We ask three questions in the evaluation of CubicleOS: (1) What is the overhead of CubicleOS for I/O-intensive workloads? (2) What is the overhead of CubicleOS for CPU/memory-intensive workloads? (3) How does its performance compare to other component-based OS designs?

To answer these questions, we port two applications to CubicleOS, the NGINX web server [41] and the SQLite embedded database engine [50]. We compare the performance of CubicleOS with Genode [17], which is a framework for the development of component-based applications that supports various OS kernels, including Linux, SeL4 [46], Fiasco.OC [16], and NOVA [42].

Table 2: Sizes of CubicleOS components

Component	SLOC	Language	Description
Monitor	110	ASM	Cross-cubicle calls
Monitor	3,000	C	All components
Builder	640	Python	Trampoline generation
Unikraft	600	C	Windows
SQLite	620	C	Windows
NGINX	390	C	Windows

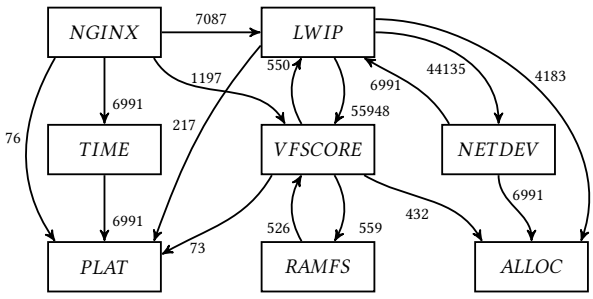


Figure 5: NGINX with cubicles (Call counts obtained during benchmark measurement time.)

6.1 Experimental Set-up

We use a machine with an Intel Xeon Silver 4210 CPU (2.20 GHz) and 32 GB of DDR4 memory. The software environment is based on Debian Linux 4.19.98-1 with kernel 4.19.0, Genode version 20.05, SQLite 3.30.1, NGINX 1.15.6 and Unikraft 0.4.0. The versions of Fiasco.OC, seL4, and NOVA are shipped with the Genode framework. We use LLVM 9.0 to generate LLVM IR and GCC 8.3.0 as a compiler. QEMU 3.1 (with KVM enabled) is used for the microkernel-based evaluation.

6.2 Developer Effort

Table 1 shows the source code size of CubicleOS. The core runtime components of CubicleOS consist of 3,000 lines of C code and 110 of assembly. The former implements cross-cubicle calls, the loader, and the builder, while the latter is a part of cross-cubicle calls. The builder parses the LLVM IR of Unikraft binaries and generates the trampolines, and is written in roughly 640 lines of Python. The window support added to Unikraft is around 600 lines. NGINX and SQLite require porting to CubicleOS, with an effort of 390 and 620 SLOCs, respectively.

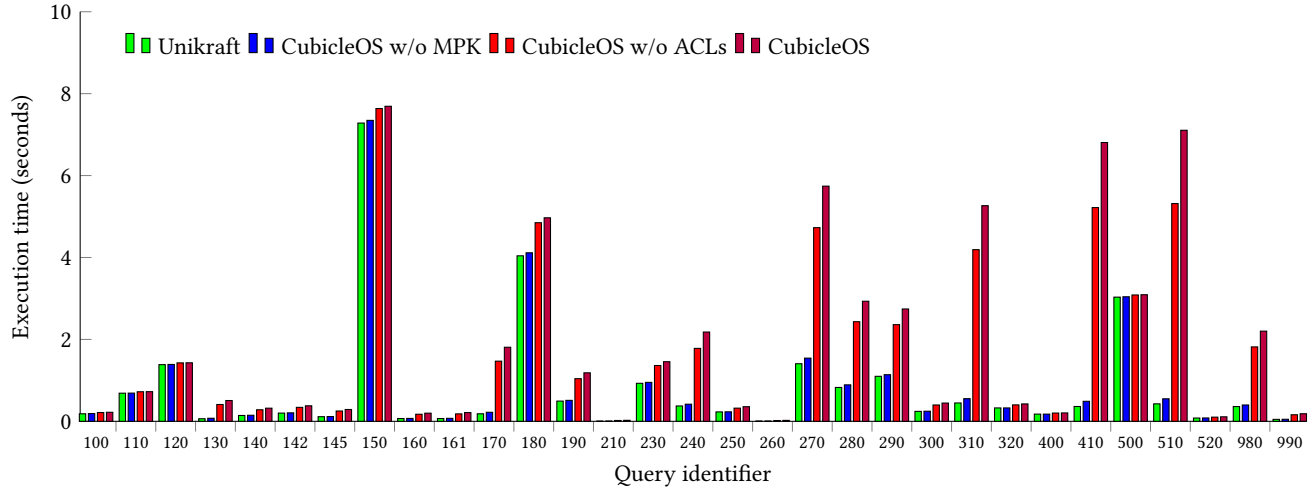


Figure 6: Query execution times for SQLite under CubicleOS (local execution)

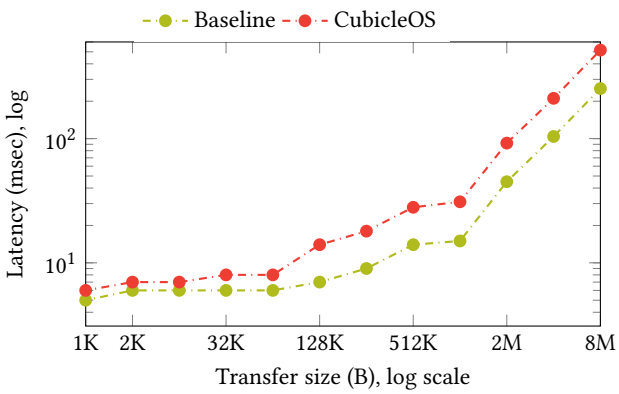


Figure 7: NGINX download latencies for different file sizes

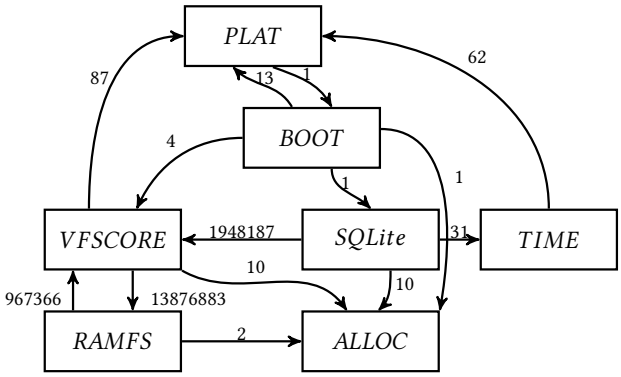


Figure 8: SQLite with cubicles. (Call counts include boot time.)

6.3 I/O-Intensive Workload (NGINX)

To benchmark NGINGX, we use the *siege* utility [48] to generate requests to random static files located inside the RAMFS of the server. As a performance metric, we measure the download latency of files with different sizes. As a baseline, we compare against the baseline Unikraft system.

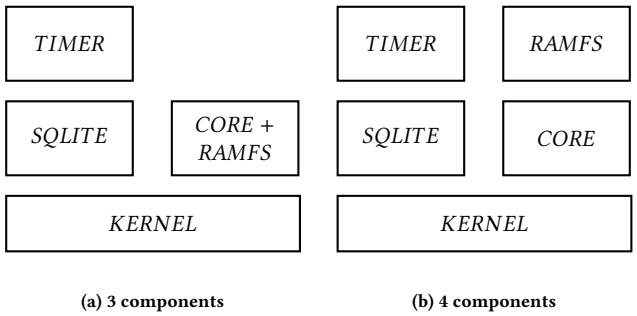
Figure 5 shows the cubicles used as part of NGINX, and the number of cross-cubicle calls during the execution. The main isolated cubicles are the application (`NGINX`), the TCP/IP stack (`LWIP`), the network device driver (`NETDEV`), the virtual file system (`VFSCORE`), the file system backend (`RAMFS`), and the platform code (`PLAT`). In addition, there is a system-wide memory allocator (`ALLOC`) and the time module (`TIME`). Shared cubicles are not shown, but are comprised of *newlibc* and the random device driver.

Figure 7 shows the request latencies for different file sizes. Latency is almost constant for small files (5–6 ms for the baseline; 6–7 ms for CubicleOS), and grows when the file size reaches 64 kB. After that, it grows almost linearly with file size. The change in slope for files larger than 1 MB is due to the buffer size inside `LWIP`. The overhead of cubicles also changes after 64 kB: it increases from 15% to 2×. In other words, the partitioning of NGINX into eight components that exchange a high volume of data halves the throughput.

6.4 CPU/Memory-Intensive Workload (SQLite)

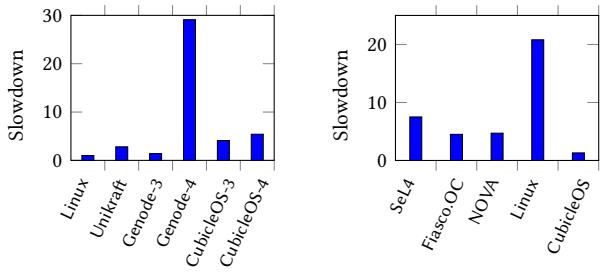
We port SQLite [50] to CubicleOS and observe performance under the *speedtest1* [49] benchmark workload. This benchmark includes various queries, starting from simple `INSERT` queries and ending with complex multi-way `JOINs`.

Figure 8 shows the configuration of cubicles. We use 7 isolated cubicles and 4 shared ones (not shown). Three isolated cubicles provide the main functionality and are thus invoked frequently: the `SQLite` library combined with the *speedtest1* benchmark, the virtual file system `VFSCORE`, and the driver for the `RAMFS` file system. Compared to NGINX, each cubicle uses only its own memory allocation library, and `ALLOC` is used only for coarse-grained allocations.



(a) 3 components (b) 4 components

Figure 9: Different ways of partitioning SQLite



(a) Compared to Linux (b) Compared to 3 components

Figure 10: CubicleOS overhead compared to different kernels

In this experiment, we want to compare the impact of the CubicleOS mechanisms on performance. We explore CubicleOS's three main mechanisms: (i) the switching mechanism between cubicles; (ii) the MPK-based memory protection; and (iii) the window-based cross-cubicle calls. We measure the impact of each mechanism on performance. Therefore, we compare: (a) the baseline Unikraft; (b) CubicleOS without MPK protection; (c) CubicleOS with MPK protection but without ACLs (i.e., the windows are "open" for any access); and (d) full-fledged CubicleOS.

Figure 6 shows the results for query execution times. The queries can be separated into two groups: almost two thirds of queries (100–120, 140–161, 180, 190, 230, 250, 300, 320, 400, 500, 520, 990) demonstrate a low moderate impact of cubicles on performance. Here, the introduction of the trampolines adds 2% overhead, MPK adds 50%, while the windows add 20% overhead. On average, CubicleOS-based SQLite requires 1.8× longer to process a query. A common feature of this group of queries is that they use the OS interface infrequently. They benefit from caching and only involve the OS interface to write batched pages evicted from the cache.

The second group of queries has significant overhead. While the trampolines add a reasonable overhead (17%), the use of MPK and windows increases request time by 4× and 1.2×, respectively. For these queries, on average, CubicleOS-based SQLite requires 8× more time to process queries. These queries benefit less from the use of the database page cache, and they significantly more often use the OS interface. The overhead of cubicle switches therefore become more significant, and the average slowdown factor is 4×.

6.5 Impact of Partitioning on Performance

We use the *speedtest1* benchmark for SQLite to compare the partitioning overhead in CubicleOS with that of other component-based approaches, the Genode framework and different microkernels. We create identical compartment configurations with CubicleOS and Genode, and measure the overhead caused by adding one extra compartment that separates the file system driver from the virtual file system. This operation is supported in both CubicleOS and Genode, thus comparing different kernels and interface implementations.

Figure 9 shows the deployments of SQLite used in this experiment. We first use the monolithic virtual filesystem module with the builtin `RAMFS` driver (Figure 9a), and compare with a partitioning in which the `RAMFS` driver is derived from the virtual file system. In the Genode terms, the virtual file system is the *Core* module; in the terms of CubicleOS, the virtual file system is a module that combines the `PLAT`, `VFSCORE`, `ALLOC`, and `BOOT` cubicles.

First, we compare Unikraft, Genode and CubicleOS. We configure all systems to run on top of the same Linux kernel, thus comparing the performance of user-level Unikraft with that of Linux/Genode interfaces and of cubicles. For the baseline, we use the same SQLite benchmark executed on top of the Linux kernel (Linux).

Figure 10a shows the average slowdown factor across all *speedtest1* queries against Linux. Unikraft is 2.8× slower. Genode with three modules (Genode-3) achieves better performance than Unikraft (1.4× slowdown). CubicleOS with three cubicles (CubicleOS-3) is 4.1× slower than Linux but, compared to Unikraft, the slowdown is similar to Genode (1.4×). The separation of `RAMFS` leads to a 29× performance drop for Genode (Genode-4); CubicleOS only decreases performance by 5.4× (CubicleOS-4), which is more than five times better. Compared to Unikraft, CubicleOS-4 is 1.9× slower.

Next, we also measure the overhead of adding an extra compartment in modern microkernels. We use the same platform, framework (Genode) and benchmark, but different kernels. Thus comparing the performance of message-based interfaces with that of CubicleOS. Here we use SeL4 [46], Fiasco.OC [16], NOVA [42]. We measure the performance of the systems with three and four compartments and, as the baseline, we use a partitioning with three components.

Figure 10b shows the slowdown compared to 3 components for the different kernels. Compared to Linux, the separation of `RAMFS` has less performance impact for microkernels, which is not surprising. The average slowdown for SeL4 is 7.5×, while for Fiasco.OC and NOVA, it is 4.5× and 4.7×, respectively. However, the performance overhead of the separation for CubicleOS is only 1.4×, showing how CubicleOS becomes more efficient with multiple partitions.

In summary, our evaluation has explored if it is possible to turn a monolithic library OS design into a compartmentalised one without the use of message-based interface. Our results show that CubicleOS's window-based mechanism allows the use of ordinary call semantics, while enabling fast switches that decrease the overhead of transitions, which are caused by the inefficiency of the library OS design. Moreover, the results indicate that such an approach can have lower overhead than traditional component-based microkernel frameworks. Adding an extra compartment to the critical

path of SQLite (RAMFS) leads to a 4–7× performance drop for microkernels, and only 1.4× for CubicleOS; NGINX with 8 partitions is 2× slower than its monolithic version.

7 RELATED WORK

Partitioning with Intel MPK. ERIM [54] uses MPK to separate trusted and untrusted code. The call gates introduced by ERIM, however, do not include support to pass parameters and results between protection domains. CubicleOS not only generalises this to support multiple untrusted components, but it also supports dynamic passing of arguments with zero-copy.

Similarly, Hodor [21] introduces a *protected library* abstraction. Such libraries are mutually isolated from each other using MPK or VMFUNC, and use trusted function trampolines to switch between libraries. Compared to CubicleOS, Hodor cannot compartmentalize OS components, requires substantial developer efforts to explicitly manage sharing of selected data, especially for stack arguments, and fine-grained sharing results in using a large number of MPK tags (which are limited).

UnderBridge [19] uses MPK to migrate and isolate critical system services in kernel mode, focusing on speeding up the message-passing IPC typical of micro-kernels, whereas CubicleOS works at user-level and explores how to isolate existing code in a monolithic system architecture.

Sung et al. [52] apply MPK to RustyHermit [32] to isolate the unikernel and its application. They do not compartmentalise unikernel components, as the work assumes that using the type-safe Rust language is sufficient. Iso-UniK [33] also uses MPK-based isolation to implement processes in OSv [29] without partitioning.

EnclaveDom [40] enforces privilege isolation of some LibOS data structures by associating them with separate MPK tags. Only relevant system calls implemented by the LibOS have access to the protected data structures, which is achieved by manually labelling data objects. CubicleOS, in contrast, isolates both data and code and requires few changes to the target system.

OS kernel compartmentalisation. Monolithic kernels such as Linux have long been considered as a target for compartmentalisation. PerspicuOS [13] splits the monolithic kernel into a smaller trusted part and a larger untrusted one. LXFI [38] and Mondrix [58] enable the isolation of individual Linux kernel modules. CubicleOS compartmentalises user-level library OSs with little developer involvement, whereas LXFI relies on extensive source code annotations and compiler support, and Mondrix requires extensive hardware modifications (MMP). CubicleOS and Mondrix both create protection domains within a single shared address space, but Mondrix' hardware support significantly impacts its design and performance.

Compartmentalisation frameworks. Breakapp [55], GOTEE [18], Secured Routines [18] and Civet [8] compartmentalize user applications using languages with well-defined typing rules, whereas CubicleOS covers the entire system and supports type-unsafe code.

Glamdring [35] automates code partitioning for Intel SGX by annotating C source code files. Developers mark the data that must be protected, and the framework identifies necessary components that must be placed inside an SGX enclaves together with the data. CubicleOS offers more fine-grained partitioning because it is based on Intel MPK, which is however incompatible with Intel SGX.

*E*Actors [45] partitions programs by assigning execution units to different SGX enclaves. This done at compile time, requires low effort by developers but enforces an actor-based programming model. CubicleOS targets regular software components.

SOAAP [20] is an LLVM-based system for developers to make compartmentalisation decisions based on source-code annotations. It could be combined with CubicleOS to split modules and increase the partitioning granularity.

ConfLLVM [4] is another LLVM-based framework that separates trusted and untrusted code. It only supports two partitions and uses Intel MPK for isolation. Compared to CubicleOS, ConfLLVM adds a higher instrumentation overhead and enables only uni-directional data exchange between compartments: trusted code can access untrusted one, but not vice-versa. PrivTrans [6] is a source-level partitioning tool based on source annotations. It splits source code into two partitions, and the calling interface between them is based data marshalling. Wedge [3] creates isolation primitives inside processes. These "sthreads" inherit a subset of the parent's memory mappings and have only limited access to the kernel namespace. Wedge does not marshal arguments but disallows one partition to access another one. CubicleOS does not use marshalling in calls, provides multiple partitions, and allows and protects cross-cubicle memory accesses.

Hardware extensions. CHERI [59] introduces hardware-software object capabilities. They can be used for compartmentalisation within the process address space. dIPC [57] uses a hardware-enabled IPC mechanism that also can be applied to isolate partitions. In contrast, CubicleOS does not require extra support from the OS and compiler (unlike CHERI), and uses commodity CPU hardware (with a minor extension).

8 DISCUSSION

CubicleOS is built on top of a monolithic unikernel framework that was not designed for isolation, but it achieves efficient compartmentalisation with low developer effort. Benefits notwithstanding, CubicleOS has certain technical limitations discussed below.

Using a trusted builder tool in CubicleOS is at odds with DevOps tools that dynamically stack binary components into final system images. Instead, CubicleOS could exploit existing debug information in binaries to generate the trusted cross-cubicle call trampolines at deployment time [57], instead of using a trusted builder.

CubicleOS also has to confront several hardware limitations that directly impact its performance. First, the number of MPK tags is limited to 16 by hardware. CubicleOS can efficiently accomodate more compartments because it uses fewer MPK tags by design (one per compartment, as opposed to one per compartment and per communication buffer shared with other compartments). We note that our evaluation experiments have not needed more than the 16 tags provided by the hardware, but if more tags were required, CubicleOS could use existing tag virtualisation mechanism [43]. Here, it would be interesting to explore new designs that combine CubicleOS's trap-and-map approach with window-specific tags that reduces overhead for frequently-used windows.

Second, MPK does not control access to `wrpkru` instructions. CubicleOS therefore uses load-time binary analysis to detect unauthorised instructions, which is sufficient but not desirable. This

could be solved by simple hardware changes such as relying on per-page instruction permissions (as in CODOMs [56]) to authorise `wrpkru` only for trusted pages.

Third, MPK lacks tag-wide no-execute permissions, which can defeat CubicleOS's CFI. We propose a simple hardware modification to provide efficient tag-wide execution permissions, which significantly decreases the complexity of the system compared to a software-only solution.

Fourth, most architectures lack general hardware support for CFI. Together with the proposed MPK hardware modifications, CubicleOS's cross-cubicle call trampolines provide a software solution to CFI. Nevertheless, planned hardware modifications such as Intel CTA [26] would make this task even more efficient in CubicleOS.

The burden of managing windows and (in some cases) segregating allocations onto separate pages now falls entirely onto the developer. Nevertheless, others have shown how to use the compiler to identify memory regions used across isolation domains, only in some cases requiring developer annotations [10].

Unikraft, on which CubicleOS is based, provides a model in which user-level threads are multiplexed onto a single host thread. Previous work has argued that application parallelism can be achieved via multiple library OS instances instead of multiple threads [36]. In the case in which host multi-threading is necessary in an application, we speculate that multi-threading, together with MPK tag virtualisation, can reduce the time spent in CubicleOS's trap-and-map handler.

9 CONCLUSIONS

Many library OSs and unikernels offer POSIX-compatible, lightweight environments for application deployment inside containers and trusted execution environments. While monolithic library OS designs can achieve high performance without excessive switches, they lack any isolation across components, thus ignoring decades of best security practices that have led to compartmentalised systems.

We explore how to address this challenge through CubicleOS, a compartmentalised library OS that proposes *cubicles*, *windows* and *cross-cubicle calls* as new abstractions for practical compartmentalisation of a library OS. We show that these abstractions provide low-overhead fine-grained isolation and zero-copy data access across partitioned components, while preserving rich OS functionality.

ACKNOWLEDGEMENTS

We thank the reviewers and our shepherd, Yannis Smaragdakis, for their valuable feedback. This work was partially funded by the UK Government's Industrial Strategy Challenge Fund (ISCF) under the Digital Security by Design (DSbD) Programme, to deliver a DSbDtech enabled digital platform.

A ARTEFACT APPENDIX

A.1 Abstract

This artefact contains the library OS, two applications, the isolation monitor, and scripts to reproduce the experiments from the ASPLOS 2021 paper by V. A. Sartakov, L. Vilanova, R. Pietzuch — "CubicleOS: A Library OS with Software Componentisation for Practical Isolation", which isolates components of a monolithic library OS without the use of message-based IPC primitives.

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

- Algorithm: Component isolation
- Program: Unikraft, NGINX, SQLite
- Compilation: gcc 8.3, Debian 10, Linux kernel 4.19, Python 3.7
- Transformations: Software isolates components of a system in runtime by the use of Intel MPK
- Data set: SQLite Speedtest1 benchmark
- Binaries: Compiled from the source code
- Hardware: Intel MPK, Intel VT-x, we recommend Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz
- Metrics: Slowdown, latency
- Output: Performance-related metadata
- How much disk space required (approximately)?: 20 GiB
- How much time is needed to prepare workflow?: approx. 30 mins
- How much time is needed to complete experiments?: approx. 20 mins
- Publicly available?: Yes
- Workflow framework used?: Docker, Bash scripts
- Archived (provide DOI)?: 10.5281/zenodo.4321431

A.3 Description

A.3.1 How to Access. All components of CubicleOS, applications, benchmarks, and deployment scripts are available on GitHub: http://github.com/lsds/CubicleOS, in the `ASPLOS_AE` branch.

A.3.2 Hardware Dependencies. CubicleOS requires Intel MPK. To run a baseline workload (Microkernels Fiasco.OC, NOVA, SeL4), one needs to use a platform with VT-x support, given that the workload is shipped in the form of disk images and QEMU-KVM scripts.

A.3.3 Software Dependencies. Debian 10 with the Linux kernel version 4.19. The runtime of CubicleOS parses `/proc/self/maps` and the format of this file may vary between different versions of the kernel and/or OS.

A.3.4 Data Sets. All benchmarks are included in the source code. We use a simple curl-based script for NGINX, and Speedtest1 for SQLite.

A.4 Installation

First, you should check that your platform supports Intel MPK:

```
$ gcc check.c && ./a.out
pkey alloc = 1
pkey: Success
```

Then, you can build CubicleOS and all relevant tests:

```
docker build . --tag cubicles
```

The Docker script retrieves any necessary dependencies and creates a container that includes (i) CubicleOS and its components including two applications and (ii) a Genode-based build environment to generate and run baseline tests for microkernels. To run CubicleOS, the platform should have Intel MPK, to run microkernel-based systems (SeL4, Fiasco.OC, NOVA), the platform should have VT-x (or nested virtualisation enabled in the case of a cloud environment).

A.5 Experiment Workflow

To build a container and run the tests and benchmarks, one needs to execute:

```
./run.sh
```

This script includes both the NGINX and SQLite tests. To run a specific test, uncomment the corresponding line in this file, or follow the instructions in `README.md`. By default, the script only builds the container.

A.6 Evaluation and Expected Result

Each test generates raw output that requires post-processing. The output of each Speedtest1 benchmark should be manually saved in a file and processed by the `parser/parser.py` utility. This utility takes as input a directory with raw data, where each test result is stored as a separate file in accordance with the naming convention. It then processes the data and generates the results. Two directories, `yandex` and `paper`, are provided as references and contain raw results for different hardware platforms.

A.7 Experiment Customisation

The existing benchmarks can be customised. For SQLite, the size of the database can be changed via the `--stat XXX` flag (100 is the default). For NGINX, the root filesystem with files can be changed.

Note that some changes may require reconfiguration of CubicleOS, e.g. changing the HEAP sizes of some components or adding new Windows. The execution of other programs requires porting them.

A.8 Notes

CubicleOS significantly benefits from the size of CPU caches. It is expected that public cloud environments likely show results that differ from the ones reported in the paper. However, the main feature of CubicleOS – the ability to compartmentalise a LibOS with less overhead than microkernels – are reproducible on any platform.

The microkernel-based benchmarks also significantly differ from one platform to another, but the measured performance degradation caused by compartmentalisation of the RAMFS (Fig. 10b), for microkernels, is always more than 4×; for CubicleOS, this value is significantly smaller (1.3×).

A.9 Methodology

Submission, reviewing and badging methodology:

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

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Trans. Inf. Syst. Secur.*, 13(1), November 2009.

[2] ARM. ARM Security Technology: Building a Secure System using TrustZone Technology. *White paper*, 2009. 2020.

[3] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *5th USENIX Symposium on Networked Systems Design and Implementation (USENIX NSDI 08)*. USENIX Association, April 2008.

[4] Ajay Brahmakshatriya, Piyus Kedia, Derrick P. McKee, Deepak Garg, Akash Lal, Aseem Rastogi, Hamed Nemati, Anmol Panda, and Pratik Bhatu. ConfLLVM: A Compiler for Enforcing Data Confidentiality in Low-Level Code. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, 2019.

[5] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E Engelstad, and Kyrre Begnum. IncludeOS: A minimal, resource efficient unikernel for cloud services. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 250–257. IEEE, 2015.

[6] David Brumley and Dawn Song. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *13th USENIX Security Symposium (USENIX Security 04)*. USENIX Association, August 2004.

[7] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658. USENIX Association, July 2017.

[8] Chia che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca Ada Popa, and Donald E. Porter. Civet: An Efficient Java Partitioning Framework for Hardware Enclaves. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 505–522. USENIX Association, August 2020.

[9] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. StackArmor: Comprehensive Protection From Stack-based Memory Error Vulnerabilities for Binaries. In *NDSS*. Citeseer, 2015.

[10] David Chisnall, Colin Rothwell, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. Beyond the PDP-11: Architectural support for a memory-safe C abstract machine. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2015.

[11] Intel Corp. Software Guard Extensions Programming Reference, Ref. 329298-002US. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, October 2014.

[12] CVE-2018-5410. Available from MITRE, CVE-ID CVE-2018-5410.

[13] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. *SIGPLAN Not.*, 50(4):191–206, March 2015.

[14] Richard P Draves, Michael B Jones, and Mary R Thompson. *MIG-The MACH Interface Generator*. School of Computer Science, Carnegie Mellon University, 1989.

[15] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. *SIGPLAN Not.*, pages 693–707, March 2018.

[16] The Fiasco.OC Microkernel Repository. https://github.com/kernkonzept/fiasco. Last accessed: Feb 15, 2021.

[17] Genode Operating System Framework. https://github.com/genodelabs/genode. Last accessed: Feb 15, 2021.

[18] Adrien Ghosn, James R. Larus, and Edouard Bugnion. Secured Routines: Language-based Construction of Trusted Execution Environments. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 571–586. USENIX Association, July 2019.

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

[20] Khilan Gudka, Robert N.M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. Clean Application Compartmentalization with SOAAP. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1016–1031. Association for Computing Machinery, 2015.

[21] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 489–504. USENIX Association, July 2019.

[22] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Minix 3: A highly reliable, self-repairing operating system. *SIGOPS Oper. Syst. Rev.*, 40(3):80–89, July 2006.

[23] Hewlett Packard. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, February 1994.

[24] Intel. *Intel Itanium Architecture Software Developer's Manual*, January 2006.

[25] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual. 2018.

[26] Intel. Control-flow Enforcement Technology Specification. *White paper*, 2019. 2020.

[27] David Kaplan, Jeremy Powell, and Tom Woller. AMD memory encryption. *White paper*, 2016.

[28] Bernhard Kauer and Marcus Völp. L4. sec preliminary microkernel reference manual. 2005.

[29] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. OSv—Optimizing the Operating System for Virtual Machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 61–72. USENIX Association, June 2014.

[30] 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. SeL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 207–220. Association for Computing Machinery, 2009.

[31] Simon Kuenzer, Sharan Santhanam, Yuri Volchkov, Florian Schmidt, Felipe Huici, Joel Nider, Mike Rapoport, and Costin Lupu. Unleashing the Power of Unikernels with Unikraft. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, page 195. Association for Computing Machinery, 2019.

[32] Stefan Lankes, Jens Breitbart, and Simon Pickartz. Exploring Rust for Unikernel Development. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*, pages 8–15. Association for Computing Machinery, 2019.

[33] Guanyu Li, Dong Du, and Yubin Xia. Iso-UniK: lightweight multi-process unikernel through memory protection keys. *Cybersecurity*, 3:1–14, 2020.

[34] Jochen Liedtke. Improving IPC by Kernel Design. 27(5):175–188, 1993.

[35] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. Glamdring: Automatic application partitioning for intel SGX.

In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 285–298. USENIX Association, 2017.

[36] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian Leslie. Jitsu: Just-In-Time Summoning of Unikernels. In *USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, May 2015.

[37] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. *ACM SIGARCH Computer Architecture News*, 41(1):461–472, 2013.

[38] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Software Fault Isolation with API Integrity and Multi-Principal Modules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 115–128. Association for Computing Machinery, 2011.

[39] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. ClickOS and the Art of Network Function Virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (USENIX NSDI 14)*, pages 459–473. USENIX Association, April 2014.

[40] Marcela S Melara, Michael J Freedman, and Mic Bowman. EnclaveDom: Privilege separation for large-TCB applications in trusted execution environments. *arXiv preprint arXiv:1907.13245*, 2019.

[41] NGINX, an HTTP server. https://www.nginx.org. 2019.

[42] NOVA Microhypervisor. https://github.com/udosteinberg/NOVA/. Last accessed: Feb 15, 2021.

[43] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 241–254. USENIX Association, July 2019.

[44] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A Sartakov, and Peter Pietzuch. SGX-LKL: Securing the host OS interface for trusted execution. *arXiv preprint arXiv:1908.11143*, 2019.

[45] Vasily A Sartakov, Stefan Brenner, Sonia Ben Mokhtar, Sara Bouchenak, Gaël Thomas, and Rüdiger Kapitza. Eactors: Fast and flexible trusted computing using sgx. In *Proceedings of the 19th International Middleware Conference*, pages 187–200, 2018.

[46] The seL4 microkernel. https://github.com/seL4/seL4. Last accessed: Feb 15, 2021.

[47] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 121–135, 2019.

[48] Siege. https://www.joedog.org/siege-home/. 2020.

[49] Speedtest1 benchmark. http://www.sqlite.org/src/finfo?name=test/speedtest1.c. Last accessed: Feb 15, 2021.

[50] SQLite. https://www.sqlite.org. 2020.

[51] Udo Steinberg and Bernhard Kauer. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 209–222. Association for Computing Machinery, 2010.

[52] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. Intra-Unikernel Isolation with Intel Memory Protection Keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, pages 143–156, 2020.

[53] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E Porter. Cooperation and security isolation of library OSes for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[54] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1221–1238. USENIX Association, August 2019.

[55] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M Smith. BreakApp: Automated, Flexible Application Compartmentalization. In *2018 Network and Distributed System Security Symposium (NDSS'18)*.

[56] Lluís Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. CODOMs: Protecting software with code-centric memory domains. *ACM SIGARCH Computer Architecture News*, 42(3):469–480, 2014.

[57] Lluís Vilanova, Marc Jordà, Nacho Navarro, Yoav Etsion, and Mateo Valero. Direct Inter-Process Communication (dIPC) Repurposing the CODOMs Architecture to Accelerate IPC. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 16–31, 2017.

[58] Emmett Witchel, Junghwan Rhee, and Krste Asanović. Mondrix: Memory Isolation for Linux Using Mondriaan Memory Protection. volume 39, pages 31–44, New York, NY, USA, October 2005. Association for Computing Machinery.

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