uIO: Lightweight and Extensible Unikernels

Paper #80

ABSTRACT

Unikernels specialize operating systems by tailoring the kernel for a specific application at compile time. While the specialized library OS approach provides a smaller OS image—thus improving the bootup process, performance, migration costs, and reliable/trusted computing base—at the same time, unikernels lack run-time extensibility, which is imperative to support "on-demand" auxiliary tasks and tools, *e.g.*, debugging, monitoring, re-configuration, and system management and deployment in a typical cloud environment. Consequently, unikernels present a fundamental trade-off between *slimness* of the OS image size at the compile time vs. *flexibility* of supported auxiliary functionality at the run-time.

This work strives to balance this trade-off by keeping the unikernel system image as minimal as possible to solely support the application functionality in the "common case", while providing "on-demand" extensibility for auxiliary tasks at run-time. The key challenge is to support run-time extensibility through a *generic* interface in a *safe* manner.

To this end, the paper presents uIO—a "*safe overlay*" abstraction to provide runtime extensibility in unikernels, while maintaining the unikernel benefits. In particular, uIO leverages a *generic* VirtIO-based interface to provide an overlay for auxiliary programs, i.e., users can load external programs into the unikernels' address space and run them, *i.e.*, "on-demand" extensibility through a generic file system interface. To provide *safe* execution within an overlay, uIO provides isolation mechanisms leveraging hardware-assisted memory isolation (MPK) and language-runtime-based execution (eBPF). We implement a prototype of uIO based on Unikraft and demonstrate its applicability to support a range of auxiliary use cases. uIO incurs *negligible* performance overheads for application execution in the common case while providing run-time extensibility to support auxiliary use cases.

1 INTRODUCTION

Context and motivation. Unikernels [79] are specialized operating systems for a specific application. By optimizing the kernel code along with an application at compile time, unikernels achieve high performance, small image size, fast boot time, minimalistic overheads for state migration, and low trusted and reliability computing base [64, 67, 79, 80]. Given these advantages, unikernels are gaining traction for a range of application domains, including cloud services [13, 66, 68, 77], storage and networked systems [17, 43, 70, 78, 81, 83, 112], HPC [51, 71–73], serverless computing [4, 15, 31, 104], and IoT and edge computing [18, 28, 44, 84, 118].

However, at the same time, unikernels are considered to be impractical because of their lack of run-time extensibility, which is one of the major problems for production cloud environments [14, 103]. In particular, most practical cloud deployments require support for spawning auxiliary tasks in an "on-demand" manner, *e.g.*, system monitoring, logging, configuration updates, system management, debugging, state backups [3, 21, 42, 98]. Unfortunately, most unikernels do not support multi-processing due to their single-address space nature, and it is challenging to use standard tools such as ssh [87] to connect unikernels and run commands as we do with traditional operating systems [117]. Given these limitations for production deployments, unikernels are considered primarily a research prototype despite their numerous advantages [1, 14, 89].

Limitations of state-of-the-art approaches. These limitations are deep-rooted in the unikernel philosophy of designing minimalistic system images, where the application developers strip down auxiliary tools, and tailor the underlying operating system at the compile time to support only the application functionality. A naive solution to this problem would be adding normal process abstraction to the unikernel and support shells. However, this would sacrifice the benefits of unikernels as this requires many mechanisms included in unikernels that are not commonly used, making OS specialization inefficient and increasing system images. Although a few works try to improve debuggability and observability in unikernels , these are specific to profiling [3, 21, 42, 98] or specific to one application [68] and do not provide a generic mechanism where users can load and execute program on-demand.

To resolve this unikernel conundrum, we ask the following research question: *Can we have a safe and generic mechanism to extend unikernels on demand without compromising on their advantages?*

Key insights and contributions. To address this question, we propose uIO, which realizes an *overlay* interface that enables extending unikernels at run time while keeping their lightweightness. uIO exposes overlays via a generic and minimalistic interface while providing external file systems and console access usable in unikernels. Thus, a user can use uIO to send and run commands in unikernels. This interface allows users to interact with unikernels in the same context while keeping its image size as small as possible. uIO loads extra components from the file system only when needed, i.e., *extensibility on-demand.*

To provide this extensible overlay interface, we must solve two key challenges: *generality* and *safety.*

For *generality*, unikernels do not have a uniform interface for extensibility. They are designed for a single application and most of them do not have a generic mechanism to spawn a new process. Therefore, a traditional fork-and-exec model to execute programs is not applicable. We address this challenge by designing a minimalistic yet generic interface with a standardized VirtIO file system protocol [86, 94]. Our interface adapts a load-and-call execution model that dynamically loads components and calls them within the same context; thus, our approach does not compromise the performance advantages of unikernels.

For *safety*, unikernels typically do not provide a safety mechanism since everything runs within a single address space. At the same, we do not want an overlay to compromise unikernels accidentally, yet we still want maximum programmability. To address the second challenge, we provide two lightweight safe isolation execution environments: (a) hardware-assisted lightweight memory protection mechanism (MPK) [6, 48, 99], and (b) language-runtime-based safety guarantees by limiting functionalities and verifying code in advance or by checking dynamically (eBPF) [54]. The user can choose one of those execution environments depending on their requirements.

Experimental methodology and artifact availability. We implement our prototype based on Unikraft [67] targeting x86_64 with QEMU/KVM [11, 63]. We define a minimal communication interface on top of VirtIO-console [114] and VirtIO-9p [115], implement loader and linker to execute external programs in the unikernel context, and leverage eBPF [54] and MPK [6, 48] to realize lightweight safe execution. We evaluate uIO with three real-world applications (Nginx [29], SQLite [100], and Redis [92]) in terms of robustness, performance, and effectiveness of uIO. Finally, we show that uIO enables enough extensibility to realize five real-world auxiliary use cases: (1) interactive debugging, (2) online Nginx re-configuration, (3) online SQLite backup, (4) performance monitoring with performance counters, and (5) Dynamic function inspection and tracing with eBPF. uIO will be publicly available.

Limitations of the proposed approach. uIO's safe execution environment does not prevent application logic bugs such as deadlock. In addition, we consider the scenario that a cloud provider provides uIO as a one of management service, meaning that the user trusts the provider. We could potentially deploy the uIO components and the unikernel in confidential VMs, as provided by AMD SEV [7] or Intel TDX [50], to decouple the trust assumptions from the cloud provider.

2 BACKGROUND AND MOTIVATION

We first examine the strengths and weaknesses of unikernels and then identify a gap that uIO aims to fill.

2.1 Trade-offs in the Unikernel World

Unikernels. Unikernels [79] are specialized operating systems designed to run a single application. They are optimized for the specific application at compile time. Unikernels [79] are specialized operating systems optimized for a specific application at compile time. They only include the necessary functionality to run the application, resulting in better performance, smaller image sizes, and faster boot-up times [78, 80]. Since unikernel targets a single application, it can eliminate protection features necessary in a typical operating system and only contain necessary functionality to run the application. This allows unikernels to perform better thanks to less context switches and protection overheads than applications on a typical OS.

Unikernels have several advantages compared to traditional operating systems. First of all, unikernels can be specialized for the target application at compile time. This improves performance and allows the unikernel only to include the code necessary for the application's operation, resulting in a smaller image size [64, 67, 97]. A smaller image size enables faster boot-up time [78, 80]. Unikernels can provide a more secure environment than containers because virtualization provides stronger isolation than processes [80]. Additionally, since only one process runs within the unikernel, there is no need for the process isolation mechanism that is required in traditional operating systems. As a result, the application and kernel can operate in the same single address space, reducing the context switch between user and kernel and improving performance more. Unlike regular applications and containers, unikernels typically run on a hypervisor as a virtual machine (VM), and only contain the code necessary to run a specific application. Unikernels rely on the host side (the hypervisor) for actual I/O processing, scheduling, and isolation among unikernels.

Trade-offs. The benefits of unikernels are related to the trade-off between functionality and lightweightness [78, 80]. In other words, typical operating systems support the execution of multiple processes and allow for the installation and execution of new programs as needed. By eliminating these features, unikernels can optimize to a specific application and reduce context switching and process isolation overheads, resulting in high performance and a smaller image size. In addition, the underlying hypervisor provides stronger isolation between unikernels than the container-based isolation [80].

However, this nature of unikernels can be problematic in production environments. For example, in typical cloud staging environments, if applications have problems, users can log in to the VMs over the network and debug the error logs, change configuration, or even run management tasks. Unfortunately, it is impossible to support such auxiliary functionality and management tasks and tools in unikernels.

A naive solution to this problem is to support multiprocessing and shell environments in unikernels like a typical OS. However, this would contradict the design principles of unikernels as it introduces performance and file system size overheads. To this end, we target the following: *How can we design a generic and safe interface to extend the functionality of unikernels while keeping their benefits?*

2.2 The Missing Overlay Abstraction

Because unikernels aim to run a single application, the traditional shell abstraction that starts processes with a fork is inappropriate. In fact, due to the nature of unikernels, a general-purpose process environment that can handle any workload is unnecessary. Instead, it is sufficient for a unikernel to handle processing related to the main application on demand.

We need an *overlay* abstraction that provides a common interface to connect unikernels and an on-demand command execution mechanism for the main application while keeping unikernels' advantages. uIO aims to realize this abstraction. Our two key observations are: (1) most unikernels already support (not multiprocessing but) multi-threading, which can be used to support auxiliary on-demand functionalities. (2) VirtIO [86, 94] is becoming the de-facto standard for virtualized environments, which can be leveraged to define a generic run-time communication interface for unikernels deployed in production. Using these two ingredients, we aim to design a generic and minimal interface to realize the overlay abstraction, while ensuring safety with lightweight isolation.

2.3 Example Overlay Use cases

With the *overlay* abstraction, we can enable a range of new services to help administrators and developers (also see § 6.4 that evaluates implemented use cases).

Debugging. Debugging unikernels can be challenging due to the lack of an interface to access their states. One common way to debug unikernels is to run them as processes [23, 53, 117]. Unfortunately, this approach is not usable for investigating problems in production where unikernels run in a virtualized environment. uIO can provide an interactive debugging overlay where users can examine the application state. For example, uIO can provides access interface to ramfs allowing users to check the application's internal log that only exists in the memory (§ 6.4 #1).

Interactive management environment. Many applications support dynamic reconfiguration, *e.g.*, Nginx [29] supports reloading configuration files at run time [30]. However, these configuration mechanisms are usually unavailable in unikernels. uIO can provide an interactive management overlay where users can reconfigure an application (§ 6.4 #2).

Running auxiliary tasks. Many applications have auxiliary tasks, such as performance monitoring, fault detection, and resource management [52]. For example, Redis [92] can periodically take a snapshot of its contents for backup and examination [93]. uIO can offer an overlay interface to invoke such tasks on demand (§ 6.4 #3).

Performance monitoring. Performance monitoring is essential for maintaining the health of applications and identifying bottlenecks. While it is possible to perform some monitoring from the host side [37], it is generally less efficient than monitoring within virtual machines (VMs) due to the semantic gap [16]. uIO can provide performance monitoring overlays, such as accessing performance counters (§ 6.4 #4).

3 OVERVIEW

In this section, we present an overview of the proposed system.

3.1 System Overview

To realize the *unikernel overlay* abstraction (§ 2), we design uIO, which provides a common interface to extend unikernels at run time. Figure 1 shows the overview of uIO. uIO has two main components. The first is the *host-side uIO process* (the orange part) that manages a file system (*uIO-fs*) and a console (*uIO-console*). uIO-fs serves storage for an overlay while uIO-console provides console access to the user. The other component is the in-unikernel *uIO context* running in a safe overlay. uIO context is schedulable, and shares the address space with the main application, and the overlay provides lightweight isolation to ensure safe execution without accidentally compromising the main application.

The uIO context in the unikernel works with the aid of the uIO process. First, users send commands to uIO context via uIO-console. Then the context handle requests. The context uses uIO-fs to load an external program and run it in the safe overlay.

To this end, uIO strives for the following design goals:

- *Generality*: uIO provides a generic interface that allows users to extend unikernels behavior at run time.
- *Lightweightness*: uIO aims to provide a minimalistic interface without compromising unikernels' lightweightness and performance.
- *Safety*: The uIO overlay provides safety with an isolation mechanism, i.e., not accidentally compromising the unikernel yet allowing programmability.

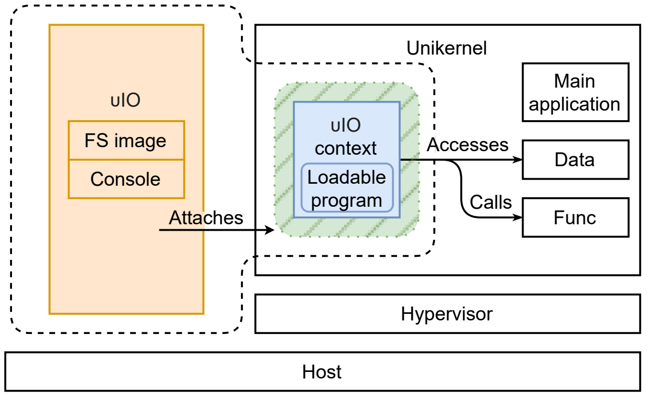

Figure 1: Overview of uIO. (*Orange components running on the host and blue components in the unikernel. Green hatched region denotes a safe overlay.*)

Figure 2 shows the detailed uIO workflow. In step ①, uIO attaches its console and file system to the unikernel. The communication layer in the unikernel handles this console and file system communication. In step ②, the user sends a command to the uIO context via the console device. ③ uIO context processes the command accordingly. If the command is to execute a new program (in this example, `hello` program), first, the uIO context loads the program via the file system, and then calls it (④). The program executes within the uIO context and is isolated from the main application.

3.2 Assumptions and System Model

For uIO, we consider a scenario where unikernels are deployed in a cloud environment. Each unikernel runs on a hypervisor, and the hypervisor provides isolation among them through hardware-assisted virtualization [63]. Hence, we trust the hypervisor and the operating systems on which unikernels and uIO run. In this scenario, cloud providers offer uIO as one of their management services. Therefore, uIO is owned and controlled by the cloud provider, acting on behalf of the customer upon their requests. We assume that the provider provides an authentication mechanism so that only registered users can use uIO. The provider accounts for the additional resource consumption caused by uIO as resources used by the virtual machine.

uIO uses a virtual console and file system on the host side. As these are common functionalities hypervisors provide, uIO does not significantly increase the TCB. In addition, the host employs process isolation [55] to isolate the uIO process to minimize the attack surface.

With uIO bridging the semantic gap [16] between hosts and VMs, not only can the owner more easily inspect and debug their VMs, but it also becomes easier for malicious parties within cloud providers to do so and harm confidentiality, integrity or availability. We think, however, that providers have a strong incentive to put strong, systematic security measures in place to prevent rough admins from leveraging uIO because providers are legally liable for such compromises. This notion ties back to the initial assumption that we trust the hypervisor and, thus, the cloud provider.

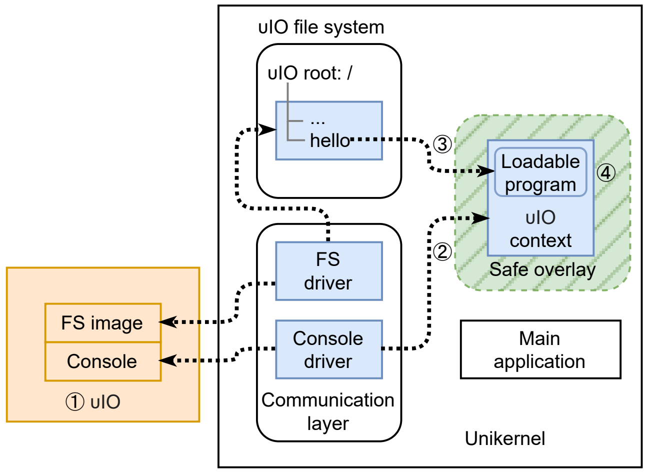

Figure 2: uIO workflow. ***Users can run load uIO-programs from uIO-fs and run it via uIO-console.***

We do not assume malicious uIO-programs, but we expect that uIO-programs may contain bugs. We provide flexible resource isolation to minimize the possibility of uIO-programs accidentally compromising the unikernel. Specifically, we provide two isolation mechanisms; one utilizes lightweight hardware-assisted memory domains where uIO-programs still have maximum programmability (§ 4.4), and the other uses a verified language runtime, namely eBPF [54], to provide a stronger safety guarantee (§ 4.5). The users can choose the desired isolation mechanism depending on their requirements.

3.3 Design Challenges and Key Ideas

We next discuss the challenges and key ideas to realize the *overlay* abstraction in unikernels.

#1 Generic overlay interface in unikernels. As described in § 3.1, it is imperative to have a communication mechanism between the host and the unikernel to realize uIO. Unfortunately, unikernels might lack a generic and minimalistic interface for this purpose, which is readily available in traditional OSes, such as the POSIX socket API [61].

To overcome this problem, we define a minimal communication interface on top of the generic VirtIO protocol [86, 94] (§ 4.1). The VirtIO protocol is a standardized interface specifically designed to communicate between virtual machines and the host system. Using VirtIO as the foundation for communication between uIO components minimizes the amount of code, which is desirable for unikernels [64, 67, 71]. uIO uses VirtIO for both console access and file system.

#2 Dynamic program loading and execution. uIO supports the loading and execution of uIO-programs to maintain the lightweight nature of unikernels. The problem here is that, unlike a general-purpose operating system, unikernels do not have a way to execute new programs as processes. Adding support for multiprocessing in unikernels contradicts its design philosophy.

To address this issue, uIO context utilizes threading and implements a mechanism to load and execute external programs within its context dynamically. When loading, uIO context performs necessary symbol relocations, allowing the loaded program to access and call the unikernel's data and functions without being compiled every time to the specific unikernel. We also provide language-runtime-based execution for safer execution (see the next paragraph.)

Table 1: Main uIO APIs (excerpt)

Type	API function

Console	`char* uio_console_gets()`
	`uio_console_puts(char *)`

File system	`fopen(), fclose(), fread(), fwrite()`
	`lseek(), fsync(), unlink() mkdir()`
	`rmdir(), readdir()`

Memory	`void* uio_alloc_memory(size, attr)`
	`void uio_free_memory(void* addr)`
	`void uio_(enable|disable)_write()`

Link	`void* uio_symbol_get(char* symbol)`

#3 Lightweight safety. uIO context shares the same memory space as the main application, making it easy to introspect the main application. However, it also means that bugs in the uIO context could affect the main application. This problem becomes even more significant when running loadable uIO-programs, as these programs may be developed by different parties than the main application and may not have been adequately tested as compared to the main application. Note that as discussed in § 3.2, we do not assume malicious uIO-programs, but still assume the possibility of the presence of bugs in them.

To mitigate these risks, we use a hardware-assisted intra-application memory isolation mechanism [48, 76, 99] to allow uIO context and the main application to share the same memory space but operate in different memory domains (§ 4.4). Within this domain, the uIO context has read-only access to the memory of the main application by default. If some operations require writing to memory, uIO context must explicitly request a change in memory permissions. This configuration helps to reduce the risk of accidental memory corruption while keeping a single address space and without introducing heavy memory space switches.

In addition, the system also provides an eBPF (Extended Berkeley Packet Filter) [8, 54, 82] execution environment to provide a safer code execution (§ 4.5). eBPF is a language runtime designed with verification and JIT in mind, and the runtime can dynamically check several safety properties, including memory safety and the bounded execution time, or verify code in advance [39, 56]. eBPF programs run in a sandboxed environment and cannot directly access the unikernel's memory, but can call pre-defined helper functions. Therefore, even if there is a bug in the eBPF program, we can minimize the impact on the main application and allow the main application to continue.

4 DESIGN

We next describe the design of uIO. The safe overlay abstraction is realized using uIO-program (§4.3), uIO context (§4.4), and uIO isolation mechanisms (§4.5). uIO supports a generic interface to communicate with safe overlays (§4.1) and a file system interface for on-demand extensibility (§4.2).

4.1 uIO Interfaces

uIO defines an interface between the host side process and in-unikernel components on the one hand, and between unikernels and in-unikernel components on the other hand. Introducing such generic interfaces on both sides of the virtualization layer requires a communication layer and associated APIs.

VirtIO-based communication layer. The uIO communication layer manages the I/Os of uIO-console and uIO-fs. The layer should be lightweight to keep the unikernels benefits and generic enough to support multiple hosts and unikernels. To this end, uIO adapts the VirtIO [86, 94] protocol as the basis of the communication. VirtIO is a virtual device designed for use in virtual environments and supports various device types, including consoles, file systems, networking, and block devices. Nowadays, VirtIO is widely used in virtualization environments, including lightweight hypervisors [2] and unikernels [64, 67]. These characteristics make VirtIO suitable for uIO communication. Especially, uIO uses VirtIO-console [114] for console and VirtIO-based file system (namely VirtIO-9P [115] or VirtIO-fs [113]) for file systems (§ 4.2)

uIO APIs. uIO also defines a common interface between in-unikernel uIO context and the underlying unikernel to ease portability among unikernels. The uIO context uses the APIs to implement core operations. Table 1 shows the list of main uIO APIs. This defines communication functions to interact with uIO-console, uIO's file system, as well as memory allocation and management interfaces. This is necessary to allocate executable memory for uIO-programs as well as change the isolation configuration. uIO APIs also have a symbol resolution function for symbol relocation when loading uIO-programs.

4.2 uIO-fs

uIO provides a file system interface for extensibility while keeping unikernels lightweight. In particular, there are two ways to handle file systems. One uses block devices, and the other uses a file system protocol. The former can achieve better performance, but it requires unikernels to implement the entire file systems such as EXT4. On the other hand, with the latter approach, unikernels offload the file system operations to the host side, thus reducing the code size in unikernels. uIO adopts a protocol-based file system for this reason.

VirtIO defines two devices for file systems: VirtIO-fs [113], and VirtIO-9p [115]. VirtIO-fs uses FUSE protocol [58] whereas VirtIO-9p uses 9p protocol [90] over VirtIO. In both cases, the actual file manipulations occur on the host side; thus, uIO can support a range of available file systems on the host without bloating unikernels. It is straightforward to implement uIO file system API on top of them as their protocols already support such operations. Table 2 shows the related 9p (precisely, 9p.2000L) and FUSE operations to realize the file system APIs.

4.3 uIO-program

A uIO-program is a loadable component of the uIO context to maximize extensibility while keeping lightweightness. uIO-programs run with the same capabilities as the uIO context (§ 4.4). Since unikernels do not have a mechanism for executing new processes, uIO context adapts a *load-and-call* execution model, which loads programs in its memory space and then calls them.

Loading and executing. Traditional operating systems define a stable interface with user programs in the form of system calls [62], and the standard library (libc) [60]. However, in unikernels, the defined interface varies. To achieve maximum flexibility, uIO allows uIO-programs to access any data and call any functions that unikernels export. The standard way to export functions and data is a shared library [59]. However, since unikernels link everything at compile time, shared libraries are not an appropriate abstraction to make symbol exports. Therefore, a uIO context directly links loaded programs to the unikernel application like Linux kernel modules [95].

The uIO context loads uIO-program from uIO-fs and places it to the executable memory region allocated by the memory API (Table 1). When loading a uIO-program, the uIO context performs symbol relocation to ensure that the program can correctly access and call functions of the unikernel, even if the program was not originally built for that specific unikernel—unless function calls maintain the same API. The uIO context executes the loaded program by calling the program's entry point.

4.4 uIO Context

uIO context is an isolated execution environment of uIO for unikernels. It shares the same address space as the main application, can access the main application's data, and can call its functions. Further, uIO context is a schedulable entity and can work concurrently with the main application. It can also load external programs to access the main application's data and call its functions. Conceptually, a uIO context is similar to a traditional OS thread but provides a memory isolation mechanism between itself and the main application (§ 4.5).

We create the uIO context thread at boot time. This thread shares the address space with the main application and has its main loop, where it waits for requests from the uIO-console and processes them when they arrive. When there is no data from the console, the thread sleeps.

Scheduling. Many unikernels only have a cooperative scheduler [13, 67, 79], meaning other threads run only when the currently running thread voluntarily yields. This is efficient in terms of performance because there are no context switches due to scheduling upon regular timer interrupts. However, this can be problematic for a uIO context, as it will only be scheduled if the main application yields. A bit surprisingly, we find that the cooperative scheduler works well for uIO in our use cases (§ 6.4). Most applications have I/Os for networking and/or storage, which trigger scheduling.

The application might use busy polling to improve I/O performance. Even in these cases, we can still schedule other threads when there is no data to process. For example, Unikraft's network stack (lwip [34]) supports polling mode but still yields to other threads regularly. We find that this works well in practice. For instance, we confirm that we can use uIO without severe performance degradation while running Nginx in polling mode and stressing the server (§ 6.2).

Table 2: Protocol messages for uIO-fs operations

Operation	9p2000.L	FUSE
`fopen()`	`open`	`FUSE_OPEN`
`fclose()`	`clunk`	`FUSE_RELEASE`
`fread()`	`read`	`FUSE_READ`
`fwrite()`	`write`	`FUSE_WRITE`
`lseek()`	`read/write` + offset	`FUSE_LSEEK`
`fsync()`	`fsync`	`FUSE_FSYNC`
`unlink()`	`unlinkat`	`FUSE_UNLINK`
`rename()`	`rename`	`FUSE_RENAME`
`mkdir()`	`mkdir`	`FUSE_MKDIR`
`rmdir()`	`unlinkat`	`FUSE_RMDIR`
`readdir()`	`readdir`	`FUSE_READDIR`

4.5 uIO Isolation

As we discussed in § 3.2, we do not assume malicious uIO components, including uIO-programs, but they may have bugs. The uIO context memory isolation aims to prevent uIO contexts (including uIO-programs) from accidentally overwriting or mutating the state of the main application. We next describe the two isolation mechanisms provided by uIO for safe overlay execution: (a) Hardware-assisted isolation with memory protection keys, and (b) Language-runtime-based isolation with eBPF.

Hardware-assisted memory isolation. We adopt a hardware-assisted intra-application protection mechanism [48, 76, 99], which allows sharing address space but having different read and write permissions for each domain, to realize isolation between the uIO context and the main application.

In particular, we leverage Memory Protection Keys (MPK) [6, 48] to realize uIO context isolation to reduce the risk of it accidentally overwriting the main application. MPK is a memory protection mechanism for user pages available on recent x86_64 CPUs [6, 48]. MPK uses the upper bits of the page table entry to assign one of 16 domains to each page, and the CPU-local PKRU register controls read and write rights to the domain. Non-privileged WRPKRU instruction updates the PKRU register without involving a TLB flush, achieving a fast domain switch with a few dozen cycles [88].

We create two memory domains, one for the uIO context and the other for the main application. Originally, the uIO context runs in a domain where it does not have write but only read permission of the main application's memory. However, if uIO needs to write to that memory region, uIO-program code explicitly changes the permission via API. This configuration helps to reduce the risk of a uIO context accidentally overwriting the main application's data.

Several recent studies have also employed intra-application memory protection to create compartments within processes [45, 74, 75, 96, 102, 111]. The main difference between the isolation of uIO from these works is that these studies aim to achieve isolation among multiple mutually untrusted components. At the same time, we strive to create isolation between uIO and the main application, with the assumption that neither is malicious. This simplifies our compartment design.

Language-runtime-based isolation. In addition to the uIO context isolation based on memory domains, uIO offers a restrictive language-runtime-based isolation mechanism based on the

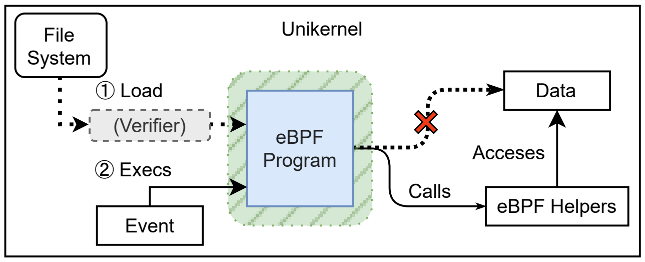

Figure 3: Overview of the eBPF execution environment. ***eBPF program runs in a sandboxed environment and cannot directly access the unikernel's data nor execute native instructions, but it can call pre-defined helper functions.***

eBPF execution environment. Figure 3 shows the overview of the eBPF execution environment. Users load the eBPF program via uIO-console (①) and execute it. In addition, users can also attach eBPF programs to some events so that they trigger them (②). For example, we can trace functions by attaching eBPF programs at a function's entry. We describe the implementation of eBPF-based function tracing in § 5.2.

eBPF program runs in the sandboxed environment with an interpreter. Although the eBPF program cannot directly call the unikernel's functions, the execution environment defines several helper functions that are callable from the eBPF program. The interpreter dynamically check several safety properties, such as memory safety, and guarantee bounded execution time by limiting the number of instructions to execute. Optionally, we can integrate eBPF verifier [39, 56]. The verifier ensures that the loaded eBPF program does not violate the safety properties in advance, allowing removing the dynamic safety checking and JIT-ted execution for better performance.

Host-side resource accounting. uIO uses a process group resource isolation mechanism (namely cgroups [55] in Linux) on the host to ensure that the uIO process shares a total amount of resources with the VM and does not use more resources than allocated to the VM. This also limits recourses uIO can access, making it difficult for attackers to exploit the uIO process to gain control of the VM.

5 IMPLEMENTATION

We implement the prototype of uIO on top of Unikraft [67] version 0.9.0 (Hyperion) [107] targeting x86_64 with QEMU/KVM [63]. We consider type-2 hypervisors such as QEMU/KVM [63] for our implementation. Our proposed system is applicable for type-1 hypervisors such as Xen [10] as well by running the host components on another VM. We do not add hypervisor-specific code. Therefore, supporting other hypervisors should require little effort.

Unikraft has support for multithreading with a cooperative scheduler and VirtIO-9p. We add support of VirtIO-console [94], eBPF runtime [106], and some basic MPK [6, 48] operations in the Unikraft. We add support of VirtIO-console [94] for uIO-console. We use VirtIO-9p for uIO-fs and use Unikraft's vfscore [27] library to realize uIO-fs I/O APIs (Table 1). We explain the details of the uIO user interface and programs (§ 5.1), the language-runtime-based

Table 3: Main uIO commands

Command	Description
load	Load a symbol file
run	Execute uIO-program
ls	List directory contents
cat	Show file contents
bpf_exec	Execute a eBPF program
bpf_map_get	Get a value from eBPF map
bpf_map_put	Put a value to eBPF map
bpf_attach	Attach a eBPF program to a function

isolation with eBPF (§ 5.2), and the hardware-assisted isolation with MPK (§ 5.3) in the following sections.

The main part of uIO has around 4.1 KLOC, including eBPF dynamic tracing support, and in addition, ubpf interpreter consists of 1.6 KLOC. uIO relies on several Unikraft libraries, including vfscore [27] and its virtio drivers. As our evaluation indicates (§ 6), normal applications also rely on these libraries, and thus uIO does not significantly increase the image size.

5.1 uIO Interface and Program

uIO-console. Users communicate with uIO-context using a console interface with VirtIO-console. For this purpose, we add a VirtIO-console driver to Unikraft. We use existing QEMU's VirtIO-console [114], and we configure QEMU to assign a VirtIO-console device to a unikernel. QEMU creates a socket file on the host with which host applications can interact with the console.

uIO-fs. The current uIO implementation uses VirtIO-9p. We configure QEMU so that it uses the specified directory as a uIO-fs. We use Unikraft's vfscore library [27] to realize uIO-fs I/O APIs (Table 1).

User interface. uIO provides built-in commands. Table 3 shows the list of main uIO commands. The `run` command is the main command in uIO, which executes uIO-programs. If the program to run has not been loaded into the unikernel's memory, the `run` command first retrieves the program using uIO-fs and performs the necessary loading and linking. Once the program has been loaded, it is cached in the unikernel's memory for faster execution in the future. We also implement common directory and file commands `ls` and `cat` as built-in for convenience. These commands support both uIO-fs and file systems that the application uses (e.g., ramfs), if any. We also define several eBPF-related commands (see § 5.2).

uIO-program. We implement uIO-programs (§ 4.3) as position-independent relocatable ELF objects [57]. The uIO context loads an ELF file from uIO-fs upon an execution request. During the load, uIO resolves any relocation entries in the ELF sections, which tells the loader where to place certain data or code in memory, with the help of `uio_symbol_get()`. We implement `uio_symbol_get()` by consulting symbol address information. We create the symbol file by extracting symbol information from the debug binary build. The current implementation assumes that unikernels keep the necessary symbols that uIO-programs require. We could solve this limitation by supporting on-demand loading of the dependent code.

Table 4: eBPF helpers

Function	Description
bpf_map_get()	Get a value from eBPF map
bpf_map_put()	Put a value to eBPF map
bpf_get_addr()	Get symbol address
bpf_probe_read()	Read from memory with safety check
bpf_puts()	Put a message to a uIO-console

5.2 eBPF Integration with uIO

We implement the eBPF execution environment by integrating user space eBPF runtime (ubpf [106]). We use ubpf's interpreter to execute eBPF programs. ubpf has no verifier, but its interpreter has several safety checks, including detecting simple infinite loops, dynamic memory access bounds checking, and zero division checking (see § 6.1 for the safety evaluation). We limit the maximum number of instructions to execute in one eBPF program to one million to ensure the termination of the program. An eBPF program can only access its 512-byte stack and bounded memory region given as an argument to the program. Users execute an eBPF program with the `bpf_exec` command, and the argument to that command is passed to the eBPF program. In the current implementation, the eBPF runtime runs the same isolation domain as the main application (see § 5.3).

We also implement several eBPF helper functions listed in Table 4 and a simple key-value store (eBPF map). An eBPF program can access the map with helper functions. uIO provides `bpf_map_get` and `bpf_map_put` commands to access the map from the uIO-console. `bpf_probe_read()` helper function reads from memory with a specified address, but if the address is not valid, then it returns zero.

Tracing with eBPF. To show the flexibility of eBPF, we implement a prototype of a dynamic tracing mechanism like ftrace [35]. To use this mechanism, we insert nop instructions at the entry of functions when compiling using gcc's mcount features [38]. Users can choose whether to insert nop for each Unikraft library, and depending on the size of the library, the application size increases from several hundred bytes to several kilobytes. The uIO runtime replaces the nop instructions with a eBPF program call when attaching eBPF program with the `bpf_attach` command. The argument to the eBPF program is the address of the attached function. An example of tracing is counting a number of function calls (§ 6.4), by retrieving, incrementing, and restoring the count number from the eBPF map using a function address given by the argument as a key.

5.3 MPK Integration with uIO

We create two MPK domains, one for the main application and one for uIO. When running the main application, there is no restriction by MPK. When running the uIO context, writing to the main application's domain is prohibited unless the program changes the permission explicitly. MPK protect violation results in a segmentation violation (SEGV).

To use MPK in Unikraft, we make all page entries as user pages, which means all privilege levels can access these pages. Note that this change does not introduce any security problems for unikernels, as the application and kernel run in the same address space by its nature. Initially, we associate all pages, including data and bss for uIO, with the main application's domain. When uIO starts, we associate (1) its stack, (2) interrupt handler's stack, and (3) memory allocated in the uIO context using API (Table 1) with the uIO domain. We need to change the domain of the interrupt handler's stack so the CPU can save registers when receiving interrupts in the uIO context. When switching the context between the main application and uIO context, including when receiving interrupts, we change the current CPU's MPK permission with WRPKRU instruction. We provide a gate function as a macro to simplify domain switching.

Limitation. MPK is page granularity. Therefore it cannot prevent within-page out-of-bound memory access. In addition, as we do not restrict any use of the WRPKRU instruction, our MPK isolation does not prevent bugs in which uIO-programs wrongly change memory permissions. Also, MPK does not prevent misusing synchronization functions resulting in deadlocks, nor guarantee termination of uIO-programs. As we discuss in § 3.2, we do not assume the presence of malicious uIO-programs, and the main purpose of the isolation is to protect the main application from accidental memory write from uIO. Note that the eBPF execution environment restricts such operations.

6 EVALUATION

We evaluate uIO across the following dimensions: robustness (§ 6.1), performance (§ 6.2), and effectiveness (§ 6.3). Finally, we present the evaluation of five use cases (§ 6.4).

Experimental setup. We perform the experiments on a machine equipped with an Intel Xeon Gold 5317 CPU running at 3.00 GHz (12 physical cores with hyper-threading disabled) and 256 GiB of DDR4 memory. The host operating system is Linux 6.2.12, and we use QEMU 7.2.0 [11] with KVM [63] as the hypervisor. We assign one vCPU and 1 GiB of memory for all experiments to unikernels. We use VirtIO-net for networking between unikernels and the host. For better reproducibility, we disable Intel Turbo Boost and pinned QEMU, vCPU threads, and benchmark tools to different, isolated, physical CPUs.

Applications. We use Unikraft's ported version of Nginx [108], Redis [109], and SQLite [110] in the experiments. Nginx uses lwip [34] with polling mode for network processing. Nginx and SQLite use ramfs for the storage of their contents. We run a client program that communicates with these applications on the host. We also use a simple application that prints counter numbers to a serial console per second (denoted as *count*) as an example of a minimal baseline application. We also use several uIO-program presented in § 6.4 in the evaluation.

Experiment variants. We use the Unikraft application without uIO as a baseline (denoted as Unikraft). As a comparison, we use the application with uIO with enabling eBPF tracing support (denoted as uIO). In several experiments, we also use uIO without MPK isolation to see the MPK overhead (denoted as uIO*).

6.1 Robustness

RQ1. *How much robustness does uIO provide?* We evaluate the robustness of the uIO and, more precisely, its isolation mechanism and the eBPF execution environment.

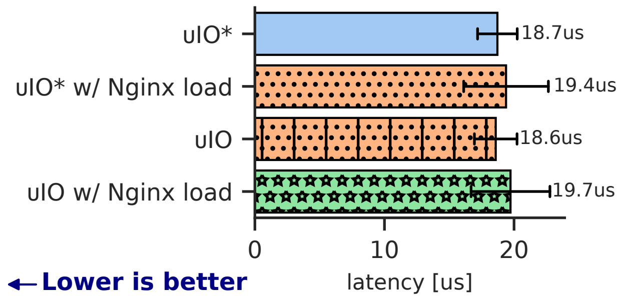

Figure 4: uIO console responsiveness. "Nginx load" means performance under stressing the server with wrk [40].

A: MPK isolation. First, we intentionally inject faulty code snippets in uIO and uIO-programs, which try to write to the main application's memory and call the main application's functions without explicit domain changes. In the second experiment, we modify the snippets and wrap memory writes and function calls with proper macros so that they change memory permission explicitly. We confirm that in the first experiment, writing to the main application's memory causes segmentation faults thanks to the MPK isolation. On the other hand, in the second experiment, memory writes, and function calls succeed.

B: eBPF execution environment. We create and execute two eBPF programs that do (1) infinite loops and (2) access out-of-bound memory. As described in § 5.2, our eBPF execution environment employs dynamic safety checks using an interpreter. We confirm that our eBPF execution environment safely handles both operations (stopping the execution of eBPF programs within a certain amount of time) and allows the main application to continue its execution.

RQ1 takeaway: MPK isolation mechanism successfully detects accidental memory writes from the uIO context to the main application. eBPF execution environment successfully prevents infinite loops and out-of-bound memory access and allows the main application to continue its execution.

6.2 Performance

RQ2. *How much performance overhead does uIO have?* We evaluate uIO's performance from several perspectives. First, we present the console responsiveness; then, we show the uIO-program loading time. We also report resource footprints regarding disks and memory, and present the main applications' performance under uIO with three real-world applications: Nginx, Redis, and SQLite. Lastly, we report the performance of uIO-fs.

A: Console responsiveness. We evaluate the uIO-console responsiveness to see if it is fast enough for interactive use even under heavy load. Note that our implementation uses a cooperative scheduler. We first attach to uIO-console, then we measure the round-trip time of sending an empty newline and receiving the prompt message 30 times. We report the mean values with the standard deviations.

Figure 4 shows the results. Our measurements show that the uIO's average latency is 18.6 µs. Even when attaching uIO-console to an Nginx under load with a stressing tool wrk [40], the average latency only increases by 1.1 µs. MPK isolation introduces little

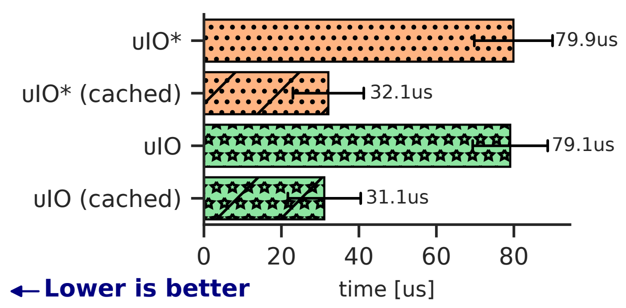

Figure 5: uIO response times to load the `perf` uIO-program.

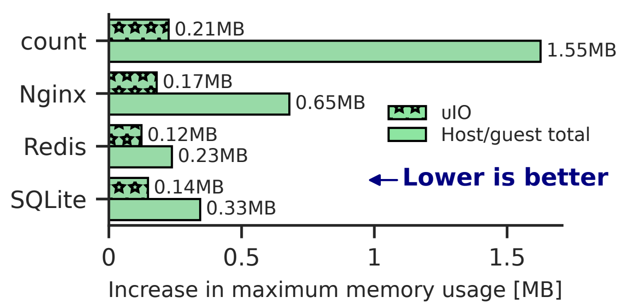

Figure 6: Increase in maximum memory usage with uIO. uIO bar shows the increase in the unikernels, and the other shows the increase in the total memory measured in the host.

overhead. To summarize, our evaluation shows that uIO-console achieves practical responsiveness with a cooperative scheduler.

B: uIO-program loading time. We measure uIO-program loading time to evaluate the performance of the loader and the linker of uIO. We measure overall load time by sending the command to load `perf` program (see § 6.4 #4 for the detail). Figure 5 shows the results. uIO takes 79.1 µs whereas the cached version reduces loading time by about 61%. The isolation causes negligible overhead.

C: Disk and memory footprint for uIO. We measure the disk and memory footprint needed for uIO. First, uIO requires symbol files to load uIO-programs, and their sizes are 86KB (count), 170KB (Nginx), 227KB (Redis), and 154KB (SQLite), respectively. This file is a text file containing symbol names and their addresses. Using binary format and compression will decrease the size.

Second, we measure the memory footprint increase in the unikernel using the Unikraft memory allocator's statistics interface [25]. In addition, we measure the total memory footprint by running all uIO components in the same cgroup and calculating the entire memory usage. This includes memory used by VirtIO devices in QEMU and `socat` process for console communication. Figure 6 shows the increase of maximum memory usage we measured at the entry point of the main function. We observe a total 0.23MB (Redis) to 1.55MB (count) memory increase when using uIO. This is about 0.1 % memory when we assign 1GiB to the VM. As discussed in § 3.2, a cloud provider would account for these resource increases as resources are used by the unikernel application.

D: Application performance under uIO. We evaluate application performance under uIO with three applications: Nginx, Redis, and SQLite. For Nginx, we run wrk [40], an HTTP benchmark, for one minute with 2 threads and 30 HTTP connections. For Redis, we run redis-benchmark and made 10 million requests with 30 connections and a pipelining level of 16. For SQLite, we use the benchmark program [19], which performs insertion operations to an SQLite database located in a ramfs and reports the processing time. We run experiments ten times and report mean values.

In addition to running the application normally, we also measure the performance when (1) running the `ls` command in uIO once per second ("w/ ls"), and (2) running the `perf` uIO-program, which shows the values of performance counters once per second (see § 6.4) ("w/ perf"), and (3) attaching eBPF program via uIO and count a number of function calls ("w/ bpf"), as an example of eBPF execution. Specifically, we trace `ngx_http_process_request_-line()` in Nginx, and `processCommand()` in Redis. The application calls each function when receiving a request. As for SQLite, we do not perform the experiment with uIO running programs because insertion operations do not yield the thread, and thus uIO-context does not run. This experiment evaluates the overhead when the application does not use uIO. We repeat each experiment ten times to report mean values with the standard deviation. Figure 7 shows the results.

First, comparing the uIO*, which has no MPK isolation, and uIO results, we observe less than 1.9 % performance differences (the biggest one is Nginx w/ perf). Second, in the Nginx and Redis experiment, we observe 2.1 (Nginx) to 6.2 (Redis get) % performance overheads when integrating uIO (without doing anything on it), whereas there is no overhead for SQLite insertion operations. We presume this overhead comes from thread scheduling. Nginx and Redis try to switch contexts after doing I/Os, even though the uIO context sleeps at that time, adding additional processing time. On the other hand, SQLite insertion operations do not involve any I/Os and, thus, no context switch, leading to little overhead. Third, uIO-program and eBPF tracing overhead depend on what we do. Running `ls` command introduces little overheads, whereas running `perf` uIO-program introduces 4.2 to 8.2 % performance overheads compared to when we do not run anything on uIO. We observe up to 31 % performance overheads when tracing `processCommand()` function in Redis. The important point here is that uIO works well even when we load the application, and our eBPF execution environment also works fine when attaching an eBPF program to one of the busiest functions in the application.

Our experiments show that even if the application intensively processes I/Os (Nginx and Redis), uIO can operate with only several percent performance overheads. We expect that we could eliminate the overhead when not using uIO by injecting the uIO code and running the uIO context dynamically [105]. The SQLite insertion benchmark shows that uIO does not introduce any overheads when the application does not yield it, though uIO does not run in this case. A preemptive scheduler will solve this problem. However, this benchmark – not having any I/O – is an extreme case, and we do not expect that this will be a common case.

E: File system performance. We measure the performance of uIO-fs over VirtIO-9p. We perform sequential read/write to a 2GB file on uIO-fs and measure the throughput with several buffer sizes in a single read/write operation. The host uses ZFS as a file system

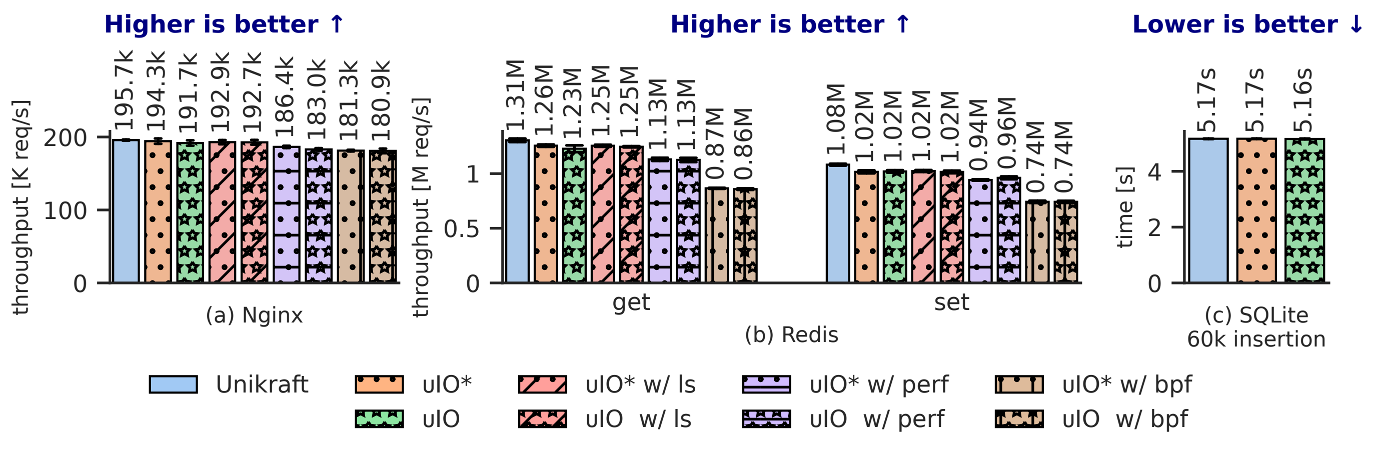

Figure 7: Application performance under uIO. uIO* is no MPK isolation. (a) Nginx throughput with wrk. (b) Redis throughput with redis-benchmark. (c) SQLite insertion time for 60k rows. "w/ ls" and "w/ perf" means we run the commands in the uIO every second. "w/ bpf" means we attach an eBPF program to some of the function calls to count the number of calls.

and stores the data there. We compare the results with VirtIO-blk with ext4 on a Linux VM as Unikraft misses the support of VirtIO-blk with file systems. We use `O_DIRECT` to bypass the page cache during the experiment. Figure 8 shows the result.

Compared to the VirtIO-blk on a Linux VM, uIO-fs over VirtIO-9p has up to 17% overhead for read and 74% overhead for write. In general, block-based I/O performs better than file-based I/O [105] but simplifies the guest-side implementation. As the primary purpose of uIO is to provide a safe overlay for extensibility and not to offer high-performance storage, we think the performance is acceptable for uIO. The guest application can use other file systems, such as ramfs, in parallel with uIO to get better performance.

RQ2 takeaway: (1) uIO achieves practical responsiveness even with a heavy network load. (2) uIO-program caching effectively reduces loading time, resulting in faster responsiveness. (3) uIO's resource cost is several hundred KB of memory in unikernels and up to 1.6MB in the host. The symbol file takes several hundred KB. (4) uIO works well even when we load the application with moderate overhead. (5) File-based I/O (9pfs) has lower performance than block-based storage but uIO uses it to prefer its simplicity.

6.3 Effectiveness

RQ3. *How much lightweightness does uIO bring?* We evaluate the effectiveness of uIO for building lightweight unikernels by quantifying the image and the program size needed to use uIO. uIO is implemented as a library, increasing the image size when integrating it. However, it is important to note that without uIO, it is impossible to realize any use case presented in § 6.4. In this experiment, we show that the increase in image size is minimal, making it a suitable choice for practical use.

We build applications with and without uIO. This experiment enables size optimization (link-time optimization and dead code elimination). We enable eBPF tracing support when integrating uIO, but disable mcount (not insert any nops). Figure 9 shows the result. uIO increases the image size by from 60 KB (SQLite) up to 190 KB (count). uIO relies on some Unikernel libraries, such as vfscore and some libc functions, and Nginx, Redis, and SQLite also rely on such libraries, resulting in smaller size differences.

The uIO-programs we show in § 6.4 are around several KB, and eBPF programs are around several hundred bytes. For example, `perf` program is 4.1 KB and an eBPF program for counting the number of function calls is 104 B. This is not larger than the size required by the uIO. However, it is important to note that without uIO, we cannot run any uIO-program in the first place. Additionally, uIO can load programs depending on some large libraries. For example, the size of libc (newlibc.o) we use to build applications is 0.6 MB (many parts of libc are eliminated by size optimization). Potentially, we could create uIO-programs depending on this libc and load them interactively.

RQ3 takeaway: With around several hundred KB, we can enable uIO and have the ability to load and run external programs. Without uIO, we cannot interactively execute any uIO-programs, including eBPF programs.

6.4 Implemented Use Cases

RQ4. *What kind of use cases can uIO support?* We implement five use cases to show the usefulness of uIO. Table 5 summarizes the main evaluation results.

Use case #1: Interactive debugging. First, uIO can provide an interactive debugging environment. For example, using ramfs [24] to improve Nginx performance in unikernels is crucial. However, this makes it challenging to access log files, such as `/nginx/logs/error.log`, as they only exist in the unikernel's memory. The uIO provides access to the ramfs via uIO-console, and we can view the contents of these files by executing the `cat` command. This is a simple yet useful example that uIO makes possible.

Use case #2: Online re-configuration. Nginx supports on-demand configuration reloading by sending a SIGHUP signal [30].

Table 5: Evaluated Use Cases

No.	Application	Type	Description	Implementation	Performance overheads Native [rps]	w/ uIO (overhead %)
1	Nginx	Debugging	Inspect log files in ramfs	uIO command	N/A (one-shot execution)	
2	Nginx	Management	Configuration reloading	uIO command	N/A (one-shot execution)	
3	SQLite	Auxiliary task	Dumping in-memory DB to a file	uIO-program	N/A (one-shot execution)	
4 (a)	Nginx	Monitoring	Monitor performance counters	uIO-program	195.7k	183.0k (6.5)
4 (b)	Redis	Monitoring	Monitor performance counters	uIO-program	1.31M/1.08M (get/set)	1.13M/0.96M (13.7/11.1)
5 (a)	Nginx	Tracing	Count number of requests	eBPF	195.7k	180.9k (7.6)
5 (b)	Redis	Tracing	Count number of get/set	eBPF	1.31M/1.08M (get/set)	0.86M/0.74M (34.4/31.5)

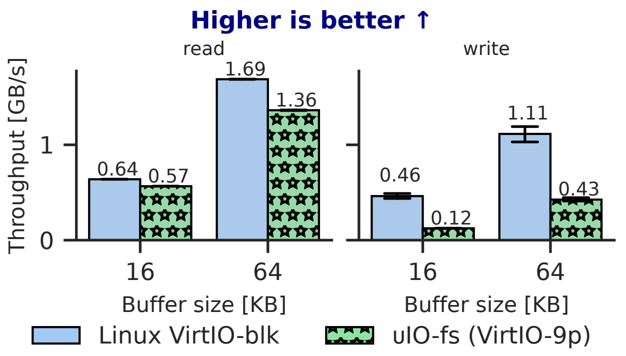

Figure 8: File system throughput.

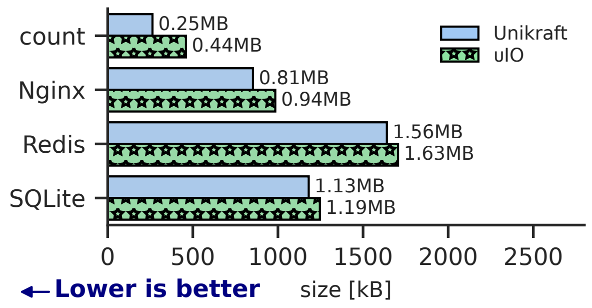

Figure 9: Image size comparison when integrating uIO

Although Unikraft supports basic signal mechanisms [26], the mechanisms for sending signals are unavailable unless the application implements the dedicated interface. We add built-in `kill` command in uIO that allows us to send signals to the main application, enabling on-demand configuration reloading. This is helpful for observing the application behavior quickly with different configurations.

Use case #3: SQLite online backup. uIO is also useful for invoking the application's auxiliary tasks. For example, SQLite supports saving the in-memory database to a file [101]. However, without uIO, there is no interface to call that function on-demand. We create a small uIO-program that calls the backup function. From uIO, we can back up in-memory SQLite data by executing that program.

Use cases #4: Performance monitoring. Performance monitoring is also where uIO shines. CPUs have an internal register for performance monitoring [49]. Without uIO, configuring and reading such registers is difficult due to a lack of interface. To demonstrate monitoring capability, we create a uIO-program called `perf` that configures and prints performance counters of instructions counts and LLC misses every second. With this program, we can get performance counter values while executing the main application.

Use cases #5: Safe application inspection and tracing with eBPF. uIO is also useful for inspecting and tracing the application's behavior. In particular, eBPF can guarantee stronger isolation yet provide flexibility to examine applications. For example, we create an eBPF program that safely consults the application's symbol data. This program takes the symbol name as an argument, and it resolves the symbols using `bpf_get_address()` and then reads its value with `bpf_probe_read()`. These helper functions ensure safe memory access. With this program, for instance, we can read Nginx's `connection_counter` to get the connection count.

In addition, we create an eBPF program that counts the number of function calls as described in § 5.2. We can attach the eBPF program to functions via uIO-console, and consult the number by executing the `bpf_get_map` command. The performance evaluation (Figure 7) shows that this tracing successfully works when attaching one of the busiest functions in the Nginx and Redis with moderate performance overhead.

RQ4 takeaway: We show that through the use cases, we can use uIO for (1) debugging, (2) interactive management environment, (3) running auxiliary tasks, (4) performance monitoring, and (5) safe inspection and tracing with eBPF. The traditional unikernels have trade-offs between extensibility and lightweightness; performing these tasks is challenging. With uIO, we are able to get both worlds, enabling users to extend unikernels on-demand only when needed.

7 RELATED WORK

Extensibility in virtualized environments. Traditional OSs have a mechanism for managing and extending VMs remotely. ssh [87] is the most common way to interact with a remote machine, which allows users to install and run additional programs on demand. Some cloud providers also offer dedicated agent software for this purpose [5, 41]. However, these methods are not directly applicable to unikernels as they would only have non-standard specialized interfaces and usually do not support spawning new processes.

VMSH [105] proposes a way to have a hypervisor-agnostic overlay without pre-installing any code in VMs. It creates VirtIO devices and spawns an agent process on demand by injecting code from the host. uIO is similar to VMSH in providing an interface with VirtIO devices. Still, VMSH's focus is achieving dynamic guest overlay on Linux in a (KVM-based-)hypervisor-agnostic manner, and it heavily relies on Linux kernel functions both in the guest and the host. On the other hand, uIO provides interactive overlay environments tailored to unikernels with lightweight isolation.

Hypershell [36] utilizes system-call redirection to make host-side applications run in the context of the guest. This allows applications to be run on the guest without installing them. However, this assumes that the guest and the host use the same kernel and does not apply to unikernels.

Hyperupcalls [8] proposes a mechanism that the hypervisor safely executes guest-provided BPF code in the host to realize flexible and less intrusive upcalls. On the other hand, uIO loads and runs BPF program in the guest to provide extensibility to the application.

Debugging unikernels. One of the main use cases of uIO is to debug unikernels interactively. The current most used ways to debug unikernels is using gdb [32] with the gdb stub provided by hypervisors [33, 116], or embedding the stub in unikernels [20, 22]. In addition, several approaches create profilers for unikernels [3, 21, 42, 85, 98]. These systems are orthogonal to uIO. Generally, gdb-style inspection requires much low-level knowledge and does not offer overlay to run programs in unikernels on-demand. On the other hand, uIO can provide an interactive overlay where users can examine the application with running loadable programs.

Running unikernels as processes on a generic OS [23, 53, 117] makes using debug tools for common applications possible. However, this technique is not usable in production where unikernels run in a virtualized environment. Lupine [69] and UKL [91] demonstrate that it is possible to tailor general-purpose OS to unikernels-like systems. This also allows us to run traditional management and debug tools, but this technique does not apply to other unikernels. uIO provides extensibility for unikernels running on a hypervisor and can provide a generic interface for debugging.

Isolation in unikernels. Several recent studies propose methods to have hardware-assisted compartments inside unikernels while keeping a single address space [74, 96, 102]. Sung et al. [102] propose an MPK-based isolation mechanism between trusted and untrusted components. FlexOS [74] allows the fine-grained memory isolation with MPK and nested paging to be specified at compile time to achieve a trade-off between security and performance according to requirements. CubicleOS [96] proposes an efficient zero-copy data-access mechanism across partitioned compartments with MPK.

In the same spirit, uIO utilizes MPK to achieve memory isolation, but our assumption makes domain design simple. We create isolation between the uIO and the main application to reduce the risk of uIO accidentally compromising the main application. uIO also offers language-runtime-based execution with eBPF to have stronger isolation and safety guarantees.

fork() in unikernels. 'fork()' [75, 77, 119] in unikernels mainly aims at running multi-process applications such as web servers. Iso-UniK [75] re-introduce page table isolation within unikernels, and KylinX [119] and Nephele [77] realize fork() through VM cloning. Using a fork is another possible approach to realize overlay, but uIO prefers multithreading as many unikernels already support it, and it fits the single-address space philosophy of unikernels. uIO leverages hardware-assisted isolation and language-runtime-based isolation to compensate for the weak isolation of multi-threading.

File systems in unikernels. Many unikernels support in-memory file systems (e.g., ramfs) [13, 64, 67, 85]. One of the primary use cases of such a file system is loading configuration and serving static files, which is helpful for, e.g., web servers. To use this, the hypervisor loads the file system in memory at boot time. In-memory file systems are fast but not persistent and extensible.

Some unikernels support persistent file systems with block devices. For example, OSv [64] supports ZFS over VirtIO-blk. Also, some support VirtIO-based file systems. OSv and Rusty-hermit [71] supports VirtIO-fs and Unikraft supports VirtIO-9p [67]. Like these unikernels, uIO uses VirtIO-based file systems, and uIO defines generic interfaces on top of them.

MiniCache [68] implements a specialized file system ("SHFS") on top of Xen's paravirtualized block device for CDN. MiniCache also implements a simple interface ("µSH") over a network that allows operators to interact with the unikernel, such as manipulating the cache file and retrieving statistics. However, this is specialized for MiniCache and does not provide a generic overlay for extensibility like uIO. With a combination of VirtIO-console and lightweight isolation mechanism, uIO provides a dynamic, safe overlay for on-demand unikernel extension.

Microkernels. Microkernels [9, 12, 46, 47, 65] implement main OS components as a user-level service, and the kernel only contains the essential functionalities such as inter-process communication and scheduling. uIO shares a commonality with microkernels in the sense that uIO uses the additional service provided by the host component.

8 CONCLUSION

In conclusion, our paper makes the following contributions.

(1) **Unikernel overlays:** We introduce an *overlay* abstraction for lightweight and extensible unikernels. Overlays enable "on-demand extensibility" for deployed unikernels in production, where developers or administrators can run auxiliary tools and workflows for management-related tasks while keeping the unikernel advantages.
(2) **Extensible file system interface:** We present an extensible file system interface for unikernels, along with a loadable program execution environment, based on the standardized VirtIO protocol in virtualized environments.
(3) **Lightweight safety:** We present two lightweight isolation mechanisms for hardening the safety properties of overlays, namely hardware-assisted memory isolation and language-based isolation; we implement these safety mechanisms by integrating MPK and eBPF with unikernels, respectively.

We implement our contributions in the uIO system based on Unikraft [67] and confirm its usefulness in detailed experiments with several real-world use cases, including interactive debugging, re-configuration, data back-ups, performance monitoring, and application tracing with eBPF.

Artifact availability. uIO will be publicly available along with the experimental setup and use cases.

REFERENCES

[1] The Big Idea Around Unikernels | Hacker News. https://news.ycombinator.com/item?id=29427449, 2021. Accessed: 2024-07-07.

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications . In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 2020.

[3] Kareem Ahmad, Alan Dearle, Jon Lewis, and Ward Jaradat. Debugging Unikernel Operating Systems. https://uksystems.org/workshop/2020/slides/presso40.pdf, 2020. Presented at 5th Annual UK Systems Research Challenges Workshop. Accessed: 2024-07-07.

[4] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-Performance Serverless Computing. In *Proceedings of the 2018 USENIX Annual Technical Conference*. USENIX Association, 2018.

[5] Amazon Web Service. AWS Systems Manager Documentation. https://docs.aws.amazon.com/systems-manager/index.html. Accessed: 2024-07-07.

[6] AMD. AMD64 Architecture Programmer's Manual Volume 2: System Programming Revision 3.40 – Section 5.6.7, 2023. Accessed: 2024-07-07.

[7] AMD. AMD Secure Encrypted Virtualization (SEV). https://www.amd.com/en/developer/sev.html, [n. d.].

[8] Nadav Amit and Michael Wei. The Design and Implementation of Hyperupcalls. In *Proceedings of the 2018 USENIX Annual Technical Conference*. USENIX Association, 2018.

[9] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 2016.

[10] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*. Association for Computing Machinery, 2003.

[11] Fabrice Bellard. QEMU, A Fast and Portable Dynamic Translator. In *Proceedings of the 2005 USENIX Annual Technical Conference*. USENIX Association, 2005.

[12] Brian Nathan Bershad, Stefen Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc Eric Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*. Association for Computing Machinery, 1995.

[13] Alfred Bratterud, Alf-Andre Walla, Harek Haugerud, Paal E. Engelstad, and Kyrre Begnum. IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services. In *Proceedings of the 7th IEEE International Conference on Cloud Computing Technology and Science*. IEEE Computer Society, 2015.

[14] Bryan Cantrill. Unikernels are Unfit for Production. https://www.tritondatacenter.com/blog/unikernels-are-unfit-for-production, 2016. Accessed: 2024-07-07.

[15] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. SEUSS: Skip Redundant Paths to Make Serverless Fast. In *Proceedings of the 15th European Conference on Computer Systems*. Association for Computing Machinery, 2020.

[16] Peter M. Chen and Brian D. Noble. When Virtual is Better than Real [Operating System Relocation to Virtual Machines]. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*. IEEE Computer Society, 2001.

[17] Maxime Compastié, Rémi Badonnel, Olivier Festor, Ruan He, and Mohamed Kassi-Lahlou. Unikernel-based Approach for Software-defined Security in Cloud Infrastructures. In *Proceedings of the 2018 IEEE/IFIP Network Operations and Management Symposium*. IEEE Computer Society, 2018.

[18] Vittorio Cozzolino, Aaron Yi Ding, and Jörg Ott. FADES: Fine-Grained Edge Offloading with Unikernels. In *Proceedings of the 1st International Workshop on Hot Topics in Container Networking and Networked Systems*. Association for Computing Machinery, 2017.

[19] Cyril Soldani. cffs/app-sqlite-test: Performance experiment with SQLite on Unikraft. https://github.com/cffs/app-sqlite-test. Accessed: 2024-07-07.

[20] Hermitux developers. Debugging · ssrg-vt/hermitux Wiki. https://github.com/ssrg-vt/hermitux/wiki/Debugging. Accessed: 2024-07-07.

[21] Hermitux developers. Profiling · ssrg-vt/hermitux Wiki. https://github.com/ssrg-vt/hermitux/wiki/Profiling. Accessed: 2024-07-07.

[22] Solo5 developers. Solo5 gdb stub. https://github.com/Solo5/solo5/blob/master/tenders/hvt/hvt_gdb_kvm_x86_64.c. Accessed: 2024-07-07.

[23] The Unikraft developers. Debugging a Unikernel – linuxu. https://unikraft.org/docs/internals/debugging#linuxu, 2022. Accessed: 2024-07-07.

[24] Unikraft developers. Unikraft libramfs library. https://github.com/unikraft/unikraft/tree/staging/lib/ramfs. Accessed: 2024-07-07.

[25] Unikraft developers. Unikraft ukalloc library. https://github.com/unikraft/unikraft/tree/staging/lib/ukalloc. Accessed: 2024-07-07.

[26] Unikraft developers. Unikraft uksignal library. https://github.com/unikraft/unikraft/tree/staging/lib/uksignal. Accessed: 2024-07-07.

[27] Unikraft developers. Unikraft vfscore library. https://github.com/unikraft/unikraft/tree/staging/lib/vfscore. Accessed: 2024-07-07.

[28] Bob Duncan, Andreas Happe, and Alfred Bratterud. Enterprise IoT Security and Scalability: How Unikernels Can Improve the Status Quo. In *Proceedings of the 9th International Conference on Utility and Cloud Computing*. Association for Computing Machinery, 2016.

[29] F5 Inc. Advanced Load Balancer, Web Server, & Reverse Proxy - NGINX. https://www.nginx.com/. Accessed: 2024-07-07.

[30] F5 Inc. Controlling NGINX Processes at Runtime. https://docs.nginx.com/nginx/admin-guide/basic-functionality/runtime-control/. Accessed: 2024-07-07.

[31] Henrique Fingler, Amogh Akshintala, and Christopher J. Rossbach. USETL: Unikernels for Serverless Extract Transform and Load Why Should You Settle For Less? In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*. Association for Computing Machinery, 2019.

[32] Free Software Foundation. GDB: The GNU Project Debugger. https://www.sourceware.org/gdb/. Accessed: 2024-07-07.

[33] Free Software Foundation. Remote Stub (Debugging with GDB). https://sourceware.org/gdb/onlinedocs/gdb/Remote-Stub.html. Accessed: 2024-07-07.

[34] Free Software Foundation. lwIP - A Lightweight TCP/IP stack - Summary. https://savannah.nongnu.org/projects/lwip/. Accessed: 2024-07-07.

[35] Mike Frysingerj. Function Tracer Design. https://docs.kernel.org/trace/ftrace-design.html. Accessed: 2024-07-07.

[36] Yangchun Fu, Junyuan Zeng, and Zhiqiang Lin. HYPERSHELL: A Practical Hypervisor Layer Guest OS Shell for Automated In-VM Management. In *Proceedings of the 2014 USENIX Annual Technical Conference*. USENIX Association, 2014.

[37] Tal Garfinkel, Mendel Rosenblum, et al. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the 2003 Network and Distributed System Security Symposium*. Internet Society, 2003.

[38] gcc developers. x86 Options (Using the GNU Compiler Collection (GCC)). https://gcc.gnu.org/onlinedocs/gcc/x86-Options.html#index-mnop-mcount. Accessed: 2024-07-07.

[39] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, 2019.

[40] Will Glozer. wg/wrk: Modern HTTP benchmarking tool. https://github.com/wg/wrk. Accessed: 2024-07-07.

[41] Google. Guest environment | Compute Engine Documentation | Google Cloud. https://cloud.google.com/compute/docs/images/guest-environment. Accessed: 2024-07-07.

[42] Brendan Gregg. Unikernel Profiling: Flame Graphs from dom0. https://www.brendangregg.com/blog/2016-01-27/unikernel-profiling-from-dom0.html, 2016. Accessed: 2024-07-07.

[43] Wassim Haddad, Heikki Mahkonen, and Ravi Manghirmalani. NFV Platforms with MirageOS Unikernels. http://unikernel.org/blog/2016/unikernel-nfv-platform, 2016. Accessed: 2024-07-07.

[44] Pengzhan Hao, Yongshu Bai, Xin Zhang, and Yifan Zhang. Edgecourier: An Edge-Hosted Personal Service for Low-Bandwidth Document Synchronization in Mobile Cloud Storage Services. In *Proceedings of the 2nd ACM/IEEE Symposium on Edge Computing*. Association for Computing Machinery, 2017.

[45] 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 *Proceedings of the 2019 USENIX Annual Technical Conference*. USENIX Association, 2019.

[46] Gernot Heiser and Kevin Elphinstone. L4 Microkernels: The Lessons from 20 Years of Research and Deployment. *ACM Trans. Comput. Syst.*, 34(1), apr 2016.

[47] 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), 2006.

[48] Intel. Intel© 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Order Number: 253668-078US – Chapter 4.6.2 Protection Keys, 2022. Accessed: 2024-07-07.

[49] Intel. Intel© 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Order Number: 253668-078US – Chapter 19 Performance Monitoring, 2022. Accessed: 2024-07-07.

[50] Intel. Intel® Trust Domain Extensions (Intel TDX). https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html, [n. d.].

[51] Seung Hyub Jeon, Seung-Jun Cha, Ramneek, Yeon Jeong Jeong, Jin Mee Kim, and Sungin Jung. Azalea-Unikernel: Unikernel into Multi-kernel Operating System for Manycore Systems. In *Proceedings of the 7th IEEE International Conference on Cloud Computing Technology and Science*. IEEE Computer Society, 2018.

[52] Yuzhuo Jing and Peng Huang. Operating System Support for Safe and Efficient Auxiliary Execution. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2022.
[53] Justin Cormack. justincormack/frankenlibc: Tools for Running Rump Unikernels in Userspace. https://github.com/justincormack/frankenlibc. Accessed: 2024-07-07.
[54] Linux kernel developers. BPF Documentation. https://docs.kernel.org/bpf/index.html. Accessed: 2024-07-07.
[55] Linux kernel developers. Cgruop-v1 Documentation. https://www.kernel.org/doc/Documentation/cgroup-v1/. Accessed: 2024-07-07.
[56] Linux kernel developers. eBPF verifier. https://docs.kernel.org/bpf/verifier.html. Accessed: 2024-07-07.
[57] Linux kernel developers. elf(5) - Linux manual page. https://man7.org/linux/man-pages/man5/elf.5.html. Accessed: 2024-07-07.
[58] Linux kernel developers. fuse(4) — Linux manual page. https://man7.org/linux/man-pages/man4/fuse.4.html. Accessed: 2024-07-07.
[59] Linux kernel developers. ld.so(8) — Linux manual page. https://man7.org/linux/man-pages/man8/ld.so.8.html. Accessed: 2024-07-07.
[60] Linux kernel developers. libc(7) — Linux manual page. https://man7.org/linux/man-pages/man7/libc.7.html. Accessed: 2024-07-07.
[61] Linux kernel developers. socket(7) – Linux manual page. https://man7.org/linux/man-pages/man7/socket.7.html. Accessed: 2024-07-07.
[62] Linux kernel developers. syscalls(2) — Linux manual page . https://man7.org/linux/man-pages/man2/syscalls.2.html. Accessed: 2024-07-07.
[63] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: the Linux Virtual Machine Monitor. In *Proceedings of the 2007 Linux Symposium*, 2007.
[64] 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 *Proceedings of the 2014 USENIX Annual Technical Conference*. USENIX Association, 2014.
[65] 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*, SOSP '09. Association for Computing Machinery, 2009.
[66] Michał Król and Ioannis Psaras. NFaaS: Named Function as a Service. In *Proceedings of the 4th ACM Conference on Information-Centric Networking*. Association for Computing Machinery, 2017.
[67] 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. Unikraft: Fast, Specialized Unikernels the Easy Way. In *Proceedings of the 16th European Conference on Computer Systems*. Association for Computing Machinery, 2021.
[68] Simon Kuenzer, Anton Ivanov, Filipe Manco, Jose Mendes, Yuri Volchkov, Florian Schmidt, Kenichi Yasukata, Michio Honda, and Felipe Huici. Unikernels Everywhere: The Case for Elastic CDNs. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. Association for Computing Machinery, 2017.
[69] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. A Linux in Unikernel Clothing. In *Proceedings of the 15th European Conference on Computer Systems*. Association for Computing Machinery, 2020.
[70] Tytus Kurek. Unikernel Network Functions: A Journey Beyond the Containers. *IEEE Communications Magazine*, 57(12), 2019.
[71] Stefan Lankes, Jonathan Klimt, Jens Breitbart, and Simon Pickartz. RustyHermit: A Scalable, Rust-Based Virtual Execution Environment. In *Proceedings of the 2020 International Conference on High Performance Computing*. Springer International Publishing, 2020.
[72] Stefan Lankes, Simon Pickartz, and Jens Breitbart. HermitCore: A Unikernel for Extreme Scale Computing. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*. Association for Computing Machinery, 2016.
[73] Stefan Lankes, Simon Pickartz, and Jens Breitbart. A Low Noise Unikernel for Extrem-Scale Systems. In *Proceedings of the 2017 Architecture of Computing Systems*. Springer International Publishing, 2017.
[74] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. FlexOS: Towards Flexible OS Isolation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 2022.
[75] Guanyu Li, Dong Du, and Yubin Xia. Iso-UniK: Lightweight Multi-process Unikernel through Memory Protection Keys. *Cybersecurity*, 3(1), 2020.
[76] Arm Limited. ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition – Domains. https://developer.arm.com/documentation/ddi0406/b/System-Level-Architecture/Virtual-Memory-System-Architecture--VMSA-/Memory-access-control/Domains. Accessed: 2024-07-07.
[77] Costin Lupu, Andrei Albişoru, Radu Nichita, Doru-Florin Blânzeanu, Mihai Pogonaru, Rlăzvan Deaconescu, and Costin Raiciu. Nephele: Extending Virtualization Environments for Cloning Unikernel-based VMs. In *Proceedings of the 18th European Conference on Computer Systems*. Association for Computing Machinery, 2023.
[78] 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 *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 2015.
[79] 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), 2013.
[80] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles*. Association for Computing Machinery, 2017.
[81] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. ClickOS and the Art of Network Function Virtualization. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 2014.
[82] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the 1993 USENIX Winter*. USENIX Association, 1993.
[83] A K M Fazla Mehrab, Ruslan Nikolaev, and Binoy Ravindran. Kite: Lightweight Critical Service Domains. In *Proceedings of the 17th European Conference on Computer Systems*. Association for Computing Machinery, 2022.
[84] Roberto Morabito, Vittorio Cozzolino, Aaron Yi Ding, Nicklas Beijar, and Jorg Ott. Consolidate IoT Edge Computing with Lightweight Virtualization. *IEEE Network*, 32(1), 2018.
[85] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. A Binary-Compatible Unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. Association for Computing Machinery, 2019.
[86] OASIS Open. Virtual I/O Device (VIRTIO) Version 1.1. https://docs.oasis-open.org/virtio/virtio/v1.1/csprd01/virtio-v1.1-csprd01.html, 2018. Accessed: 2024-07-07.
[87] OpenSSH developers. OpenSSH. https://www.openssh.com/. Accessed: 2024-07-07.
[88] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *Proceedings of the 2019 USENIX Annual Technical Conference*. USENIX Association, 2019.
[89] Per Buer. Unikernels Aren't Dead, They're Just Not Containers. https://www.infoq.com/presentations/unikernels-includeos/, 2019. Accessed: 2024-07-07.
[90] Rob Pike, David L. Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Comput. Syst.*, 8(2), 1995.
[91] Ali Raza, Thomas Unger, Matthew Boyd, Eric Munson, Parul Sohal, Ulrich Drepper, Richard Jones, Daniel Bristot de Oliveira, Larry Woodman, Renato Mancuso, Jonathan Appavoo, and Orran Krieger. Unikernel Linux (UKL). In *Proceedings of the 18th European Conference on Computer Systems*. Association for Computing Machinery, 2023.
[92] Redis Ltd. Redis. https://redis.io/. Accessed: 2024-07-07.
[93] Redis Ltd. Redis Persistence. https://redis.io/docs/manual/persistence/. Accessed: 2024-07-07.
[94] Rusty Russell. virtio: Towards a De-facto Standard for Virtual I/O Devices. *ACM SIGOPS Operating Systems Review*, 42(5), 2008.
[95] Peter Jay Salzman, Michael Burian, Ori Pomerantz, Bob Mottram, and Jim Huang. The Linux Kernel Module Programming Guide. https://sysprog21.github.io/lkmpg/. Accessed: 2024-07-07.
[96] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. 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*. Association for Computing Machinery, 2021.
[97] Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, and Jonathan Appavoo. EbbRT: A Framework for Building Per-Application Library Operating Systems. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2016.
[98] Florian Schmidt. Uniprof: A Unikernel Stack Profiler. In *Proceedings of the SIGCOMM 2017 Posters and Demos*. Association for Computing Machinery, 2017.
[99] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain Keys – Efficient in-Process Isolation for RISC-V and X86. In *Proceedings of the 29th USENIX Conference on Security Symposium*. USENIX Association, 2020.
[100] SQLite developers. SQLite Home Page. https://www.sqlite.org/index.html. Accessed: 2024-07-07.

[101] SQLite developers. Using the SQLite Online Backup API. https://www.sqlite.org/backup.html. Accessed: 2024-07-07.

[102] 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*. Association for Computing Machinery, 2020.

[103] Joshua Talbot, Przemek Pikula, Craig Sweetmore, Samuel Rowe, Hanan Hindy, Christos Tachtatzis, Robert Atkinson, and Xavier Bellekens. A Security Perspective on Unikernels. In *Proceedings of the 2020 International Conference on Cyber Security and Protection of Digital Services*. IEEE Computer Society, 2020.

[104] Bo Tan, Haikun Liu, Jia Rao, Xiaofei Liao, Hai Jin, and Yu Zhang. Towards Lightweight Serverless Computing via Unikernel as a Function. In *Proceedings of the 28th IEEE/ACM International Symposium on Quality of Service*, 2020.

[105] Jörg Thalheim, Peter Okelmann, Harshavardhan Unnibhavi, Redha Gouicem, and Pramod Bhatotia. VMSH: Hypervisor-Agnostic Guest Overlays for VMs. In *Proceedings of the 17th European Conference on Computer Systems*. Association for Computing Machinery, 2022.

[106] ubpf developers. iovisor/ubpf: Userspace eBPF VM. https://github.com/iovisor/ubpf. Accessed: 2024-07-07.

[107] Unikraft developers. Unikraft Release: v0.9.0 Hyperion. https://github.com/unikraft/unikraft/tree/RELEASE-0.9.0. Accessed: 2024-07-07.

[108] Unikraft developers. unikraft/lib-nginx: Unikraft port of NGINX. https://github.com/unikraft/lib-nginx. Accessed: 2024-07-07.

[109] Unikraft developers. unikraft/lib-redis: Unikraft port of Redis in-memory data structure store. https://github.com/unikraft/lib-redis. Accessed: 2024-07-07.

[110] Unikraft developers. unikraft/lib-sqlite: Unikraft port of SQLite. https://github.com/unikraft/lib-sqlite. Accessed: 2024-07-07.

[111] 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 *Proceedings of the 28th USENIX Security Symposium*. USENIX Association, 2019.

[112] Polychronis Valsamas, Sotiris Skaperas, and Lefteris Mamatas. Elastic Content Distribution Based on Unikernels and Change-Point Analysis. In *Proceedings of the 24th European Wireless Conference*. IEEE Computer Society, 2018.

[113] virtiofs developers. virtiofs - Shared File System for Virtual Machines. https://virtio-fs.gitlab.io/. Accessed: 2024-07-07.

[114] Fedora Project Wiki. Features/VirtioSerial - Fedora Project Wiki. https://fedoraproject.org/wiki/Features/VirtioSerial. Accessed: 2024-07-07.

[115] QEMU wiki. Documentation/9psetup - QEMU. https://wiki.qemu.org/Documentation/9psetup. Accessed: 2024-07-07.

[116] QEMU wiki. Features/gdbstub - QEMU. https://wiki.qemu.org/Features/gdbstub. Accessed: 2024-07-07.

[117] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. Unikernels as Processes. In *Proceedings of the ACM Symposium on Cloud Computing 2018*. Association for Computing Machinery, 2018.

[118] Song Wu, Chao Mei, Hai Jin, and Duoqiang Wang. Android Unikernel: Gearing Mobile Code Offloading Towards Edge Computing. *Future Generation Computer Systems*, 86, 2018.

[119] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfen Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. KylinX: A Dynamic Library Operating System for Simplified and Efficient Cloud Virtualization. In *Proceedings of the 2018 USENIX Annual Technical Conference*. USENIX Association, 2018.