POLITECNICO DI TORINO
Repository ISTITUZIONALE

SURE: Secure Unikernels Make Serverless Computing Rapid and Efficient

Original
SURE: Secure Unikernels Make Serverless Computing Rapid and Efficient / Parola, F., Qi, S., Narappa, A.B., Ramakrishnan, K.K., Risso, F.. - ELETTRONICO. - (2024), pp. 668-688. (2024 ACM Symposium on Cloud Computing Redmond, WA (USA) November 20–22, 2024) [10.1145/3698038.3698558].

Availability:
This version is available at: 11583/2995315 since: 2024-12-13T09:43:41Z

Publisher:
Association for Computing Machinery

Published
DOI:10.1145/3698038.3698558

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

25 November 2026



SURE: Secure Unikernels Make Serverless Computing Rapid and Efficient

Federico Parola[1], Shixiong Qi[2*], Anvaya B. Narappa[2], K. K. Ramakrishnan[2], Fulvio Risso[1]
[1]Politecnico di Torino, [2]University of California, Riverside

ABSTRACT

Current serverless platforms introduce non-trivial overheads when chaining and orchestrating loosely-coupled microservices. Containerized function runtimes are also constrained by insufficient isolation and excessive startup time. This motivates our exploration of a more efficient, secure, and rapid serverless design. We describe SURE, a unikernel-based serverless framework for fast function startup, equipped with a high-performance and secure data plane. SURE's data plane supports distributed zero-copy communication via the seamless interaction between zero-copy protocol stack (Z-stack) and local shared memory processing. To establish a lightweight service mesh, SURE uses library-based sidecars instead of individual userspace sidecars. We leverage Intel's Memory Protection Keys (MPK) as a lightweight capability to ensure safe access to the shared memory data plane. It also isolates the Trusted Computing Base (TCB) components in SURE's function runtime (e.g., library-based sidecar, scheduler, etc) from untrusted user code, while preserving the efficient single-address-space nature of unikernels. In particular, SURE prevents unintended privilege escalation involving MPK with an enhanced TCB. These combined efforts create a more secure and robust data plane while improving throughput up to 79X over Knative, a representative open-source serverless platform.

CCS CONCEPTS

• **Networks** → **Cloud computing**; • **Computer systems organization** → **Cloud computing**; • **Security and privacy** → **Virtualization and security**.

KEYWORDS

Serverless, Unikernel, Shared memory, MPK, Isolation

*Now at University of Kentucky.



This work is licensed under a Creative Commons Attribution-ShareAlike International 4.0 License.
SoCC '24, November 20–22, 2024, Redmond, WA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1286-9/24/11.
https://doi.org/10.1145/3698038.3698558

ACM Reference Format:
F. Parola, S. Qi, A. B. Narappa, K. K. Ramakrishnan, F. Risso. 2024. SURE: Secure Unikernels Make Serverless Computing Rapid and Efficient. In *ACM Symposium on Cloud Computing (SoCC '24), November 20–22, 2024, Redmond, WA, USA.* ACM, New York, NY, USA, 21 pages. https://doi.org/10.1145/3698038.3698558

1 INTRODUCTION

Cloud computing services are evolving to composable, *loosely-coupled* microservices [47, 68], where each microservice is more quickly developed, tested, and deployed independently. The request from an external client triggers a series of invocations to a sequence of inter-dependent microservices, potentially depicted as a *call graph* [68]. Serverless computing, often referred to as Function-as-a-Service (FaaS [22]), is a natural fit for loosely coupled microservices because of event-driven execution, fine-grained billing ("pay-as-you-go") and the unlimited elasticity provided by a serverless infrastructure, thus simplifying the management of the application [7, 50, 52, 54, 65, 86]. Furthermore, with serverless computing, loosely-coupled microservices can be organized into a "function chain", following their call graph dependencies [53, 68]. This helps transition production-level microservice workloads to serverless platforms [34].

One of the main challenges in serverless computing is the cold-start problem, where the initial invocation of a function incurs significant latency as the runtime environment is initialized [37, 66, 91, 104]. This latency can adversely impact the responsiveness of serverless applications. Achieving fast startup of serverless functions is crucial for mitigating the cold-start issue. In this context, ***unikernels*** are particularly attractive for serverless computing due to their ability to achieve fast startup times for serverless functions [24, 30, 74, 95]. Unlike traditional full-size VMs or containers, unikernels are designed to be lightweight, with a single address space with only the necessary libraries and drivers [57, 58, 71, 72]. This streamlined architecture allows unikernels to boot rapidly, often in milliseconds, making them ideal for scenarios requiring rapid autoscaling and responsive scaling to handle increases in the workload [30, 74]. Further, the entire software stack of a unikernel can be specialized, typically resulting in a much smaller Trusted Computing Base (TCB) and potentially fewer vulnerabilities [108]. These desirable characteristics make unikernels well suited for secure, lightweight deployment of serverless functions,

compared to widely used containers or other heavyweight virtualization approaches [95]. Additionally, compatibility with existing applications is managed by the introduction of a compatibility layer in the unikernel [3]. Popular container orchestration platforms, such as Kubernetes [10], have expanded to interface with unikernels [19], making them production-ready, such as with NanoVMs [14].

State of the Landscape. Despite serverless computing offering many positive capabilities, current platforms have high latency (millisecond-scale), and overheads [50]. Kernel-based inter-function networking and a heavyweight service mesh using a loosely-coupled userspace sidecar are significant contributors [33, 82, 110]. But, considering the "killer microseconds" [27] that serverless computing needs to address, a naive unikernel-based solution is not yet ready to support loosely coupled microservices due to the following constraints: ***(1)*** Unikernel-based serverless environments [95] face the same problem of slow inter-function networking as with containerized environments. While shared memory processing helps with optimizing containerized environments [82, 106] such designs only consider intra-node data plane optimizations and don't fully address the required isolation across function chains. Cross-node communication still uses kernel-based networking [82]. ***(2)*** The performance and resource consumption drawbacks of using an individual sidecar for service mesh functionality persist. eBPF-based acceleration [82] provides tremendous improvements over individual sidecars, and its event-driven execution perfectly aligns with the idea of serverless computing. However, corresponding eBPF functionality (i.e., eBPF hooks) is not yet available in unikernel-based environments. An eBPF-based sidecar also faces limitations in achieving full (L7) payload visibility [93] due to the constraints imposed by the eBPF verifier [98] for the sake of safe in-kernel execution. ***(3)*** Intra-unikernel isolation remains a concern. Unikernel's single-address-space design eliminates the kernel-userspace boundary, making it lightweight and fast for function execution compared to containers [39, 95]. However, there is no isolation between user code and the necessary LibOS modules (e.g., scheduler) and data structures (e.g., thread contexts).

This paper describes a unikernel-based serverless computing framework that strives to operate in the best possible region of the design space to support rapid function startup and low latency networking between functions while maintaining isolation. We call our work **SURE**, for **S**ecure **U**nikernels that are **R**apid and **E**fficient. SURE deploys a serverless function as a unikernel-based VM (called a SURE VM), which offers substantial agility through fast function startup and inherently adapts the VM-based isolation at the granularity of independent functions. SURE facilitates low-latency and high-performance inter-function networking through *distributed*, zero-copy communication rather than kernel-based networking. It first utilizes shared memory processing for zero-copy intra-node communication, as demonstrated in [82] and [106]. Going beyond those designs, we introduce a zero-copy protocol stack (called Z-stack) to facilitate *distributed* zero-copy communication *across nodes*. Z-stack has a full-fledged FreeBSD TCP/IP stack implementation that works in userspace to mitigate kernel-related overheads. Importantly, Z-stack seamlessly interfaces with the local shared memory data plane, augmenting zero-copy communication without being limited to a single node. We introduce a per-node SURE gateway for coordinating inter-node zero-copy communication. The SURE gateway works with Z-stack to provide consolidated protocol processing for all functions co-located within the same node (see §4.1).

SURE fully takes advantage of the LibOS-based design of unikernels by deploying the sidecar as a library linked into the function code within the unikernel. The unikernel's single-address-space design eliminates boundary crossings between kernel and userspace. It simplifies data exchange between the library-based sidecar and user code by using *internal* function calls. This overcomes the shortcomings of an individual userspace sidecar. SURE introduces various sidecar hooks, which allow for the attachment of eBPF-like event-driven sidecars in a unikernel-based runtime. But unlike eBPF, SURE's library-based sidecar allows us to implement complex L7 sidecar functionality with full payload visibility to enable a full-functional service mesh (see §4.2).

SURE pays particular attention to security vulnerabilities that may arise from the use of shared memory processing and single-address-space unikernels (details in §3.3). Going beyond existing solutions (*e.g.,* [82]) that use group-based security domains to enable coarse-grained isolation at the level of shared memory pools, SURE offers more fine-grained isolation at the level of memory pages. This allows us to manage the ownership of shared memory pages for each individual function, which can prevent a misbehaving function from inadvertently manipulating shared memory pages owned by other functions, even if they are in the same security domain. Additionally, we isolate the library-based sidecar in SURE, which is part of the serverless infrastructure and contains sensitive data, such as sidecar statistics, from user code that is typically untrusted in a cloud environment. To achieve these design goals without compromising its high-performance data plane, SURE uses Intel's Memory Protection Keys (MPK) [13] for its lightweight memory isolation. SURE provides an MPK-based "call gate" abstraction for user code to safely interact with memory pages belonging to protected unikernel modules (e.g., sidecar). The call gate also uses MPK to selectively grant access to shared memory pages (see §5.2).

However, having a single-address-space unikernel still means that unprivileged user code is capable of modifying and causing potential MPK privilege escalation and gaining disallowed access to protected pages, which was overlooked in previous work [61, 63, 88]. The root cause lies in not protecting the unikernel TCB components (including the scheduler, page table management, and interrupt service handling) that have control of access privileges to protected memory pages. These components have direct access to sensitive data structures in the single address space. SURE protects the enhanced unikernel TCB to prevent such exploits (see §5.3).

Summary of Contributions.

(1) SURE uses event-driven, shared memory processing while retaining the philosophy of serverless computing. SURE adds key extensions, including connection management and backpressure for lossless communication between functions.

(2) SURE enhances existing designs [82, 106] by extending zero-copy communication to go across nodes and integrating zero-copy TCP/IP processing with the local shared memory data plane for intra-node communication.

(3) Our use of a library-based sidecar fully exploits the single-address-space benefits of unikernels, resulting in more than ***100***× CPU cycle savings and 16× performance improvement compared to the individual userspace sidecar.

(4) SURE's MPK-based call gates manage the ownership of each shared memory page, which is more secure than the descriptor-based ownership management used in [82], but retains the high performance of shared memory processing.

(5) We identify and overcome vulnerabilities of a unikernel TCB that may cause privilege escalation of MPK. SURE (adapted from Unikraft [58]) does so by carefully protecting and controlling access to each of the distinct TCB components.

SURE is publicly available at https://github.com/ucr-serverless/sure.git.

2 BACKGROUND AND MOTIVATION

We start by examining serverless architectures, understanding the challenges that remain, and discuss related work.

2.1 The tradeoff between isolation and agility in serverless computing

Isolating serverless functions: Serverless computing is supported in an open, shared cloud computing environment. But, without proper isolation there is a risk of unwanted access of sensitive data as well as resource contention. Containers offer lightweight virtualization with relatively rapid startup. However, they are less secure due to their state being managed by the host OS [25, 42, 45, 73, 103]. This has provided incentives to commercial serverless providers to revert to virtualization similar to VMs (individual application VMs have a separate OS kernel) to enhance their platform's security [6, 24]. But running a serverless function as a VM installed with a full-fledged guest OS can lead to unacceptable delays in starting the function, which can adversely affect the responsiveness of serverless applications [57, 95].

Unikernel can make serverless functions agile: Using the concept of a LibOS [81], we can rebuild a traditional OS into libraries and bind the application with only the required OS libraries, all running in a single address space (also called a unikernel [71]). This customization makes unikernels lightweight, with faster startup than containers even when working with VM-based virtualization (as we evaluate in §6.2), while offering stronger isolation than containers [57]. This design also eliminates the kernel-userspace boundary crossing within the unikernel, further reducing the runtime overhead compared to a conventional VM [57].

Enhancing Protection in Unikernels. Unikernel-based applications heavily rely on platform-provided OS components such as the scheduler, page table management, and interrupt service routine handling (more details in §3.5 and §5.3). With a single-address-space unikernel, the serverless cloud service provider would need to trust the integrity of these components inside the unikernel. To achieve high performance, we also include sensitive data such as the library-based sidecar and shared memory within the unikernel. We protect this sensitive data and code using MPK, by including them as well as the OS components as part of the TCB.

> *Design Implication#1*: Although a unikernel-based VM runtime enables rapid function startup, which is desired for serverless computing, and provides strict isolation between functions, we still need to enhance intra-unikernel isolation to protect TCB components from user code.

2.2 Inter-function networking and service mesh in serverless computing

Serverless support for loosely-coupled microservices. As depicted in Fig. 1, serverless computing has three important building blocks in the infrastructure to support loosely-coupled microservices that are organized as a "function chain": (1) *Inter-function networking* for communication between decoupled functions. Essential components include a virtual switch (vSwitch) for L2 forwarding; a network protocol stack (e.g., TCP/IP) for handling application layer messages; and virtual device interfaces (vDevices) to interface between the virtualized functions and the vSwitch; (2) A *service mesh* (*e.g.,* Istio [1]) that transparently facilitates orchestration (observability, traffic management, and access control) of serverless functions in distributed environments by attaching an *individual* sidecar to the serverless function [110]; (3) A *virtualized function runtime* (sandbox) at the individual microservice (function) level that is needed for fine-grained isolation in public clouds.

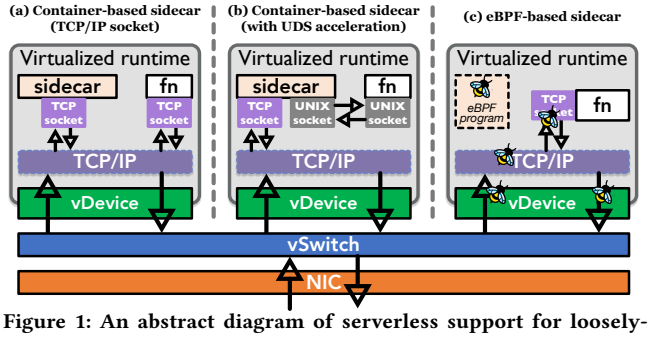

Figure 1: An abstract diagram of serverless support for loosely-coupled microservices. We list existing sidecar designs: (a) container-based sidecar using TCP/IP socket [1, 110], (b) container-based sidecar using UDS acceleration [110], (c) eBPF-based sidecar [41, 82].

2.2.1 Cost of kernel-based inter-function networking. Most of the data plane overheads for function chains: *e.g.,* data copies, context switches, interrupts, protocol processing, and serialization/deserialization, come from kernel-based networking [31, 82]. There is duplicate processing at different layers, even if functions are co-located on the same node [82].

Existing solution: shared memory processing. [82, 106] design a high-performance data plane for serverless function chaining using shared memory enabling *zero-copy* communication between functions without incurring any kernel networking overheads.

Zero-copy networking is limited to intra-node communication. The *zero-copy* shared memory communication in [82, 106], is limited to a single node. For cross-node communication or when functions need to interact with external clients/servers, [82, 106] still depend on kernel-based networking. Both [51] and [32] suggest maximizing locality in workload placement to reduce cross-node communication which is not always feasible. Production workloads may contain complex and large-scale microservice call graphs. E.g., Alibaba reports a trace analysis showing that more than 3000 microservices in their workload have interdependencies [68]. Functions can also be resource-intensive (as in data analytic jobs [51]) and need to be spread across multiple nodes.

> *Design Implication#2*: *Zero-copy* networking should be extended for inter-node communication. It is important to reduce kernel-related overheads, and still achieve high performance for cross-node communication.

Shared memory processing is considered not safe. Shared memory processing can become a potential conduit for data leakage and corruption [36, 101, 105]. [106] assumes functions from different users are never co-located, thereby, functions on the same node are from the same user and can trust each other. [82] considers group-based separation, assuming the same user's functions trust each other. Different users have separate memory pools, even in the same node. But [82] lacks memory-level management of access privileges within the function chain to protect against buggy code.

> *Design Implication#3*: Need secure shared memory processing while retaining its high performance for function chains, carefully granting access privileges at the individual message buffer level. This restricts access to a shared memory pool combined with group-based separation. Communication channels established through shared memory must be secured to prevent manipulation by an unauthorized entity (i.e., an unintended recipient).

2.2.2 Cost of individual userspace sidecars. Existing service mesh designs deploy a sidecar as an individual component (*e.g.,* container), independent of the user function. Communication between the individual sidecar and the user function needs to traverse the TCP/IP stack [110] (Fig. 1 (a)), incurring unnecessary networking overheads in the data plane [33, 82, 110]. Acceleration includes using Unix domain sockets [21] to redirect the payload between the user function sockets and the individual sidecar (Fig. 1 (b)), bypassing protocol processing. But, running an individual sidecar still incurs data copy and serialization/deserialization overheads.

eBPF streamlines the Service Mesh. eBPF-based acceleration [41] has been used in containerized environments to provide the service mesh functionality [2, 82] instead of the individual sidecar (Fig. 1 (c)). eBPF-based sidecars are developed as run-to-completion programs attached to in-kernel eBPF hooks (XDP [46], TC [67], and SOCK_MSG [82]). Executed in the kernel, the eBPF-based sidecar avoids the frequent userspace-kernel boundary crossing and duplicate overhead of inter-container communication with the individual userspace sidecar. The execution of the eBPF-based sidecar is triggered upon events, which makes it particularly suitable for event-driven serverless computing [82].

eBPF-based sidecar is not suitable for unikernel environments. But, eBPF cannot be fully utilized in unikernel environments, due to the lack of certain eBPF hooks (e.g., SOCK_MSG), limiting its use. An eBPF-based sidecar doesn't provide the full (L7) payload visibility and corresponding L7 functionality that container-based sidecars provide, which is critical for serverless computing [93]. The constrained programming model of eBPF is also a deterrent [109].

> *Design Implication#4*: An ideal sidecar design needs to go beyond by providing full L7 visibility and not being constrained by the eBPF bytecode limitations. It should still have the significant benefits of an eBPF-based sidecar, i.e., event-driven execution and avoiding unnecessary kernel-userspace boundary crossings, to overcome the shortcomings of the individual userspace sidecar.

2.3 Related work

As discussed, prior work [82, 106] is limited with regard to secure shared memory processing and enable high-performance inter-node communication. An eBPF-based sidecar, as in [82],

is also not viable for an unikernel-based serverless environment. [30, 39, 95] use unikernels as the runtime for the fast startup of serverless functions. However, they lack support for zero-copy communication and don't have a lightweight service mesh, making them less suitable for supporting decoupled microservices. The lack of intra-unikernel isolation in [30, 39, 95] is also a concern.

Unikernel/LibOS Virtualization: Unikernels are very suitable for single-purpose microservices that need to run isolated, secure tasks with minimal overhead [95]. But they may not be ideal for complex serverless workloads requiring dynamic, multi-process environments or use extensive system services. Jonas et al. [52] observed a growing demand for fine-grained isolation in serverless computing and identified unikernels as a potential solution to minimize the attack surface. Several past works have optimized different aspects of unikernels, including system development kits [57–59, 81], multi-process support [63, 69, 96], fast startup [70, 73, 100], TCP proxies for connection acceleration [92]. SURE can take advantage of these unikernels' startup optimizations [70, 73, 100] to reduce the cold-start penalty of serverless computing. CubicleOS [88] and FlexOS [61] offer intra-unikernel isolation using MPK. However, they are not focused on data plane optimization and lack service mesh support, crucial for serverless computing.

High-performance data plane: Many high-performance data plane designs for disaggregated cloud applications have been proposed [28, 29, 44, 48, 62, 64, 70, 82, 102, 107]. Compared to SURE, they lack necessary isolation or memory protection on the data plane [82, 107], or do not support lightweight manageability [28, 44, 62, 70, 102], making them not ideal for serverless computing. There are multiple high-performance TCP/IP protocol stack implementations [5, 49, 56, 102, 107]: StackMap [102] offers zero-copy TCP/IP processing by integrating a full-fledged Linux network stack with the interrupt-driven packet handling of netmap [84]. This could be less efficient under high loads compared to a full-fledged TCP/IP stack that uses DPDK's polling-based packet handling [5, 56]; Demikernel [107] supports zero-copy processing through its TCP/IP stack. However, it is not full-fledged. In constrast, SURE's Z-stack combines zero-copy TCP/IP processing with the full-fledged FreeBSD TCP/IP stack and low-latency DPDK-based packet handling.

MPK-based isolation: Multiple proposals study the application and optimization of Intel's MPK [63, 79, 89, 90, 94, 97]. Jenny [89] filters MPK-related syscalls to prevent unauthorized changes to the MPK key. ERIM [97] enforces binary inspection and rewriting to prevent misuse of MPK. libmpk [79] overcomes the limit of 16 MPK keys by carefully recycling and redistributing keys. EPK [43] extends the supported number of MPK-protected memory domains by using a hardware-based protection key extension – extended page table (EPT). This can enable MPK-protected shared memory communication between a large number of domains (up to 7680) [43]. These works are complementary to SURE. Other efforts focus on protecting the unikernel (e.g., OCaml in Mirage [72]) and shared memory processing (Rust in RedLeaf [78]) at the language level. They can be used as further enhancement to SURE's unikernel runtime and shared memory data plane.

3 OVERVIEW OF SURE

3.1 System architecture of SURE

Fig. 2 shows the overall architecture of SURE, including the following core building blocks: **(1) Unikernel-based function runtime.** Each function runs as an individual unikernel-based VM in SURE (i. e., the SURE VM in Fig. 2) to strike an ideal balance between providing isolation and being lightweight. However, SURE also supports running multiple functions within the same unikernel under certain conditions, e.g., when these functions mutually trust and work together within the same virtualized runtime. The hypervisor on each worker node is in charge of controlling the life cycle of SURE VMs: creating and destroying VMs, allocating addresses, and initializing shared memory. **(2) A distributed, zero-copy data plane.** When functions are co-located on the same node, we leverage shared memory processing for zero-copy intra-node communication (§4.1). For cross-node traffic, we develop a high-performance, *zero-copy* user-space TCP/IP stack, Z-stack (§4.1) — a zero-copy enhancement to the existing DPDK F-stack [5]. Z-stack seamlessly interfaces with SURE's *zero-copy* intra-node data plane. **(3) Consolidated protocol processing.** To avoid duplicate processing between functions, we consolidate all of the protocol processing in a single per-node SURE gateway (§4.1). **(4) Lightweight library-based sidecar.** As shown in Fig. 2, SURE moves each sidecar to be a library within the SURE VM (§4.2). The message exchanges between the sidecar and the functions are simple internal function calls, entirely eliminating a number of data plane overheads of the individual sidecar deployment model. **(5) MPK-based isolation in the shared address space.** We provide an MPK-based call gate abstraction in SURE's

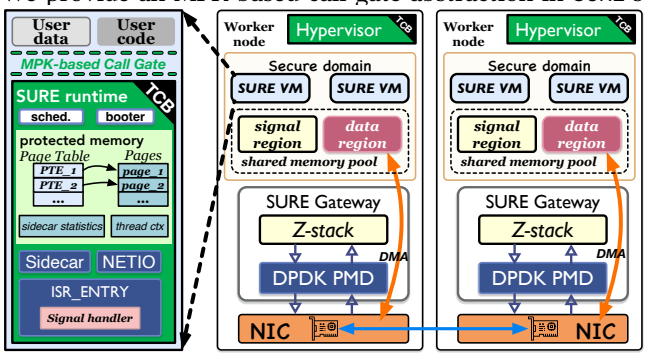

Figure 2: The overall architecture of SURE. Note that we only show a single security domain here.

unikernel runtime to enable untrusted user code to interact safely with protected memory pages in the shared memory data plane and also in the TCB components. We also enhance the unikernel TCB in SURE to prevent privilege escalation of MPK (see §3.5).

3.2 SURE's trust model

SURE assumes a one-way trust model typically considered in a public serverless cloud: Users trust the serverless infrastructure (*i.e.,* SURE VM) provided by SURE. However, SURE does not trust users, as applications may contain security vulnerabilities, *e.g.,* buggy code. However, functions running in SURE are exposed to threats from other, potentially adversarial, users in the same cloud. As shown in Fig. 2, SURE treats the hypervisor and associated toolchains (*e.g.,* emulation of hardware devices and peripherals required by VMs) as part of the TCB. We further establish another layer of trust within the SURE VM, which is responsible for enforcing intra-unikernel isolation between the untrusted user code and the unikernel TCB modules (scheduler, booter, sidecar, network I/O lib, and other OS modules, as shown in Fig. 2).

3.3 SURE's threat model

Based on SURE's trust model and system architecture, we identify the following threat sources due to the inevitable sharing of the address space in SURE: **(1) Vulnerabilities from shared memory processing.** Without rigorously enforced access controls, a malicious function might exploit shared memory to gain unauthorized access to sensitive data or perform memory-based attacks, e.g., Flush+Reload [101], buffer overflows [35] or injection attacks [83]. This risk is particularly pronounced in a public cloud environment, shared by functions from different users. In addition, buggy (even if not malicious) code in user functions may accidentally and improperly manipulate shared data. **(2) Intra-unikernel vulnerabilities from a single address space.** SURE's function runtime (see Fig. 2), including the library-based sidecar and many other TCB modules, are part of the serverless infrastructure and require additional isolation from untrusted user functions. This cannot be guaranteed within a unikernel's *single address space.* A typical threat involves tampering with application-level observability: buggy function code could inject false metrics into the sidecar, disrupting the service mesh that relies on the integrity of metrics to orchestrate the deployment of functions.

3.4 Isolation in SURE

By operating each function as a unikernel-based VM, SURE leverages the VM-based sandbox and the inherent hardware-level virtualization to reduce the attack surface of containerized sandboxes. It also allows for fine-grained isolation that aligns closely with the disaggregated microservice deployment paradigm. SURE additionally introduces the following features to enable inter-function access control, protect shared memory processing, and enable intra-unikernel isolation to separate user code with TCB modules in the unikernel:

Group-based security domains with isolated memory pools. We require a "security domain" to be specified when deploying a function on SURE. We assign mutually trusted SURE VMs (typically from the same user) to the same security domain. Each security domain possesses a private memory pool, established as a POSIX shared memory backend in the host file system, corresponding to a *file.* SURE only allows shared memory processing when functions are within the same domain. We adopt techniques very similar to NetVM [48] for constructing the shared memory pool in a security domain (refer to [48] for details).

Access control with sidecar and SURE gateway. Ownership transfer of a shared memory buffer occurs through a descriptor exchange within the security domain. We verify the eligibility of the receiver of the descriptor to prevent unauthorized shared memory access. SURE takes advantage of the library-based sidecar to apply traffic filtering (with a whitelist of allowed peers) on the RX and TX paths of the SURE VM (Fig. 5). The sidecar discards the descriptor if the whitelist does not match. Further, we use the SURE gateway to perform a copy between memory pools (on the same node) in different security domains (Fig. 3). We also use the SURE gateway to enforce cross-node access control by having traffic filtering as part of Z-stack's protocol processing.

Memory isolation and MPK-based call gates. In general, shared memory processing and intra-unikernel interactions (e.g., between user code and LibOS modules) essentially become memory accesses. This requires memory-level isolation to prevent unwanted memory access in SURE. Rather than adhering to the conventional approach of kernel-userspace separation and relying on heavyweight system calls (*e.g.,* mmap, mprotect) to constrain memory access initiated by user code, SURE opts for a more streamlined and rapid solution by using Intel's MPK [13] to mitigate the risk of unwanted memory accesses. This effectively retains the data plane performance enhancements of SURE, while enabling isolation at the granularity of memory pages. We protect the integrity of our sidecar and unikernel LibOS modules from possibly faulty (or malicious) user-provided function code. With MPK, we can selectively grant access to portions of the shared memory to VMs taking part in a data exchange. SURE offers crucial support to take care of dependencies, such as NetIO lib, for users to transparently and securely interact with the SURE data plane and sidecar. This allows us to construct and expose *secure* APIs derived from platform-provided libraries. These APIs serve to standardize MPK utilization, thereby reducing the risk of unintentional privilege escalation.

3.5 Preventing privilege escalation of MPK with enhanced TCB

There are several critical subsystems (including scheduler, page table management, interrupt service handling) in the TCB of the unikernel that must manipulate the PKRU register or the PTE that stores the MPK key during their operation (more details in §5.1). Ensuring these TCB components in the unikernel are safe and free of errors is needed, which is to guarantee that the MPK-based isolation in a unikernel is effective. This helps avoid unforeseen privilege escalation and invalid MPK protection in SURE. However, the single-address-space design of unikernels does not guarantee their integrity in the presence of malicious/misbehaving code.

The root cause is the lack of protection for sensitive data structures (*e.g.*, PTEs, thread contexts) in the TCB that are directly related to the access privilege of a particular MPK key. For example, the PTE contains the MPK key that represents the access privilege of the memory page, and the thread context contains PKRU that specifies the access privilege of the given MPK key. Without proper protection, buggy code could inadvertently elevate access privileges and access memory pages it is not allowed to access.

SURE strengthens vulnerable TCB components to prevent unintended MPK privilege escalation. The key idea is to add additional memory protection to sensitive data structures in TCB. Our enhancement (details in §5.3) includes a blacklist-based mechanism designed to prevent unwanted alterations to the PTEs of protected memory pages. We also safeguard the thread context with MPK, which can only be accessed with our call gate. We apply checks at entry/exit points of interrupt handling to prevent the PKRU leakage.

4 DATA PLANE DESIGN IN SURE

4.1 Distributed, zero-copy communication

SURE comprehensively utilizes shared memory for unikernels in serverless computing. SURE supports reliable (i.e., no loss, in-order) data transfer with back-pressure for shared memory processing. Fig. 3 shows the intra-node shared memory data plane of SURE. VMs in the same security domain on a node share a dedicated memory pool.[1] Each memory pool is composed of pre-allocated buffers (in the *data region*) to store actual message payloads,[2] and a *signal region* for

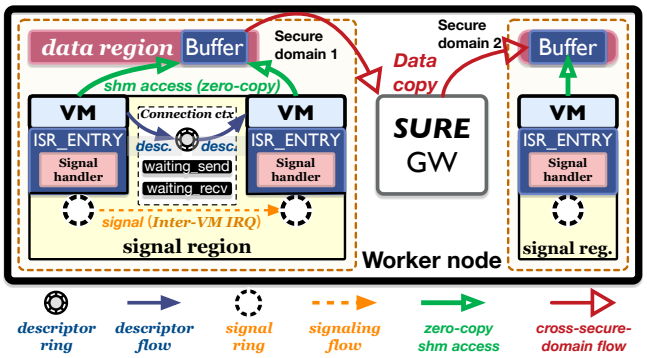

Figure 3: Intra-node data plane in SURE. Communication across security domains within the same node uses SURE gateway (GW) to copy data between memory pools. ISR: Interrupt Service Routine.

descriptor exchanges. SURE allows provisioned concurrency via concurrent threads in a SURE VM to multiplex warm function instances, to reduce cold start, as recommended by AWS Lambda [17]. Threads in a SURE VM can be differentiated by distinct connections. Hence, SURE requires an a priori established connection for data transmission.

Connection management in SURE's shared memory processing. This involves the setup of the connection context in the *signal region*. SURE exposes a set of APIs (i.e., `listen()`, `accept()`, `connect()`, `close()`) for the purpose of connection lifecycle management with the same semantics as is used in traditional socket programming. We can dedicate a distinct connection between a sender thread and receiver thread pair to avoid the head-of-line (HoL) blocking caused by multiplexing over a single connection. As Fig. 3 shows, each connection is represented by a "connection context" in the *signal region*, with a single-producer, single-consumer *descriptor ring* used by the receiver to receive the descriptor. This eliminates potential race conditions in descriptor exchanges between a sender and receiver. Each connection context has two pointers: `waiting_send` and `waiting_recv`. `waiting_recv` supports blocking receive in the event-driven signaling mechanism; `waiting_send` enables the back-pressure between the connected receiver and sender.

Event-driven signaling mechanism: Each SURE VM has a multiple-producer, single-consumer *signal ring* in the *signal region* to receive the wake-up signal from peer VMs. Each SURE VM also has a *signal handler* thread to process *pending* wake-up signals in a batch, triggered by an inter-VM interrupt from hypervisor [8], handled by the corresponding Interrupt Service Routine (ISR). Instead of busy-polling the *descriptor ring*, a receiver in SURE can block if *no* descriptor is currently available. The blocking receiver thread registers its pointer in the `waiting_recv`, waiting for the sender to signal the presence of new descriptors. The sender only signals the blocked receiver by checking `waiting_recv` in the connection context. This reduces the chance of concurrent writes (multiple-producer) to the receiver's *signal ring*. Note

[1] SURE could further achieve very low-overhead memory management by adopting the design in [76, 85, 99]. This requires mapping the address space of a new function to shared memory during the new function's creation, so that other functions can access it (they need to map the new address space locally in their runtime).

[2] SURE doesn't impose any constraint on the format of exchanged data, allowing the exchange of both serialized content (e.g., an HTTP payload) for decoupled applications, or raw binary data for binary-compatible applications, avoiding the serialization/de-serialization overheads. Applications in SURE can also allocate a large enough buffer to store the complete payload, avoiding assembly and disassembly during shared memory data transfer.

that if there are available descriptors, the receiver directly consumes the descriptor without being blocked.

Reliable data transfer and Back-pressure. Entries in the *descriptor ring* are handled in FIFO (first-in, first-out) order. This guarantees the message produced by the sender is always consumed by the receiver in order when there is *no loss*, ensured by the back-pressure mechanism. Back-pressure in SURE blocks the sender if the receiver's *descriptor ring* is full, indicating the recipient is not consuming descriptors at a sufficient rate. In such a case, the sender thread registers its pointer (used by the receiver to identify the sender to be woken up) to `waiting_send` in the connection context, before the sender is blocked. On the other side, the receiver wakes up the sender when room becomes available in its *descriptor ring* of the corresponding connection: When performing a receive on the connection, if the sender is registered to be woken up, the receiver sends it a signal.

Z-stack: Zero-copy userspace TCP/IP stack. Like previous efforts, we also seek to bypass the kernel in Z-stack. But, going a step further, we work with DPDK's Poll Mode Driver (PMD [4]) to DMA the packet between the NIC and shared memory in userspace, while also avoiding context switches and interrupts incurred by the kernel protocol stack [31, 75]. In addition, most protocol stack designs rely on the POSIX-style socket interface (e.g., `send()` [20], `recv()` [18]) to interact with user-space applications. This introduces an additional copy when moving the data between the application's send/receive buffer and the socket buffer (accessed by the TCP/IP protocol stack), and can consume more than 50% of the total CPU cycles [31]. Such a design is influenced by the desire for kernel-userspace isolation, i.e., using a copy to ensure data isolation between the user and kernel space. However, it is unnecessary in SURE as our data plane operates entirely in userspace.

Fig. 4 shows the design of Z-stack. We introduce two new zero-copy APIs (`z_recv()` and `z_send()`) in Z-stack to interface between the application layer and underlying TCP/IP protocol layers by exchanging pointers. Passing the buffer between different layers involves modifications to protocol headers. We base our implementation of Z-stack on top of the existing DPDK F-stack [5], which offers a fully-functional TCP/IP stack ported from FreeBSD integrated with the DPDK PMD. However, F-stack introduces data copies during protocol processing, which we eliminate in Z-stack.

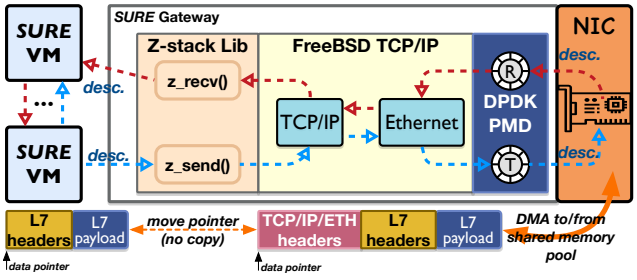

Figure 4: Protocol Processing Pipeline within the Z-stack.

Consolidated protocol processing. The single per-node SURE gateway provides consolidated protocol processing that is performed once for all incoming/outgoing traffic, with the processed payload residing in the shared memory pool. The SURE gateway coordinates between Z-stack and intra-node shared memory processing that helps SURE achieves true *zero-copy* communication in and out of the node, unlike previous efforts such as SPRIGHT [82].

Fault tolerance of SURE gateway: The SURE gateway can be a single point of failure, potentially disrupting inter-function communication, both within a node (when going across security domains) or when going across nodes [77]. We maintain multiple replicas of the SURE gateway, each responsible for handling multiple NIC receive queues. This can be achieved by enabling NIC Receive Side Scaling (RSS) to balance the load between replicas [26]. When a replica crashes, the Indirection Table of RSS is modified to redirect traffic to a healthy replica, and restart a new replica.

Routing management. SURE maintains an IP routing table for each security domain, stored in the *signal region*. Intra-node routing *within* the security domain involves a routing table lookup on the sender side, using the IP 5-tuple to find the correct connection context of the receiver. IP-based routing helps maintain compatibility with inter-node routing (including routing to external clients).

Both intra-node routing *across* security domains and inter-node routing relies on the SURE gateway. In both cases, the sender VM establishes a connection to the SURE gateway (when the receiving VM doesn't belong to the same security domain on the same node or if the receiving VM is located on a remote node). The sender hands the descriptor to the local SURE gateway: (1) if the receiver belongs to another security domain on the same node, the local SURE gateway performs a copy to move the data across security domains, then routes the descriptor to the receiver after looking up the routing table in the destination security domain. (2) if the receiver is located on a remote node, the local SURE gateway sends the data to the SURE gateway on the remote node, using Z-stack. The remote SURE gateway then performs intra-node routing to route the data to the receiver. The routing table is read-only to a SURE VM (guarded by the call gate in §5.2). Only the hypervisor can update the routing table upon establishment of a new connection.

Application compatibility: Unlike applications using stack memory or language runtimes with garbage collection, applications in SURE must explicitly manage (allocate/free) shared memory buffers for inter-function communication. To port legacy applications (e.g., using POSIX, HTTP/REST, or gRPC APIs), we need to replace legacy APIs with SURE's zero copy APIs. We can further leverage automated source-code level

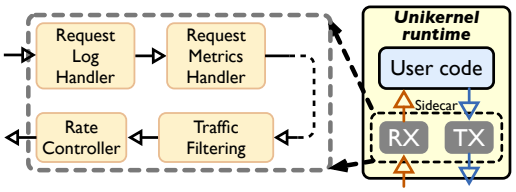

Figure 5: Library-based sidecar in SURE. The sidecar contains a sequence of handlers that perform certain sidecar functionalities on both RX and TX data path of the user function.

program transformation (e.g., tools used by Coccinelle [60]) or create customized communication libraries (exposed as POSIX, HTTP/REST, or gRPC) built with SURE's zero copy APIs, allowing seamless interactions with SURE's data plane.

4.2 Library-based, event-driven sidecar

By following the philosophy of LibOSes, SURE provides sidecar functionalities as a library component, embedded in the application and accessible through simple function calls. This allows us to extend the application in LibOS with a corresponding lightweight sidecar functionality. Instead of having a separate sidecar process running, SURE's LibOS exposes event-based execution hooks for running additional sidecar functions, which has proven successful in eBPF-based service meshes with improved resource efficiency [41, 82]. With this extensible event-driven functionality, SURE offers typical sidecar capabilities in its unikernel, including monitoring and traffic management, with full visibility to L7 payload and is capable of offering good expressiveness of service mesh policies, as in [93]. Unlike other library-based sidecar solutions (e.g., in ServiceRouter [87]), which do not provide isolation between user code and the sidecar module, SURE's library-based sidecar is further protected using MPK.

Event-driven execution hooks: In order to maintain transparent operation of the sidecar relative to the serverless function, we predefine two hook points each on the Receive (RX) path and Transmit (TX) path, located in `send()` and `recv()` in SURE's network I/O libs (Fig. 5) to invoke sidecar functions. SURE allows cloud providers to customize the sidecar by adding/removing sidecar functions to/from hooks based on events they are interested in. Required sidecar functions are organized in an execution sequence, driven by I/O events occurring on the RX/TX paths.

Implementation of library-based sidecar: We adopt the implementation methodology from the popular service mesh solution, Istio [1], which encapsulates a sidecar functionality into a handler function and organizes the sidecar functions into a call sequence, as shown in Fig. 5. We have implemented a set of commonly used sidecar functions in SURE (such as request logging, metrics collection, rate controllers, and access control) to enable critical service mesh capabilities, such as monitoring and traffic management.

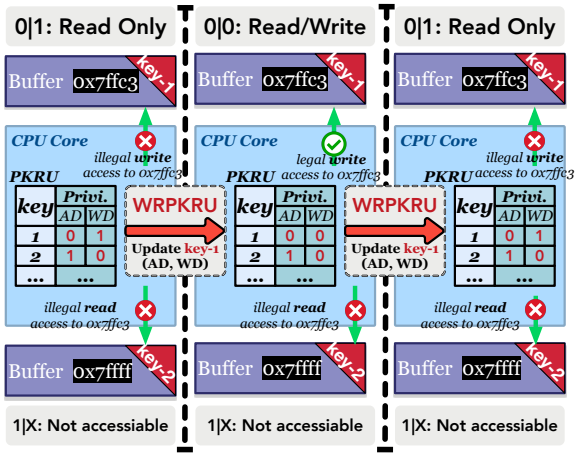

Figure 6: An example of switching the access privileges of MPK keys via WRPKRU. The memory page `0x7ffc3` is tagged with *key-1*. The memory page `0x7ffff` is tagged with *key-2*.

5 MEMORY-LEVEL ISOLATION IN SURE

In this section, we describe how we use MPK to enable memory-level isolation in a shared address space to protect (1) the shared memory data plane between functions and (2) the unikernel TCB (including the library-based sidecar) from an untrusted user application within the single-address-space of SURE VM.

5.1 A Primer on MPK

MPK is a hardware-level, intra-process memory isolation feature introduced in Intel's server CPUs, starting from the Skylake microarchitecture [43]. MPK allows for a total of 16 keys (using a 4-bit ID) to be defined *locally* within a single process [13]. The process is essentially a VM in the context of SURE. The access privilege of all 16 MPK keys is defined by a per-core, 32-bit CPU register, PKRU (Protection Key Register User), where each key's access privilege is described by 2 bits (*AD* and *WD* in Fig. 6) in PKRU [79]: (i) "Access Disable" bit (denoted *AD*) defines whether access (both read and write) is disabled; and (ii) "Write Disable" bit (denoted *WD*), that specifies whether write is disabled. MPK offers a unique x86 instruction WRPKRU to change the access privilege of the MPK key by modifying its corresponding (*AD*, *WD*) bits in PKRU: Read/Write (0, 0), Read-Only (0, 1), or No-Access (1, ×) [79]. The access privilege of an MPK key will be in effect for a memory page after being "tagged" to the Page Table Entry (PTE) of that page. This is done by modifying the 4 reserved bits (representing the ID of the key) in the corresponding PTE and flushing it from the TLB. There are two distinct but complementary approaches to switch the access privilege of a memory page when using MPK, as described next.

#1 – Switching the access privilege of certain MPK keys via WRPKRU. An example of this approach is depicted in Fig. 6. We have two distinct memory pages: `0x7ffc3` and `0x7ffff`. We tag MPK *key-1* to page `0x7ffc3`'s PTE and tag MPK *key-2*

to page `0x7ffff`'s PTE. The PKRU in the user context stores the access privilege of keys: The initial (AD, WD) bits of *key-1* is (0, 1) (Read-Only) and the initial (AD, WD) bits of *key-2* is (1, 0) (No-Access). As such, write access to the page `0x7ffc3` (tagged *key-1*) is illegal and results in a page fault. The read or write access to the page `0x7ffff` (tagged *key-2*) is also illegal as the page `0x7ffff` is tagged with a (No-Access) key.

The user uses WRPKRU to change the access privilege of *key-1* to (0, 0) (Read/Write). After that, write access to the page `0x7ffc3` (tagged *key-1*) is allowed. Note that MPK allows tagging a key to multiple memory pages. The privilege update of the key, achieved using WRPKRU, will be applied to all tagged pages.

#2 – Switching access privilege of certain memory pages by updating the MPK key ID in PTE. Instead of relying on WRPKRU, we can also change the access privilege to a memory page by updating the 4 bits reserved for the MPK key ID in the PTE (i.e., replacing the tagged MPK key for the memory page). Compared to using WRPKRU, *this approach allows for more granular management of the access privilege at the individual page level of a VM.*

5.2 Secure APIs based on SURE call gates

Since MPK provides only 16 distinct keys within a VM (this could be far fewer than the potential number of memory pages in a single VM), it is not feasible to tag each memory page with a distinct MPK key to enable fine-grained access management (via WRPKRU) at the individual page level.

We further adopt Approach-2 in §5.1 to complement WRPKRU (i.e., Approach-1 in §5.1), to enable page-level access privilege management within a SURE VM. Since we only need to distinguish between *unprotected* and *protected* pages, we use 2 out of the 16 distinct keys within a SURE VM: one (called **UK**) is used for tagging unprotected memory pages and the other (called **PK**) is used for tagging protected memory pages. The value of the PKRU is always configured to allow access to memory pages tagged with **UK**, while access to pages tagged with **PK** is disabled when executing user code and only enabled when executing the privileged code of the TCB.

We design a ***call gate*** abstraction to update the PKRU to enable/disable access to **PK** pages and switch execution of the function onto a stack in protected memory when the user code executes a privileged function of the TCB. All other updates to access privilege (*i.e.,* writes to PKRU) are illegal, prohibited via the binary inspection (see §5.3). While most of the memory related to TCB components is statically tagged with the **PK**, shared memory buffers are dynamically tagged by changing the key (**PK** or **UK**). API functions receiving a message in a buffer or allocating a new buffer update the corresponding key to **UK** to allow unprotected access from user code. When functions send a message or free a buffer we perform the inverse operation, setting the key to **PK**.

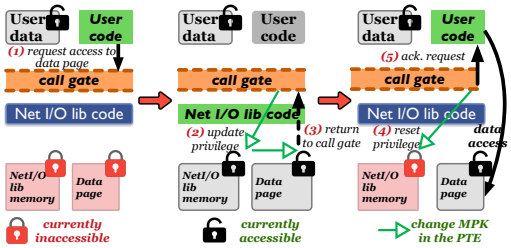

Figure 7: SURE uses call gates to secure function calls by user code. SURE can dynamically change the access privilege of memory pages.

Fig. 7 shows how the privileged network API `recv()` function (within TCB) is invoked by untrusted user code via the MPK-based call gate. Other APIs in the TCB (e.g., in scheduler or page table management) are guarded by the MPK-based call gate in the same way. When running user code, the set of protected pages (i.e., local data in a SURE VM or shared memory) is configured to be inaccessible. ① the user code invokes functions in NetI/O lib via the call gate. ② the call gate adjusts permissions, making protected memory accessible, and invokes the `recv()` function; the NetI/O function receives a buffer descriptor and updates the corresponding MPK key to allow unprivileged user access to the buffer. ③ Upon return to user code, ④ the call gate disables access to protected memory, while the received buffer remains accessible.

Exclusive, Zero-copy Access to Shared Memory Buffers. Once user code has exchanged ownership of a shared memory buffer through our secure API, it may need to access its content multiple times. To avoid the cost of copying the whole message to an unprotected buffer, we selectively disable memory protection for memory buffers owned by the VM, by changing the key from **PK** to **UK** in their PTEs. This operation requires updating the PTE of the buffer and flushing the corresponding TLB entry. Although this operation is not as lightweight as the WRPKRU instruction, it is still overall cheaper than copying the whole buffer, as we evaluate in §6.3. Memory protection for a buffer is lifted in the `buffer_get()` and `recv()` functions, and re-enforced with the `buffer_put()` and `send()` functions.

5.3 Enhanced unikernel TCB in SURE

As discussed in §3.5, SURE has to secure three additional systems that in a traditional OS would be protected by the user/-supervisor privilege separation to guarantee the integrity of MPK-based memory protection: the paging API; the scheduler; and interrupts.[3]

(1) Paging API protection. SURE protects memory pages by writing an MPK key in their corresponding PTEs. However, PTEs are not protected in the single-address-space unikernel. Thus, untrusted code may manipulate these PTEs to

[3]Note that SURE does not offer control flow integrity of MPK, nor protection against side-channel and microarchitectural attacks, which are beyond the scope of our threat model. These issues can be addressed by existing approaches [38, 55, 97].

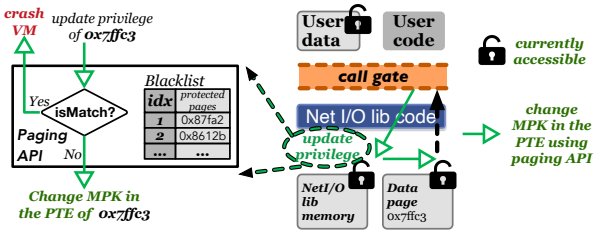

Figure 8: SURE uses the blacklist mechanism to prevent unwanted updates to the PTEs of protected pages (invoked by SURE's paging API).

gain unauthorized access to protected memory pages. This may be done either by changing the MPK key in the PTE or remapping the (guest) physical address of a protected page to a different, unprotected PTE.

To prevent these exploits, SURE stores the (guest) physical address of all protected pages in a blacklist. All pages containing a page table are also tagged with the protected key, hence preventing untrusted code from directly modifying the PTEs. To interact with the page table, a trusted paging API protected with call gates is provided. Fig. 8 shows the example of how the blacklist works when malicious code tries to update the MPK key of a protected page: whenever a PTE is modified, the API functions check whether the physical address in the PTE that is being modified belongs to the blacklist (i.e., the physical page is protected), and in case of a match, crash the unikernel. If the requested page address does not match any entry in the blacklist, the paging API then updates the MPK key in the PTE of the requested page, making the page accessible to the user code. This same procedure is applied when remapping the address of a page to a different PTE via the paging API in SURE.

(2) Scheduler protection. The scheduler is in charge of core operations of the lifecycle of a thread in the unikernel, including context switching that involves storing/loading the thread context into/from memory. The thread context in the SURE VM contains the PKRU register (Fig. 6), directly related to SURE's guarantee of memory protection. Thread context switches occur often in SURE's data plane, *e.g.,* blocking receive, back-pressure, and interrupt handling. Thus, the thread context must be stored in protected memory throughout its lifecycle (Fig. 2). Hence, we run the whole scheduler as privileged code and allow its access from untrusted code only through call gates for safe operation of thread context.

(3) Interrupt validation. SURE relies on Interrupt Service Routines (ISRs) to handle receiving event-driven shared memory processing signals (Fig. 3). Typically, upon entering an ISR, the unikernel scheduler stores the interrupted thread context (*e.g.,* PKRU, instruction pointer (IP) and stack pointer (SP)) on the *unprotected* interrupt stack. The implemented interrupt handler is then executed to handle the interrupt event. Upon completion, the ISR exits and restores the interrupted thread context to continue the execution of the interrupted task. However, we note that upon entry of the ISR, the PKRU on the CPU currently executing the ISR remains the same as the PKRU value (*AD*, *WD*) of the interrupted thread. An untrusted ISR gains unallowed privileges and could corrupt these values (IP, SP) and, in case of interruption of privileged code, cause the return to unprivileged code with access to protected memory.

The root cause is the lack of checks upon ISR entry and exit. To prevent the exploitation of this feature, we extend the entry code common to all ISRs to (1) set the PKRU to the unprivileged value upon interrupt entry and restore it to the privileged value upon exit if a privileged function was interrupted; (2) store the (IP, SP) pair in a per-CPU protected-memory area when privileged code is interrupted, and check that the values have not been modified on the stack before returning from the interrupt. We further protect the memory region containing the IDT[4] with MPK, to guarantee that the ISR's entry code cannot be changed. As SURE only protects the entry/exit points of the ISR, we retain the flexibility for the user to customize interrupt handlers in their ISR, while avoiding unintended memory access caused by buggy code in the ISR. Interrupt validation in SURE can also be applied when we include other ISRs in the SURE VM, e.g., for interrupt handling for virtual I/O devices (disk, virtio, etc).

(4) Binary inspection of MPK-related instructions. To guarantee that only safe occurrences of MPK-related instructions (WRPKRU and XRSTOR) are present and untrusted code cannot manipulate PKRU without passing through our call gate, we couple MPK with binary inspection of the executable, as in [97]. We also enforce a strict write-xor-execute memory policy [23] in SURE. This guarantees that executable pages are not modified at run time with instructions that change memory access permissions.

6 PERFORMANCE EVALUATION OF SURE

We quantitatively evaluate the performance improvement and resource efficiency with SURE's data plane. We start with a microbenchmark analysis, to quantify the benefit of each design choice in SURE's data plane. We also evaluate SURE with the realistic online boutique workload [16] from Google.

Implementation of SURE: We base the development of SURE on Unikraft [58] (version 0.16.2), an automated system for building unikernels. This allows us to reuse key OS building blocks from Unikraft, such as the scheduler, memory allocator, and file systems. We use QEMU/KVM (8.2.0) as the hypervisor. SURE adds support for the shared memory device and library-based sidecar in Unikraft. SURE employs the Inter-VM Shared Memory Device [8] from the QEMU VMM to expose the shared memory pool as a PCIe device to

[4]On x86 systems, interrupts are handled through an Interrupt Descriptor Table (IDT) which stores, for each interrupt vector, the address of the entry point of its ISR.

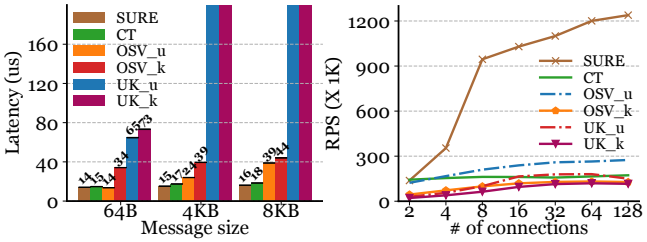

Figure 9: Intra-node data plane performance. `CT`: container with Linux bridge; `OSv_u`: OSv with userspace OVS; `OSv_k`: OSv with Linux bridge; `UK_u`: Unikraft with userspace OVS; `UK_k`: Unikraft with Linux bridge; (average of 30 repetitions)

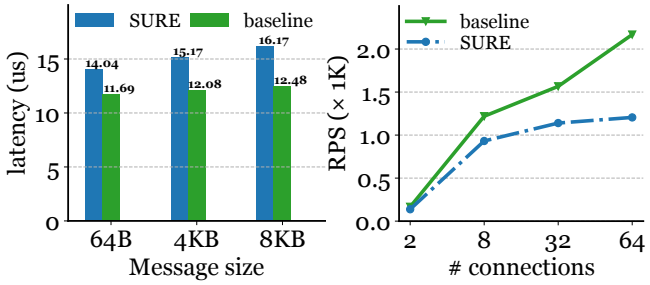

Figure 10: The impact of MPK on SURE's performance.

the VM. We enhanced the Unikraft TCB, including page table management, scheduler, and interrupts handler, to protect them from MPK privilege escalation. These changes are also applicable (suitably modified) to OSv [57] unikernels.

Note that some VMMs, including Firecracker [24], currently do not support inter-VM shared memory devices. This makes it challenging to deploy SURE's shared memory data plane in FaaS offerings such as AWS Lambda, which uses Firecracker as the VMM. We have kept this as an additional implementation (primarily engineering) effort for the future. However, SURE's design, including the library-based sidecar, MPK-based call gate, and enhanced unikernel TCB, can generally be integrated into existing FaaS offerings to strengthen the isolation of serverless functions.

Testbed Configuration: We built our testbed on NSF Cloudlab using three nodes, equipped with a 32-core Intel Xeon Silver 4314 CPU running at 2.4 GHz, 128GB of memory, and a 100Gb NIC for network connectivity. The CPU has MPK support. Throughout the experiments, we use Ubuntu 22.04, kernel version 5.15 as the OS.

6.1 Microbenchmark Analysis

6.1.1 Improvement from shared memory processing. We evaluate the round-trip latency and throughput between a client and server pair on the *same* node first. We choose three message sizes: 64B, 4KB, and 8KB. There is very little variation for SURE for packet size ranging from 64B to 1KB. We compare SURE's intra-node shared memory data plane with the following widely used alternative commercial and open-sourced serverless platforms: (1) Container (denoted **CT**); (2) Unikraft unikernel [58] (denoted **UK**); (3) OSv unikernel [57] (denoted **OSv**). For these alternatives, we use the kernel Linux bridge for L2 connectivity. We additionally consider the userspace Open vSwitch (OVS [80]) for **UK** and **OSv**.

To match SURE's reliable data transfer, we use TCP for reliable data transfer with the others. Note that container uses the host's TCP/IP stack, Unikraft uses *lwip* [12], and OSv's TCP/IP stack is ported from FreeBSD. For throughput measurements, we add sufficient clients to saturate the server, and metrics are collected on the server. To accurately assess the improvements from shared memory processing, we disable SURE's sidecar for this experiment.

SURE *achieves low-latency.* We show the latency for a single client-server connection in Fig. 9 (left). SURE has the lowest latency (14-16us) across all evaluated message sizes. Unlike other alternatives, SURE's shared memory zero-copy data transfer results in the latency being flat with increasing message sizes. We note that *lwip* (used in the Unikraft setup) incurs higher latency as it is under-optimized to work with virtio devices (no checksum offload[5] and extra copy[6]).

SURE *is more scalable and efficient.* We evaluate throughput (requests per second (RPS)) as the number of concurrent connections increases. Fig. 9 (right) shows the RPS for a message size of 64B. Our observations are consistent across other message sizes (4KB, 8KB). Compared to other alternatives, SURE is also more efficient: with more than 16 connections, all alternatives have their server's one assigned CPU core saturated. But SURE has a much higher RPS (and still increases with increasing concurrent connections).

6.1.2 Cost of Memory Isolation with SURE. SURE has MPK enabled by default. To see the performance impact of MPK, we consider a variant of SURE with MPK disabled (**`baseline`**). We use the same functions, varying the message sizes.

MPK in* SURE *has limited penalty. Fig. 10 shows the normalized latency and throughput. With a single connection, SURE shows 1.2-1.3× increased delay compared to the **`baseline`**. As we increase the number of concurrent connections, SURE's RPS decreases (*e.g.,* 1.8× reduction at 64 connections). We believe this overhead is relatively small for the reward of robust memory-level isolation in our shared memory data plane and the unikernel TCB. `mprotect()` is a system call that can change the access privilege of specified memory pages, similar to MPK. However, as reported in [79], switching the access privilege with MPK incurs only ~20 CPU cycles. Using `mprotect()` on the other hand requires more than 1000 CPU cycles to complete, resulting in much poorer performance.

6.1.3 Protecting message buffers with PTE updates. We compare SURE's zero-copy approach of enabling/disabling access

[5]https://savannah.nongnu.org/patch/?10111
[6]https://github.com/unikraft/lib-lwip/blob/staging/uknetdev.c#L166-L167

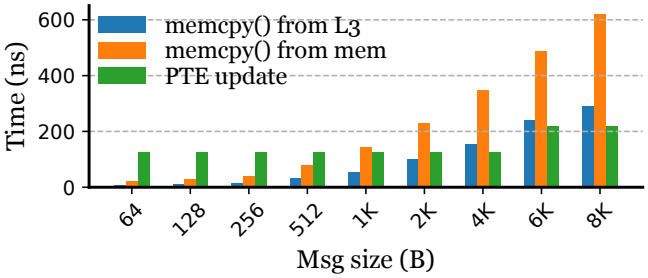

Figure 11: Cost of copying messages of different sizes from L3 cache or main memory compared to updating their buffer PTEs.

to shared memory buffers (§5.2) with the alternative of copying messages from protected shared memory to user memory, in the context of receiving a message. To test SURE's approach, we flag pages containing the message buffer as accessible in their PTEs, flush the corresponding (outdated) TLB entries, and access the buffer. This would cause a TLB miss and the fetching of updated entries from the page table. For the copy approach, we use the `memcpy()` function from **glibc** to copy the content of the message buffer to a local user buffer. Since we expect a message coming from another function running on a different CPU core to reside either in the shared L3 cache or in main memory, we design the experiment to access the remote buffer from the L3 cache or from main memory.[7] Fig. 11 shows how SURE's approach of PTE updates outperforms copying 1 KB or larger messages from main memory. It is also better than copying for a message greater than 4 KB, when it is in the L3 cache. PTE remapping clearly wins for large messages, while copying is still better for small messages. Currently, SURE adopts PTE updates for any message size. We leave the design of a combined approach similar to [62] for the future.

6.1.4 Improvement with library-based sidecar. We consider the individual container for each sidecar as the baseline to compare with. Each sidecar connects to the user function container over the kernel loopback interface [110]. We use the NGINX proxy as the implementation of the sidecar, as demonstrated in production [15]. For the NGINX setup, we assign one CPU core to the NGINX sidecar and another core to the user function. In the SURE setup, the library-based sidecar shares a CPU core with the user function code. Note that we measure the CPU cycle consumption and the added delay of the sidecar for a single connection. When measuring the throughput, we use concurrent connections to saturate the user function and sidecar. We show the throughput results with 64 connections in Table 1. However, the observations are consistent for other values (*e.g.,* 32, 128 connections).

Library-based sidecar shows negligible overhead. Table 1 compares the CPU cycles spent on the sidecar for different alternatives. The CPU cycles consumed by our library-based sidecar are negligible compared to those of an NGINX sidecar (only 0.9%). The higher CPU cycle consumption by the NGINX sidecar is due to the loose coupling between the sidecar and the function, which results in additional overhead from the kernel's loopback interface. The extra CPU consumption also results in increased network delay and decreased throughput: the library-based sidecar adds only 0.21-0.23 μs to the data path, while the NGINX sidecar adds more than 24μs of delay, potentially severely impacting data plane performance. The throughput of SURE with a library-based sidecar is also close to SURE with sidecar disabled ("*SR (no SC)*" in Table 1). This is the advantage of our library-based sidecar - maintaining low latency and high throughput, with negligible CPU consumption.

[7]We do this by randomly accessing multiple remote buffers distributed over an increasingly large memory area.

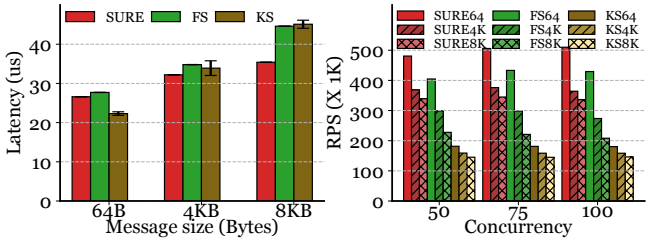

Figure 12: Performance of SURE's Z-stack: (left) latency with 1 connection; (right) RPS under different concurrency levels. FS: F-stack; KS: kernel stack.

Library-based sidecar has lower memory footprint. The individual sidecar serves as a reverse proxy between the user function and the external client, inevitably requiring additional dependencies and having a larger memory footprint. Our library-based sidecar avoids this overhead almost entirely. Our analysis shows that SURE's library-based sidecar (125KB) reduces the memory footprint by 165×, compared to the NGINX sidecar (20.2MB). This has many benefits, including increased function density on every node.

6.1.5 Benefit of the zero-copy TCP/IP stack. We compare Z-stack (denoted **ZS**) with F-stack [5] (denoted **FS**) for protocol processing in the SURE gateway (Fig. 12).

The F-stack gateway incurs a data copy when exchanging payloads with the function. We also compare Z-stack with the Linux kernel's protocol stack (denoted **KS**) to evaluate the performance improvement and costs of using the DPDK PMD. In the **KS** setup, we let the function directly access the kernel stack without involving the gateway. We use a TCP echo server/client pair (integrated with the different alternatives) for this experiment. The server and the client are deployed on different nodes.

Table 1: Library-based sidecar (SR) vs. Individual sidecar (NGINX)

Msg size	CPU cycles (× 1K)		Added delay (us)		Throughput (MBytes per second)		
	SR	*NGINX*	*SR*	*NGINX*	*SR (no SC)*	*SR*	*NGINX*
256B	0.50	60.4	0.21	25.2	342	309	12.3
4KB	0.55	59.5	0.23	24.8	3697	3533	185
8KB	0.55	58.2	0.23	24.2	5525	5369	337

ZS achieves a ~1.2× RPS improvement and latency reduction under high traffic load (more than 100 connections) compared to **FS** (Fig. 12). This clearly showcases the advantage of having zero-copy protocol processing. On the other hand, **FS** inevitably introduces data copies between the server function and the F-stack gateway, resulting in lower performance. SURE also shows significant RPS improvement compared to the kernel protocol stack. SURE not only avoids data copies, it also eliminates other kernel-related overheads. Note that SURE shows slightly higher latency than **KS** under very light loads with small packets (e.g., 64B, single connection in Fig. 12 (left)) as SURE uses the SURE gateway to relay between the function and Z-stack, resulting in some additional delay. But SURE is significantly better than **KS** for larger message sizes.

Assessing the polling "tax" of Z-stack. SURE chooses to use DPDK's busy-polling PMD to move packets between the SURE gateway (with the Z-stack) and the NIC. SURE dedicates a CPU core to the SURE gateway for protocol processing. This CPU cost is spent independent of traffic load, unlike an interrupt-driven kernel, and is the "polling tax" of Z-stack.

The interrupt-driven kernel stack (**KS**) achieves better CPU efficiency at light load due to on-demand execution (at 30K, 60K RPS in Table 2). But, **KS** is inefficient for higher loads (≥90K), because of interrupt handling [75] and other kernel overheads. In comparison, SURE achieves the same RPS consistently using a *single* CPU core with its kernel-bypass, eliminating interrupts. Busy-polling on a *single* CPU core has better overload behavior and is more efficient under heavy traffic load than an interrupt-based kernel stack. The increased function density on a single node (*e.g.,* with bin-packing-based function placement) can facilitate the sharing of the SURE gateway and amortize the cost of busy polling.

Table 2: Polling tax of Z-stack.

Tested Load (X 1K RPS)	Kernel stack CPU (%)			Z-stack CPU (%)
	Interrupt	*Others*	*Total*	
30	10	14	**24**	100
60	36	33	**69**	100
90	54	58	112	**100**
120	75	78	153	**100**
240	108	94	202	**100**

6.2 Function Startup in SURE

We compare the boot time of running functions within unikernel VMs (including SURE) against containers, as well as the impact that our shared memory data plane and MPK-based memory protection have on boot time.

We use a bare-bones Unikraft VM (**UK-BB**) as the baseline for the unikernel boot time (no network interfaces, no support for virtual memory or multithreading, etc.). We compare a SURE VM w/ and w/o MPK memory protection (respectively, **SURE MPK** and **SURE**), both without a sidecar, with the baseline. We also compare with a Unikraft VM suitable for

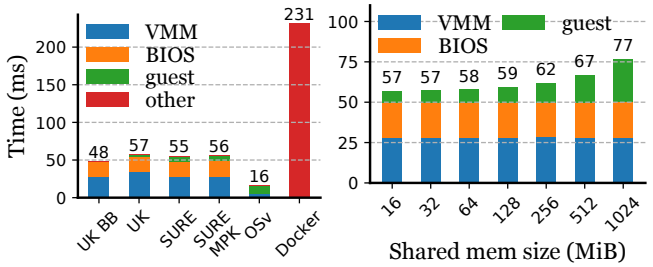

Figure 13: Boot time: (left) comparison of different isolation solutions; (right) influence of the shared memory size on SURE VM's with MPK protection.

running serverless functions (**UK**), equipped with a vhost network interface, virtual memory, and multithreading, as well as a Docker container (**Docker**). All VMs are equipped with 8 MiB of RAM, and SURE VMs are additionally attached to a 16 MiB shared memory region. For the Unikraft and SURE based solutions, we use QEMU as the VMM and measure the boot time as described in [40], which provides a breakdown of the time needed to configure the VM in the VMM, to run the BIOS, and to boot the guest. We additionally consider a microVM-based solution (called "**OSv**" in our figure) that uses the OSv unikernel [57] and Firecracker as the VMM [24]. We run the Docker container with the default configuration and measure the time elapsed between issuing the `docker run` command (with the image available locally, no download), and the execution of the first instruction in the `main()`.

Fig. 13 shows how unikernel-based solutions generally allow much faster boot times. SURE achieves a ~4× speedup compared to Docker containers. Additionally, the shared memory data plane employed by SURE (**SURE** column) is comparable to a traditional network setup based on virtio/vhost interfaces (**UK**). Adding memory protection has a negligible impact on boot time. Note that most of the unikernel boot time is contributed by the VMM and BIOS, which can be improved by leveraging a microVM VMM such as Firecracker [24] (Fig. 13 shows the guest boot time of SURE (based on Unikraft) is comparable to OSv+Firecracker). This can benefit both SURE and the other unikernel approaches.

The amount of shared memory attached to a SURE VM has a non-negligible impact on the guest boot time when MPK protection is enabled, as shown in Fig. 13. The reason is the need to map the memory in the page table, and tag pages with the proper MPK key. This underscores the need to carefully size the shared memory pool to maintain fast boot times. In our experiments, 16 MB of shared memory, with 8 MB for control structures (connections, signal rings, etc.) and 8 MB storing 2048 4K message buffers, was sufficient to support the demanding Online Boutique application (§6.3).

6.3 Realistic Workload Evaluation

Online Boutique [16] is an example microservices-based online store application from Google containing 10 functions and up to 6 different function chains in a serverful setup,

by default using a container-based runtime and using gRPC to interconnect functions (called "**SF-CT**"). We additionally consider a serverful alternative based on OSv unikernels [57] with Firecracker VMM [24] (called "**SF-OSv**"). We use the kernel Linux bridge for connectivity between functions in **SF-OSv** and **SF-CT**. For the call graphs of the Online Boutique function chains, refer to [16].

We consider three serverless platforms to compare against: Knative (termed "**SL-KN**") as the baseline, two state-of-the-art serverless platforms, SPRIGHT [82] (termed "**SL-SP**") and NightCore [50] (termed "**SL-NC**"). We use Locust [11] as the load generator using the boutique's default workload [16]. Note that we disable the user wait time of the default to generate a heavier workload. We ported the boutique's microservices to SURE and OSv. We compare these alternatives with two distinct deployment settings: (1) All functions are deployed on the same node (intra-node), and (2) Deploy the Frontend, Checkout, and Recommendation functions (intermediate functions that could become hotspots) on one node, and deploy the remaining (leaf) functions on a second node (inter-node). Note, NightCore [50] does not support inter-node communication between functions of the same chain.

RPS and Tail Latency. As shown in Fig. 14 (for intra-node setup), the RPS of SURE is up to 17× and 79× higher than the **SF-CT** and **SL-KN**. Both **SF-CT** and **SL-KN** are CPU resource-limited (the RPS barely increases) for concurrency above 16. In addition, the 95%ile latency of SURE (at a concurrency of 16) is below a millisecond (0.39ms), making SURE highly attractive for latency-sensitive microservices. This shows the compelling performance improvement of the shared-memory processing and library-based sidecar of SURE. **SL-SP** also uses shared memory processing for intra-node networking. But, incoming client requests are handled by the kernel protocol stack, before being delivered to shared memory. This reduces RPS by 8×, with significant increases in tail latency compared to SURE. **SL-NC**'s RPS and latency performance is also worse than SURE due to **SL-NC**'s reliance on the kernel protocol processing and additional queuing delays in the NightCore engine (the data plane component equivalent to our SURE gateway).

With the inter-node setup (Fig. 15), SURE still maintains its superior performance: SURE achieves up to 6× and 19× higher RPS than **SF-CT** and **SL-KN**. Still, the zero-copy protocol processing in Z-stack of SURE maintains sub-millisecond latency, even at the 95%ile (CDF shown in Fig. 15 (b)), unlike **SL-SP**, whose tail latency is close to **SF-CT**, as intra-node shared memory processing does not help for the inter-node setup. This is consistent with the microbenchmark of Z-stack in Fig. 12. **SL-KN**'s tail latency is 15× higher, and **SF-CT**'s tail latency is also 4.8× higher than SURE's. Note that SURE and SPRIGHT use a static CPU core allocation for the gateway,

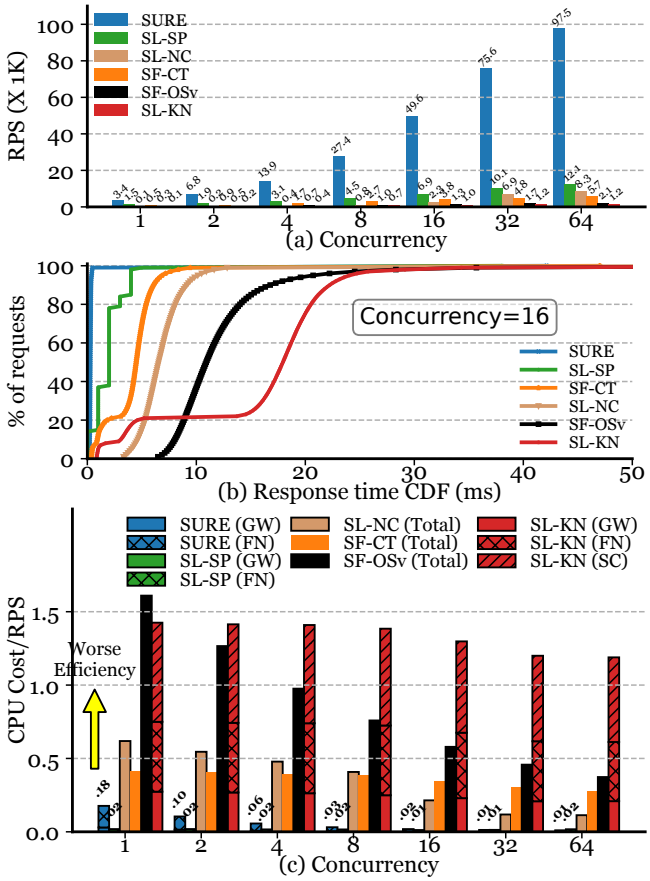

Figure 14: Online boutique results (intra-node): (a) RPS, (b) Response Time, (c) CPU Cost Per RPS.

while the other alternatives support dynamic multi-core scaling. This results in some alternatives (e.g.,**SF-CT**) seeing improved performance when switching from the intra-node setup to the inter-node setup.

The RPS and response time performance of **SF-OSv** is far worse than SURE (both intra-node and inter-node cases), as observed in Fig. 14 and Fig. 15. E.g., at a concurrency level of 64, **SF-OSv**'s RPS is 46X lower than SURE in the intra-node case, and **SF-OSv**'s RPS is 25X less than SURE in the inter-node case. **SF-OSv**'s poor performance is because Firecracker does not support vhost-net acceleration for its network devices (for security reasons, disallowing VM-host kernel interactions[8]), resulting in all network transfers having to traverse the VMM in user space (similar to Xen). This concern of security is not an issue for SURE, since SURE uses full-userspace networking and uses MPK to enforce isolation between the SURE gateway and untrusted functions.

CPU Efficiency. For an apples-to-apples comparison of CPU usage across different alternatives, we use the metric "CPU Cost Per RPS", which is defined as the average utilization of

[8]https://github.com/firecracker-microvm/firecracker/issues/3707

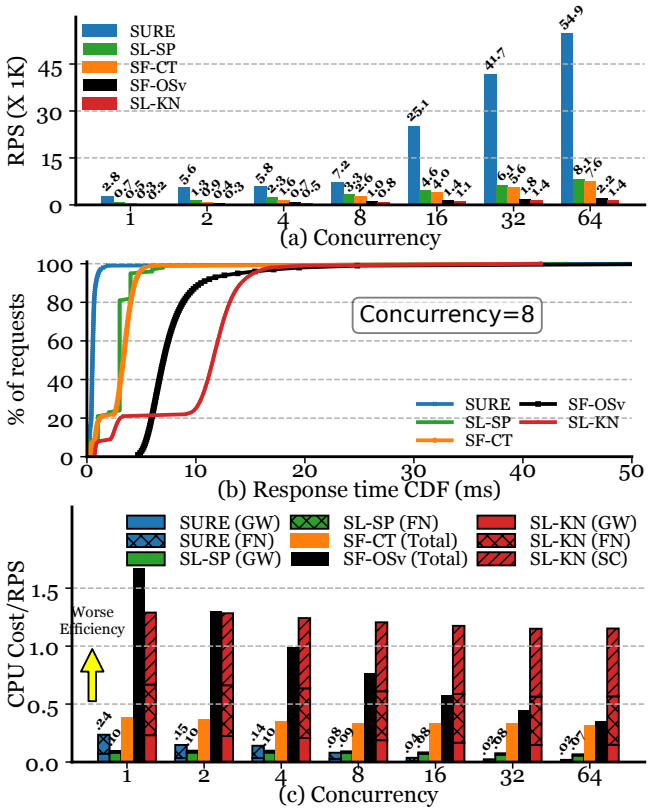

Figure 15: Online boutique results (inter-node): (a) RPS, (b) Response Time, (c) CPU Cost Per RPS.

all CPU cores (expressed as a percentage) divided by Request Per Second (RPS): $\frac{\text{Average CPU Core Utilization}}{\text{RPS}}$. This metric indicates how much CPU is utilized per request. *Lower* values of CPU Cost per RPS suggest that each request requires fewer CPU cycles, indicating a *more efficient* use of the CPU.

We show the intra-node CPU Cost Per RPS comparison in Fig. 14 (c) and inter-node case in Fig. 15 (c). We further break down this CPU Cost into different components: "GW" denotes gateway, "FN" denotes function, "SC" in **SL-KN** setup specifically denotes container-based sidecar. We can observe that **(1) SL-SP** is consistently efficient in using the CPU since it uses eBPF-based event-driven shared memory processing, as observed in [82]. **(2)** At a low concurrency ($\leq$ 16 for intra-node and $\leq$ 4 for inter-node), SURE's efficiency is lower than **SL-SP** (comes from polling and the use of a CPU for each function), but SURE delivers a much higher RPS and is much more efficient than **SL-KN** and **SF-CT**. As concurrency goes up in both intra-node and inter-node cases, SURE is more efficient than **SL-SP** under high concurrency levels. This is because *(i)* - **SL-SP** depends on kernel-based networking for inter-node traffic, which in total takes more than 2 CPU cores (aggregating the CPU usage of the gateways in **SL-SP** on the two nodes) for concurrency level more than 4; *(ii)* - More concurrent processing amortizes the cost of polling in SURE's gateway and functions. The CPU usage of SURE gateway also does not grow substantially, unlike **SL-KN** and **SL-SP**. **(3) SF-OSv**, **SL-KN** and **SF-CT** are very inefficient because they involve kernel networking. **SL-KN** has the worst efficiency as it adopts a heavyweight sidecar for each function within the function chain. **SF-OSv** has poor efficiency for low concurrency levels ($\leq$ 16), which we attribute to the lack of vhost-net acceleration, causing packets to traverse the VMM, which is equivalent to an individual sidecar. **SF-CT** is slightly better since it avoids the sidecar. However, the cumulative CPU cost of **SF-CT** will be unacceptable since functions in **SF-CT** are always-on ("Serverful") and occupy CPU/memory resources even with no requests to process.

7 CONCLUSIONS

SURE is a unikernel-based, lightweight serverless framework that offers high-performance inter-function networking and lightweight library-based sidecars for service mesh. The unikernel-based runtime brings 4× faster startup time compared to docker containers. SURE utilizes MPK-based call gates to enable fine-grained access management at page level. We further enhance the unikernel TCB in SURE to mitigate the vulnerabilities of shared memory processing and single-address-space unikernels while retaining high performance and resource (CPU and memory) efficiency. Benchmarked against a serverful gRPC-based alternative in a multi-node deployment for a complex web workload, SURE's data plane delivers up to 6× increase in throughput and a 4.8× reduction in tail latency while being more secure. The popular open-source containerized serverless platform (Knative [9]) achieves only 5.3% of SURE's throughput and has up to a 15× higher tail latency. SURE's zero-copy protocol processing, Z-stack, expands shared memory processing go across nodes. When handling inter-node traffic, SURE achieves 4× higher RPS than SPRIGHT, which depends on kernel-based networking for inter-node traffic. SURE seamlessly supports serverless function processing to multiple nodes.

Acknowledgements: We thank the US NSF for their generous support through grants CNS-1818971, CNS-2210379. This work was also funded by: the EU (European Union) under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, partnership on "Telecommunications of the Future" (PE00000001 - program "RESTART"); the Italian Ministry of Education and Research (MUR) through the PRIN project NEWTON (No. 2022ZA8T22); the Smart Networks and Services Joint Undertaking (SNS JU) under the EU's Horizon Europe research and innovation programme under Grant No. 101139067. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the EU. Neither the EU nor the granting authority can be held responsible for them. Federico Parola was also funded by TIM S.p.A. through the PhD scholarship.

REFERENCES

[1] 2024. https://istio.io/. [online].
[2] 2024. Cilium. https://cilium.io/. [online].
[3] 2024. Compatibility of Unikraft. https://unikraft.org/docs/concepts/compatibility. [online].
[4] 2024. DPDK Poll Mode Driver. https://doc.dpdk.org/guides/prog_guide/poll_mode_drv.html. [online].
[5] 2024. F-Stack. https://www.f-stack.org/. [online].
[6] 2024. gvisor: The Container Security Platform. https://gvisor.dev/. [online].
[7] 2024. Implementing Microservices on AWS. https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/microservices-on-aws.html. [online].
[8] 2024. Inter-VM Shared Memory device. https://www.qemu.org/docs/master/system/devices/ivshmem.html. [online].
[9] 2024. Knative. https://knative.dev/docs/. [online].
[10] 2024. Kubernetes. https://kubernetes.io/. [online].
[11] 2024. Locust: A Modern Load Testing Framework. https://locust.io/. [online].
[12] 2024. lwIP - A Lightweight TCP/IP stack. https://savannah.nongnu.org/projects/lwip/. [online].
[13] 2024. Memory protection keys. https://lwn.net/Articles/643797/. [online].
[14] 2024. NanoVMs. https://nanovms.com/. [online].
[15] 2024. NGINX Service Mesh. https://www.nginx.com/products/nginx-service-mesh/. [online].
[16] 2024. Online Boutique. https://github.com/GoogleCloudPlatform/microservices-demo. [online].
[17] 2024. Provisioned Concurrency for Lambda Functions. https://aws.amazon.com/blogs/aws/new-provisioned-concurrency-for-lambda-functions/. [online].
[18] 2024. recv(2) — Linux manual page. https://man7.org/linux/man-pages/man2/recv.2.html. [online].
[19] 2024. runu OCI runtime. https://unikraft.org/docs/getting-started/integrations/container-runtimes. [online].
[20] 2024. send(2) — Linux manual page. https://man7.org/linux/man-pages/man2/send.2.html. [online].
[21] 2024. Unix domain socket. https://en.wikipedia.org/wiki/Unix_domain_socket. [online].
[22] 2024. What is FaaS? https://www.ibm.com/topics/faas. [online].
[23] 2024. Write XOR eXecute. https://en.wikipedia.org/w/index.php?title=W%5EX&oldid=1145060869. [online].
[24] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache
[25] Anjali, Tyler Caraza-Harter, and Michael M. Swift. 2020. Blending Containers and Virtual Machines: A Study of Firecracker and GVisor. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Lausanne, Switzerland) *(VEE '20)*. Association for Computing Machinery, New York, NY, USA, 101–113. https://doi.org/10.1145/3381052.3381315
[26] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire, and Dejan Kostić. 2019. RSS++: load and state-aware receive side scaling. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies* (Orlando, Florida) *(CoNEXT '19)*. Association for Computing Machinery, New York, NY, USA, 318–333. https://doi.org/10.1145/3359989.3365412
[27] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the Killer Microseconds. *Commun. ACM* 60, 4 (mar 2017), 48–54. https://doi.org/10.1145/3015146
[28] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 49–65. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay
[29] Daniel Bittman, Peter Alvaro, Pankaj Mehra, Darrell D. E. Long, and Ethan L. Miller. 2020. Twizzler: a Data-Centric OS for Non-Volatile Memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 65–80. https://www.usenix.org/conference/atc20/presentation/bittman
[30] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: Skip Redundant Paths to Make Serverless Fast. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 32, 15 pages. https://doi.org/10.1145/3342195.3392698
[31] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding Host Network Stack Overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 65–77. https://doi.org/10.1145/3452296.3472888
[32] Lianjie Cao and Puneet Sharma. 2021. Co-Locating Containerized Workload Using Service Mesh Telemetry. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies* (Virtual Event, Germany) *(CoNEXT '21)*. Association for Computing Machinery, New York, NY, USA, 168–174. https://doi.org/10.1145/3485983.3494867
[33] Jingrong Chen, Yongji Wu, Shihan Lin, Yechen Xu, Xinhao Kong, Thomas Anderson, Matthew Lentz, Xiaowei Yang, and Danyang Zhuo. 2023. Remote Procedure Call as a Managed System Service. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 141–159. https://www.usenix.org/conference/nsdi23/presentation/chen-jingrong
[34] Qiong Chen, Jianmin Qian, Yulin Che, Ziqi Lin, Jianfeng Wang, Jie Zhou, Licheng Song, Yi Liang, Jie Wu, Wei Zheng, Wei Liu, Linfeng Li, Fangming Liu, and Kun Tan. 2024. YuanRong: A Production General-purpose Serverless System for Distributed Applications in the Cloud Experience Track. In *Proceedings of the ACM SIGCOMM 2024 Conference (SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA.
[35] C. Cowan, F. Wagle, Calton Pu, S. Beattie, and J. Walpole. 2000. Buffer overflows: attacks and defenses for the vulnerability of the decade. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, Vol. 2. 119–129 vol.2. https://doi.org/10.1109/DISCEX.2000.821514
[36] Drew Dean and Alan J Hu. 2004. Fixing races for fun and profit: how to use access (2).. In *USENIX security symposium*. 195–206.
[37] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 467–481. https://doi.org/10.1145/3373376.3378512

[38] Riley Ryan Evtyushkin, Dmitry, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 693–707. https://doi.org/10.1145/3173162.3173204

[39] Henrique Fingler, Amogh Akshintala, and Christopher J. Rossbach. 2019. 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* (Hangzhou, China) *(APSys '19)*. Association for Computing Machinery, New York, NY, USA, 23–30. https://doi.org/10.1145/3343737.3343750

[40] Stefano Garzarella. 2019. How to measure the boot time of a Linux VM with QEMU/KVM. https://stefano-garzarella.github.io/posts/2019-08-24-qemu-linux-boot-time/. [online].

[41] Thomas Graf. 2024. How eBPF will solve Service Mesh – Goodbye Sidecars. https://isovalent.com/blog/post/2021-12-08-ebpf-servicemesh/. [online].

[42] Aaron Grattafiori. 2016. Understanding and hardening linux containers. *Whitepaper, NCC Group* (2016).

[43] Jinyu Gu, Hao Li, Wentai Li, Yubin Xia, and Haibo Chen. 2022. EPK: Scalable and Efficient Memory Protection Keys. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 609–624. https://www.usenix.org/conference/atc22/presentation/gu-jinyu

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

[45] Jesse Hertz. 2016. Abusing privileged and unprivileged linux containers. *Whitepaper, NCC Group* 48 (2016).

[46] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The EXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies* (Heraklion, Greece) *(CoNEXT '18)*. Association for Computing Machinery, New York, NY, USA, 54–66. https://doi.org/10.1145/3281411.3281443

[47] Darby Huye, Yuri Shkuro, and Raja R. Sambasivan. 2023. Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 419–432. https://www.usenix.org/conference/atc23/presentation/huye

[48] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. 2014. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 445–458. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/hwang

[49] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 489–502. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong

[50] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 152–166. https://doi.org/10.1145/3445814.3446701

[51] Chao Jin, Zili Zhang, Xingyu Xiang, Songyun Zou, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. Ditto: Efficient Serverless Analytics with Elastic Parallelism. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 406–419. https://doi.org/10.1145/3603269.3604816

[52] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. arXiv:1902.03383 [cs.OS]

[53] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, and Adam Barker. 2023. How Does It Function? Characterizing Long-term Trends in Production Serverless Workloads. In *Proceedings of the 2023 ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '23)*. Association for Computing Machinery, New York, NY, USA, 443–458. https://doi.org/10.1145/3620678.3624783

[54] Konstantinos Kallas, Haoran Zhang, Rajeev Alur, Sebastian Angel, and Vincent Liu. 2023. Executing Microservice Applications on Serverless, Correctly. *Proc. ACM Program. Lang.* 7, POPL, Article 13 (jan 2023), 29 pages. https://doi.org/10.1145/3571206

[55] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757* (2018).

[56] Avi Kivity. 2019. Building efficient I/O intensive applications with Seastar. https://github.com/CoreCppIL/CoreCpp2019/blob/master/Presentations/Avi_Building_efficient_IO_intensive_applications_with_Seastar.pdf. [online].

[57] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. 2014. OSv—Optimizing the Operating System for Virtual Machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 61–72. https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity

[58] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. 2021. Unikraft: Fast, Specialized Unikernels the Easy Way. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 376–394. https://doi.org/10.1145/3447786.3456248

[59] Stefan Lankes, Simon Pickartz, and Jens Breitbart. 2016. HermitCore: A Unikernel for Extreme Scale Computing. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers* (Kyoto, Japan) *(ROSS '16)*. Association for Computing Machinery, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/2931088.2931093

[60] Julia Lawall and Gilles Muller. 2018. Coccinelle: 10 Years of Automated Evolution in the Linux Kernel. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 601–614. https://www.usenix.org/conference/atc18/presentation/lawall

[61] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. 2022. FlexOS: Towards Flexible OS Isolation. In *Proceedings of the 27th ACM International Conference on Architectural Support for*

Programming Languages and Operating Systems (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 467–482. https://doi.org/10.1145/3503222.3507759

[62] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. 2019. Socksdirect: Datacenter Sockets Can Be Fast and Compatible. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 90–103. https://doi.org/10.1145/3341302.3342071

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

[64] Hao Li, Yihan Dang, Guangda Sun, Guyue Liu, Danfeng Shan, and Peng Zhang. 2023. LemonNFV: Consolidating Heterogeneous Network Functions at Line Speed. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1451–1468. https://www.usenix.org/conference/nsdi23/presentation/li-hao

[65] Suyi Li, Wei Wang, Jun Yang, Guangzhen Chen, and Daohe Lu. 2023. Golgi: Performance-Aware, Resource-Efficient Function Scheduling for Serverless Computing. In *Proceedings of the 2023 ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '23)*. Association for Computing Machinery, New York, NY, USA, 32–47. https://doi.org/10.1145/3620678.3624645

[66] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. 2022. Help Rather Than Recycle: Alleviating Cold Startup in Serverless Computing Through Inter-Function Container Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 69–84. https://www.usenix.org/conference/atc22/presentation/li-zijun-help

[67] Linux Programmer's Manual. 2024. tc-bpf(8). https://man7.org/linux/man-pages/man8/tc-bpf.8.html. [online].

[68] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 412–426. https://doi.org/10.1145/3472883.3487003

[69] Costin Lupu, Andrei Albiundefinedoru, Radu Nichita, Doru-Florin Blânzeanu, Mihai Pogonaru, Răzvan Deaconescu, and Costin Raiciu. 2023. Nephele: Extending Virtualization Environments for Cloning Unikernel-Based VMs. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 574–589. https://doi.org/10.1145/3552326.3587454

[70] 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. 2015. Jitsu: Just-In-Time Summoning of Unikernels. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 559–573. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/madhavapeddy

[71] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 461–472. https://doi.org/10.1145/2451116.2451167

[72] Anil Madhavapeddy and David J. Scott. 2013. Unikernels: Rise of the Virtual Library Operating System: What If All the Software Layers in a Virtual Appliance Were Compiled within the Same Safe, High-Level Language Framework? *Queue* 11, 11 (dec 2013), 30–44. https://doi.org/10.1145/2557963.2566628

[73] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 218–233. https://doi.org/10.1145/3132747.3132763

[74] Felix Moebius, Tobias Pfandzelter, and David Bermbach. 2024. Are Unikernels Ready for Serverless on the Edge? arXiv:2403.00515 [cs.DC] https://arxiv.org/abs/2403.00515

[75] Jeffrey C Mogul and Kadangode K Ramakrishnan. 1997. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems* 15, 3 (1997), 217–252.

[76] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. 2021. OFC: an opportunistic caching system for FaaS platforms. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 228–244. https://doi.org/10.1145/3447786.3456239

[77] Djob Mvondo, Alain Tchana, Renaud Lachaize, Daniel Hagimont, and Noël De Palma. 2020. Fine-Grained Fault Tolerance for Resilient pVM-Based Virtual Machine Monitors. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 197–208. https://doi.org/10.1109/DSN48063.2020.00037

[78] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. 2020. RedLeaf: Isolation and Communication in a Safe Operating System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 21–39. https://www.usenix.org/conference/osdi20/presentation/narayanan-vikram

[79] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. 2019. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 241–254. https://www.usenix.org/conference/atc19/presentation/park-soyeon

[80] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. 2015. The Design and Implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 117–130.

[81] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. 2011. Rethinking the Library OS from the Top Down. 39, 1 (mar 2011), 291–304. https://doi.org/10.1145/1961295.1950399

[82] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and K. K. Ramakrishnan. 2022. SPRIGHT: Extracting the Server from Serverless Computing! High-Performance EBPF-Based Event-Driven, Shared-Memory Processing. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 780–794. https://doi.org/10.1145/3544216.3544259

[83] Donald Ray and Jay Ligatti. 2012. Defining Code-Injection Attacks. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA,

179–190. https://doi.org/10.1145/2103656.2103678

[84] Luigi Rizzo. 2012. netmap: A Novel Framework for Fast Packet I/O. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 101–112. https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo

[85] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. 2021. Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 122–137. https://doi.org/10.1145/3472883.3486974

[86] Alireza Sahraei, Soteris Demetriou, Amirali Sobhgol, Haoran Zhang, Abhigna Nagaraja, Neeraj Pathak, Girish Joshi, Carla Souza, Bo Huang, Wyatt Cook, Andrii Golovei, Pradeep Venkat, Andrew Mcfague, Dimitrios Skarlatos, Vipul Patel, Ravinder Thind, Ernesto Gonzalez, Yun Jin, and Chunqiang Tang. 2023. XFaaS: Hyperscale and Low Cost Serverless Functions at Meta. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 231–246. https://doi.org/10.1145/3600006.3613155

[87] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. 2023. ServiceRouter: Hyperscale and Minimal Cost Service Mesh at Meta. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 969–985. https://www.usenix.org/conference/osdi23/presentation/saokar

[88] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. 2021. CubicleOS: A Library OS with Software Componentisation for Practical Isolation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 546–558. https://doi.org/10.1145/3445814.3446731

[89] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. 2022. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 936–952. https://www.usenix.org/conference/usenixsecurity22/presentation/schrammel

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

[91] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[92] Giuseppe Siracusano, Roberto Bifulco, Simon Kuenzer, Stefano Salsano, Nicola Blefari Melazzi, and Felipe Huici. 2016. On the Fly TCP Acceleration with Miniproxy. In *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization* (Florianopolis, Brazil) *(HotMIddlebox '16)*. Association for Computing Machinery, New York, NY, USA, 44–49. https://doi.org/10.1145/2940147.2940149

[93] Enge Song, Yang Song, Lu Chengyun, Tian Pan, Shaokai Zhang, Jianyuan Lu, Jiangu Zhao, Xining Wang, Xiaomin Wu, Minglan Gao, Zongquan Li, Ziyang Fang, Biao Lyu, Pengyu Zhang, Rong Wen, Li Yi, Zhigang Zong, and Shunmin Zhu. 2024. Canal Mesh: A Cloud-Scale Sidecar-Free Multi-Tenant Service Mesh Architecture. In *Proceedings of the ACM SIGCOMM 2024 Conference (SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA.

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

[95] Bo Tan, Haikun Liu, Jia Rao, Xiaofei Liao, Hai Jin, and Yu Zhang. 2020. Towards Lightweight Serverless Computing via Unikernel as a Function. In *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*. 1–10. https://doi.org/10.1109/IWQoS49365.2020.9213020

[96] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. 2014. Cooperation and Security Isolation of Library OSes for Multi-Process Applications. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands) *(EuroSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 9, 14 pages. https://doi.org/10.1145/2592798.2592812

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

[98] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. 2020. Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications. *ACM Comput. Surv.* 53, 1, Article 16 (feb 2020), 36 pages. https://doi.org/10.1145/3371038

[99] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. 2020. InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 267–281. https://www.usenix.org/conference/fast20/presentation/wang-ao

[100] Dan Williams and Ricardo Koller. 2016. Unikernel Monitors: Extending Minimalism Outside of the Box. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. USENIX Association, Denver, CO. https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/williams

[101] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 719–732. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[102] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. 2016. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 43–56. https://www.usenix.org/conference/atc16/technical-sessions/presentation/yasukata

[103] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2019. The True Cost of Containing: A gVisor Case Study. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. USENIX Association, Renton,

WA. https://www.usenix.org/conference/hotcloud19/presentation/young

[104] Hanfei Yu, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. 2024. RainbowCake: Mitigating Cold-starts in Serverless with Layer-wise Container Caching and Sharing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 335–350. https://doi.org/10.1145/3617232.3624871

[105] Jason Zhijingcheng Yu, Shweta Shinde, Trevor E. Carlson, and Prateek Saxena. 2022. Elasticlave: An Efficient Memory Model for Enclaves. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 4111–4128. https://www.usenix.org/conference/usenixsecurity22/presentation/yu-jason

[106] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. 2023. Following the Data, Not the Function: Rethinking Function Orchestration in Serverless Computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1489–1504. https://www.usenix.org/conference/nsdi23/presentation/yu

[107] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. 2021. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 195–211. https://doi.org/10.1145/3477132.3483569

[108] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfen Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. 2018. KylinX: A Dynamic Library Operating System for Simplified and Efficient Cloud Virtualization. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 173–186. https://www.usenix.org/conference/atc18/presentation/zhang-yiming

[109] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. 2023. Electrode: Accelerating Distributed Protocols with eBPF. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1391–1407. https://www.usenix.org/conference/nsdi23/presentation/zhou

[110] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, Xiongchun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, and Ratul Mahajan. 2022. Dissecting Service Mesh Overheads. arXiv:2207.00592 [cs.DC]