POLITECNICO DI TORINO
Repository ISTITUZIONALE

SURE: A High-Performance, Efficient, and Secure Serverless Framework Based on Unikernels

Original
SURE: A High-Performance, Efficient, and Secure Serverless Framework Based on Unikernels / Parola, F., Qi, S., Narappa, A.B., Ramakrishnan, K.K., Risso, F.. - In: IEEE TRANSACTIONS ON CLOUD COMPUTING. - ISSN 2168-7161. - 14:3(2026), pp. 2136-2153. [10.1109/TCC.2026.3697819]

Availability:
This version is available at: 11583/3011677 since: 2026-06-04T10:51:02Z

Publisher:
IEEE

Published
DOI:10.1109/TCC.2026.3697819

Terms of use:

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

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2026 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

19 November 2026

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2026.3697819

SURE: A High-performance, Efficient, and Secure Serverless Framework based on Unikernels

Federico Parola[1], Shixiong Qi[2], Anvaya B. Narappa[3], K. K. Ramakrishnan[3], *Life Fellow, IEEE*, Fulvio Risso[1]
[1]Politecnico di Torino, [2]University of Kentucky, [3]University of California, Riverside

***Abstract*—Current serverless platforms face substantial overhead from kernel-based networking and per-function sidecars. In addition, container-based runtimes suffer from excessive startup times and limited isolation. These limitations motivate the need for a more efficient and secure design.**

We present SURE, a unikernel-based serverless framework that combines rapid function startup with a high-performance, secure data plane. SURE enables distributed zero-copy communication through seamless integration of a userspace zero-copy TCP/IP stack (Z-stack) with intra-node shared memory processing. To eliminate the inefficiency of per-function sidecars, SURE introduces a lightweight library-based sidecar, which reduces CPU overhead by over two orders of magnitude compared to traditional userspace sidecars.

For security, SURE leverages Intel's Memory Protection Keys (MPK) to enforce fine-grained, page-level isolation in the shared memory data plane and to isolate the Trusted Computing Base (TCB) components in the function runtime (e.g., library-based sidecar, scheduler, etc) from untrusted user code. This is complemented by memory-pool-based security domains that isolate each function chain, ensuring scalability to large deployments. SURE further integrates a combination of binary inspection, W⊕X enforcement, and TCB-page blacklisting to prevent MPK privilege escalation within the single-address-space unikernel.

These combined efforts create a more secure and efficient data plane with improved performance. Our evaluation shows that SURE improves throughput by 6×-8× compared to SPRIGHT, a high-performance serverless platform.

***Index Terms*—Serverless computing, unikernels, shared memory, memory protection keys, sidecar**

I. Introduction

Cloud computing services are evolving to composable, *loosely-coupled* microservices [1], where each microservice can be independently developed, deployed, and scaled. Serverless computing, or Function-as-a-Service (FaaS [2]), naturally fits this paradigm. Its event-driven execution model, fine-grained billing, and elastic scaling simplify the management of complex microservice applications [2], [3]. In particular, serverless platforms enable microservices to be organized into a "function chain" following call graph dependencies [1], facilitating the transition of production-scale microservice workloads to serverless deployments [4].

A longstanding challenge in serverless computing is the *cold-start* latency, where the first invocation of a function incurs significant delays due to runtime initialization [5]. Cold-start latency degrades application responsiveness, making the rapid startup of functions essential. ***Unikernels*** are particularly attractive in this context because they boot up in milliseconds, enabling rapid cold-start, and responsive autoscaling to handle bursts [6], [7]. Unlike traditional (full-size) VMs or containers, unikernels provide a single-address-space runtime with only a minimal set of libraries and drivers [8]–[11]. This streamlined design yields faster boot times, a smaller Trusted Computing Base (TCB), and potentially fewer vulnerabilities [11], [12]. Recent systems further extend unikernel compatibility layers [13] and integrate with container orchestration platforms such as Kubernetes [14], making more practical unikernel deployments possible [15]. However, fast startup alone is not sufficient to support microsecond-scale serverless function execution [16]. Naive unikernel adoption still faces three key limitations: (1) it inherits the same inter-function networking bottlenecks as containers, especially for cross-node communication; (2) per-function sidecars remain costly; and (3) single-address-space unikernels provide weak isolation between untrusted code and privileged runtime components.

Our Approach: This paper presents **SURE**,[1] a unikernel-based serverless framework that combines fast startup, low-latency inter-function networking, lightweight library-based sidecar, and stronger isolation. SURE deploys each function as a lightweight unikernel VM, combining rapid startup with VM-grade isolation at the function level.

For high-performance networking, SURE introduces a distributed zero-copy data plane that: (1) leverages shared memory processing for intra-node communication, and (2) extends zero-copy semantics across nodes via *Z-stack*, a userspace full-fledged FreeBSD TCP/IP stack integrated with shared memory and coordinated by a per-node SURE gateway (§IV-E). We introduce a per-node SURE gateway to consolidate protocol processing for all functions co-located on the same node. To efficiently coordinate shared memory processing among communicating functions, SURE introduces inter-VM event-driven signaling to facilitate lightweight descriptor exchanges (§IV-B). SURE further enforces *back-pressure* between communicating functions to ensure the reliability of shared memory processing (§IV-C).

SURE re-evaluates the sidecar model. Instead of a standalone userspace process, the sidecar is linked directly into the unikernel as a library. This eliminates costly kernel-userspace boundary crossings, enables efficient function-to-sidecar calls, and supports eBPF-like event hooks without the associated verifier restrictions. As a result, SURE achieves full L7 payload visibility and a fully functional service mesh in a unikernel runtime (§IV-F).

[1]We call our work **SURE** for "**S**ecure **U**nikernels that are **R**apid and **E**fficient." This work was first published in the ACM SoCC 2024 [17]. It has been extended here with additional design details and experimental results.

Authorized licensed use limited to: Politecnico di Torino. Downloaded on June 15,2026 at 08:32:37 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2026.3697819

SURE pays particular attention to security vulnerabilities that may arise from shared memory processing and single-address-space unikernels (details in §III-C). To complement high performance with scalable isolation, SURE introduces two layers of isolation. First, it groups functions within the same chain into a **security domain**. Each security domain is backed by a private memory pool that is only visible to the functions from the same domain. Second, within each unikernel, SURE enforces **page-level** memory isolation using Intel's Memory Protection Key (MPK [18]). We introduce an MPK-based *call gate* abstraction to mediate controlled access to shared memory buffers and privileged TCB components within the unikernel runtime (§V-C), preventing untrusted code from violating protection boundaries.

MPK has a hardware scalability limit of only 16 distinct protection keys (pkeys) per thread. To remain efficient and scalable, SURE employs a hybrid access privilege management design that uses just *four* keys per thread. One key is dedicated to managing access privileges for all pages belonging to the TCB components (such as the sidecar or I/O subsystems), which are frequently accessed by user code through privileged calls. For these fast-path operations, SURE employs the lightweight *WRPKRU* instruction (20-30 cycles) to temporarily enable and revoke access to TCB pages, ensuring protection without adding measurable delay to the critical execution path.

The remaining three keys are configured with fixed permissions (one as *read/write*, one as *read-only*, and one as *no-access*) and are used for fine-grained control of shared memory buffer pages. Here, we implement a PTE update mechanism, which the call gate uses to alternate between the three keys in the PTEs of the buffer pages. Although PTE update is slower (around 300 cycles) than *WRPKRU*, it is well-suited for managing the less frequent privilege transitions of shared memory buffers, which lie outside the latency-critical path.

Finally, SURE secures MPK enforcement inside the unikernel runtime to prevent MPK privilege escalation attacks unique to single-address-space designs (§VI). Specifically, SURE identifies concrete attack scenarios, such as unmediated *WRPKRU*/*XSAVE*/*XRSTOR* execution, PTE tampering, and thread context corruption. We introduce complementary defenses including binary inspection, W⊕X enforcement, call-gate mediation, and blacklisting of protected TCB pages. These mechanisms ensure that MPK-based isolation remains reliable in a single-address-space unikernel environment.

Summary of Contributions.

1) SURE extends zero-copy communication across nodes via Z-stack, integrating zero-copy TCP/IP processing with the intra-node shared memory data plane. This goes 6×-8× beyond the high-performance serverless data plane design of SPRIGHT, which is limited to intra-node shared memory communication [19].
2) SURE's library-based sidecar design exploits the unikernel's single-address-space, yielding over ***100***× CPU cycle savings and up to 16× performance improvement over individual userspace sidecars.
3) A scalable memory isolation framework combining MPK-based page-level protection with memory-pool-based security domains between function chains.
4) SURE introduces a hybrid access privilege management mechanism to overcome MPK's per-thread hardware limit.
5) SURE provides securing strategies for MPK privilege escalation in a single-address-space environment.

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

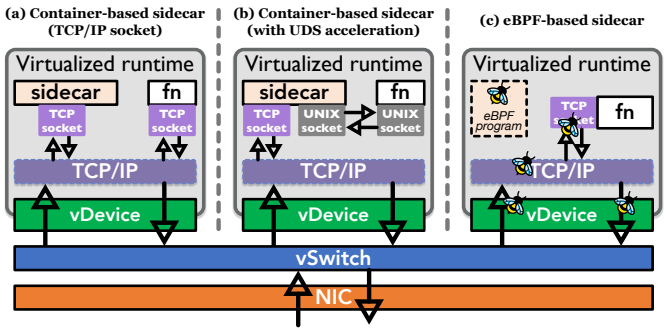


Fig. 1: An abstract diagram of the serverless data plane. We list existing sidecar designs: (a) container-based sidecar using TCP/IP socket [20], (b) container-based sidecar using Unix domain socket acceleration [21], (c) eBPF-based sidecar [19].

II. Background and Motivation

We begin with a background of serverless computing, highlighting the challenges that remain, and then motivate the design of our system.

A. Background: Networking, Service Mesh, and Isolation in Serverless Computing

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 distinct functions of a chain. Essential components include a virtual switch (vSwitch) for L2 forwarding; a network protocol stack (*i.e.,* TCP/IP) for protocol processing; and virtual device interfaces to interconnect functions with the vSwitch. However, kernel-based networking dominates data-plane overhead for function chains, contributed by data copies, context switches, interrupts, protocol processing, and serialization overheads [19]. These costs persist even when functions are co-located on the same node.

(2) A *service mesh* that transparently enables observability, traffic management, and access control via per-function sidecars [21]. Traditional service meshes attach a per-function userspace sidecar (often as a distinct container) that communicates with the function over the TCP/IP stack [21] (Fig. 1 (a)). This results in significant overhead due to data copies, serialization, and redundant protocol processing [19], [21]. Acceleration can help, using Unix domain sockets (UDS) to redirect the payload between the user function sockets and the sidecar (Fig. 1 (b)), bypassing protocol processing. But, this still incurs data copy and serialization/deserialization overheads between the function and sidecar.

(3) A *virtualized runtime* that provides fine-grained isolation at the function level. Serverless platforms operate in multi-

Authorized licensed use limited to: Politecnico di Torino. Downloaded on June 15,2026 at 08:32:37 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2026.3697819

tenant cloud environments where isolation is critical for protecting sensitive user data and avoiding resource contention. Containers provide lightweight virtualization and reduced startup time, but their state is still managed by the host kernel, leaving them more vulnerable to kernel-level exploits [22]. As a result, commercial serverless providers (*e.g.,* AWS Lambda) increasingly rely on VM-based isolation, where each function runs in its own VM with a dedicated kernel [23]. However, VMs with a full-fledged guest OS introduce high startup latency, undermining application responsiveness [6], [9]. Thus, there is a tradeoff between isolation and startup latency.

B. Prior Work and Challenges

1) Unikernel-based runtimes: Unikernels follow the LibOS model [24]: an application is statically linked with only the OS components it needs and runs in a single address space [8]. This yields a lightweight guest runtime with fast startup, small memory footprint, and low overhead. When deployed inside VMs, unikernels further benefit from VM-level isolation, making the combination particularly attractive for serverless computing [9]. However, the same single-address-space design places privileged runtime components and untrusted user code in the same address space, so the runtime TCB must be protected from untrusted code.

Design Implication#1: Strong intra-unikernel isolation is needed to protect the runtime TCB components from untrusted user code.

2) Cost of kernel-based inter-function networking: Prior work [4], [19], [25] accelerates function chains by bypassing the kernel and enabling *zero-copy* shared memory communication between functions. However, shared memory communication is confined to a single node. Cross-node communication still falls back to kernel-based networking [19], [25]. While locality-aware placement [26] can reduce cross-node traffic, it is not always feasible for large-scale production workloads [1]. Functions can also be resource-intensive and need to be spread across multiple nodes.

Design Implication#2: Zero-copy networking should be extended across nodes to reduce kernel overheads and achieve high performance for distributed function chains.

3) Shared memory is considered unsafe: Shared memory is often viewed as being unsafe because it can expose applications to information leakage and corruption attacks [27]–[29]. Prior serverless systems therefore either assume that co-located functions trust each other [25], or isolate memory only at the granularity of a function group or tenant [19]. Such assumptions are insufficient for SURE, where buggy or malicious code within the same tenant may still pose a risk to other functions and therefore requires finer-grained protection.

Design Implication#3: Shared memory must provide access control at the individual buffer level to combine high performance with fine-grained isolation. Secure channels built on shared memory must prevent unauthorized access, even within the same chain.

4) Cost of individual userspace sidecars: Recent designs replace sidecars with in-kernel eBPF programs [19] (Fig. 1 (c)). By executing the sidecar logic within the kernel, such designs avoid extra userspace–kernel crossings and inter-process communication overheads, and their event-driven execution model aligns well with serverless workloads [19]. However, eBPF is not a complete solution for unikernel environments. Critical hooks (*e.g.,* SOCK_MSG) are not available, limiting deployability, and eBPF-based sidecars do not provide the full L7 payload visibility needed by service meshes [30]. Its constrained programming model further limits flexibility.

Design Implication#4: An ideal sidecar design should provide full L7 visibility, while retaining the key benefits of eBPF-based sidecars: event-driven execution and avoiding unnecessary kernel–userspace crossings.

III. Overall System Architecture of SURE

Fig. 2 depicts the architecture of SURE, including the following core building blocks: **(1) Unikernel-based VM runtime.** Each function runs inside a lightweight unikernel-based VM (a SURE VM). Within the VM, user code and user data coexist with a small set of runtime components that form the SURE TCB (*e.g.,* scheduler, booter, and page-table management). While each VM typically hosts a single function, multiple mutually trusted functions may also share a VM. **(2) MPK-based call gates.** SURE utilizes an MPK-based call-gate abstraction in the unikernel runtime to mediate controlled access to protected shared memory buffers and TCB components (§V-C). **(3) Security domains.** Each worker node hosts one or more secure domains. Each domain groups a set of SURE VMs that share a local memory pool. The hypervisor isolates security domains by ensuring that their memory pools are strictly separated, while also managing the VM lifecycle and initialization of the shared memory. Refer to Appendix-C in the supplementary material for additional details on the management of security domains. **(4) Per-node SURE gateway.** Each node runs a single SURE gateway, which executes Z-stack, manages DMA interactions with the NIC, and multiplexes/demultiplexes communication for co-located VMs. **(5) Library-based sidecar.** SURE embeds the sidecar directly as a library inside a unikernel (§IV-F).

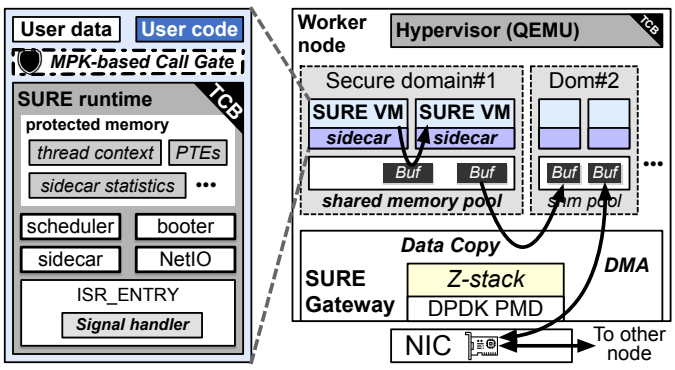


Fig. 2: The overall architecture of SURE.

Authorized licensed use limited to: Politecnico di Torino. Downloaded on June 15,2026 at 08:32:37 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2026.3697819

Function–sidecar interactions become internal function calls rather than using inter-process communication.

A. Inter-function Communication Overview

SURE supports three communication cases. First, when two functions are co-located on the same node and belong to the same security domain, they communicate through the shared memory pool of that domain. In this case, descriptors are exchanged while payloads remain in shared memory, enabling zero-copy intra-node communication.

Second, when communication crosses security domains on the same node, shared memory is no longer allowed. Since each security domain is backed by a separate private memory pool, directly sharing buffers across domains would weaken the isolation boundary between potentially adversarial tenants. Therefore, SURE performs an explicit data copy between the corresponding memory pools (via the SURE gateway) for inter-domain communication. This is a deliberate design choice driven by SURE's trust model.

Third, for cross-node communication, SURE uses the SURE gateway together with Z-stack, a userspace TCP/IP stack that preserves the zero-copy semantics. The SURE gateway is deployed as a privileged service that can access the shared memory pool of any security domain, allowing it to bridge inter-node traffic with SURE's shared memory data plane. This design enables efficient cross-node communication while preserving the isolation boundary between security domains. §IV-E describes the detailed TX/RX paths.

B. SURE's Trust Model

SURE assumes an asymmetric trust model common in public serverless clouds: Users trust the serverless infrastructure (*i.e.,* SURE VMs and hypervisor), but the infrastructure does not trust user applications, as they may contain security vulnerabilities, *e.g.,* bugs or adversarial code. Further, functions are exposed to threats from other tenants in the same cloud who may be potentially adversarial users. As such, SURE treats the hypervisor and associated toolchains (*e.g.,* emulation of hardware devices and peripherals required by VMs) as part of the TCB (as shown in Fig. 2). Inside each VM, SURE further establishes another layer of trust, enforcing intra-unikernel isolation between the untrusted user code and trusted runtime components (scheduler, booter, sidecar, NetIO lib, and other OS modules in the unikernel TCB) (details in §V).

C. 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) Shared memory vulnerabilities.** 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 [27], buffer overflows [31], or injection attacks [32]. This risk is particularly pronounced in a public cloud environment, which is shared by functions from different users. In addition, buggy (even if not malicious) code in user functions can accidentally and improperly manipulate shared data. **(2) Intra-unikernel vulnerabilities.** SURE's function runtime (see Fig. 2), including the library-based sidecar and many other TCB modules, is part of the serverless infrastructure and requires 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.

D. Overview of Isolation Framework in SURE

Beyond the VM-based isolation, SURE introduces three isolation mechanisms that protect shared memory processing, enforce inter-function access control, and separate user code from trusted runtime components within the unikernel.

Name-based security domains. When deploying a function, a "security domain" must be specified, *i.e.,* the name of the POSIX shared memory backend (§IV-A). Mutually trusted SURE VMs (typically from the same user) are assigned to the same domain, where they share a dedicated memory pool.

Access control with sidecar and SURE gateway. Within a security domain, ownership transfer of shared memory buffers occurs via descriptor exchanges. Before granting access, SURE verifies the receiver's eligibility, thus preventing unauthorized access. The library-based sidecar enforces traffic filtering using whitelists of allowed peers, discarding unauthorized descriptors. For cross-node communication, the per-node SURE gateway integrates access control into Z-stack's `ipfw` firewall (adopted from FreeBSD) to block unauthorized traffic.

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. SURE employs Intel's MPK [18] to enforce page-level access control at low cost. MPK protects the sidecar and other LibOS components from faulty or malicious user code, while allowing selective access to shared memory pages or TCB components for VMs that are authorized to participate in a data exchange. To support this securely, SURE provides libraries (*e.g.,* NetIO, sidecar) wrapped with MPK-based call gates, exposing standardized APIs that reduce the risk of privilege escalation and allow user code to interact transparently with the data plane.

IV. Distributed Zero-Copy Data Plane in SURE

In this section, we describe the core building blocks of SURE's data plane, including the shared memory, inter-VM signaling, reliable message delivery, Z-stack, SURE gateway, and the library-based event-driven sidecar. For additional design details on failure mitigation, refer to Appendix-D of the supplementary material.

A. Shared Memory Management

Each security domain is provisioned with a private memory pool, created as a POSIX shared memory backend in the host

Authorized licensed use limited to: Politecnico di Torino. Downloaded on June 15,2026 at 08:32:37 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2026.3697819

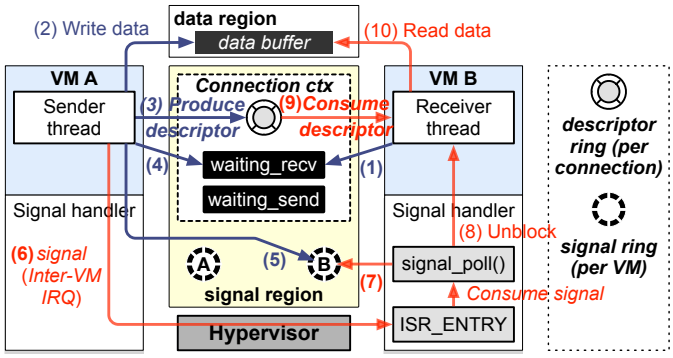


Fig. 3: Inter-VM event-driven signaling in SURE.

file system (under `/dev/shm/`). Pools are uniquely named, similar to DPDK's file-prefix mechanism, so memory metadata cannot collide across domains. At VM launch, SURE employs QEMU's Inter-VM Shared Memory Device (*ivshmem*) [33] to map the corresponding shared memory backend into the VM. To prevent accidental reuse, the host enforces file permissions on used pools. As a result, shared memory is visible only to VMs within the same security domain. VMs in one domain cannot access another domain's pool. Inside each VM, applications access the pool by `mmap()` on the device's PCI BAR, ensuring that all VMs in a domain see the same physical pages.

A memory pool consists of two regions: a *data region* and a *signal region*. The *data region* comprises pre-allocated buffers that hold packet payload.[2] The *signal region* supports lightweight descriptor exchanges by enabling inter-VM event-driven signaling. Each SURE VM has a multiple-producer, single-consumer *signal ring* in the *signal region* to receive the wake-up signal from peer VMs (Fig. 3). Multiple producers enqueue notifications into the ring using the standard lock-free multi-producer ring discipline.[3] Within the *signal region*, SURE maintains a per-connection context that encapsulates all state required for communication between a sender thread and a receiver thread. Each connection context includes: (i) a single-producer, single-consumer descriptor ring dedicated to that sender–receiver thread pair, which tracks message descriptors exchanged between them; and (ii) two synchronization flags, `waiting_send` and `waiting_recv`, which coordinate back-pressure and blocking operations.

B. Inter-VM Event-driven Signaling

A dedicated *signal handler* thread in each VM processes *pending* wake-up signals in batches, triggered by inter-VM interrupts raised by the hypervisor [33] and dispatched through the VM's Interrupt Service Routine (ISR).

Instead of busy-polling its *descriptor ring*, a receiver thread in SURE can perform a blocking receive (steps 1-5 in Fig. 3). If descriptors are available, the receiver thread consumes them directly without blocking. When no descriptors are available for the receiver to consume, the receiver thread registers its thread pointer in `waiting_recv` and suspends execution. When the sender thread later enqueues a new descriptor (steps 2, 3), it checks `waiting_recv` (step 4), and if the receiver thread is blocked, writes the receiver's pointer into the receiver's *signal ring* (step 5). The sender thread then issues an inter-VM interrupt, which causes the receiver's ISR to invoke `signal_poll()`, consume the signal, and wake the blocked receiver thread to process the newly arrived descriptor (steps 6-10 in Fig. 3).

C. Reliable Data Transfer and Back-pressure

Entries in the *descriptor ring* are consumed in FIFO order, ensuring in-order delivery. Reliability is enforced through *back-pressure*: when a receiver's *descriptor ring* becomes full, the sender cannot enqueue additional descriptors. Instead, the sender registers its thread pointer in `waiting_send`, and blocks. This mechanism guarantees that descriptors are never dropped due to the ring overflowing. When the receiver later consumes descriptors and the *descriptor ring* becomes available, it checks `waiting_send`. If a sender is registered, the receiver signals the sender to wake it up. We use Z-stack to guarantee reliable data transfer across nodes.

D. Connection and Routing Management

SURE supports provisioned concurrency by allowing multiple threads within a SURE VM to multiplex warm function instances, reducing cold starts [34]. To avoid head-of-line blocking, a dedicated connection can be assigned to each sender-receiver thread pair. Threads are differentiated by distinct connections, so data transmission requires an established connection. SURE exposes a familiar socket-like API (`listen()`, `accept()`, `connect()`, `close()`) to manage connection lifecycles. Connection teardown with `close()` awakens any blocked threads and safely recycles resources once both endpoints have closed.

Each security domain maintains an IP routing table stored in the *signal region*. Intra-domain routing maps the sender's IP 5-tuple to the correct connection context of the receiver, ensuring compatibility with inter-node IP routing. Routing tables are read-only to SURE VMs (enforced by call gates in §V-C) and are updated only by the hypervisor during connection establishment.

E. SURE gateway and Z-stack

A key inefficiency in most existing userspace protocol stacks (such as F-stack [35]) stems from their reliance on the

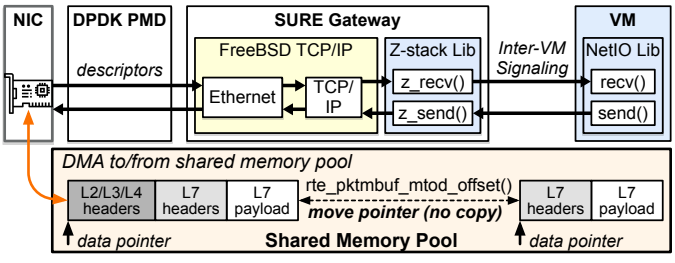


Fig. 4: Zero-copy inter-node communication in SURE.

[2]Applications in SURE can also allocate a large enough buffer to store the complete payload, avoiding the need to assemble and disassemble during shared memory data transfer.

[3]https://lwn.net/Articles/340400/

Authorized licensed use limited to: Politecnico di Torino. Downloaded on June 15,2026 at 08:32:37 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2026.3697819

POSIX socket interface. This interface requires copying data between an application's send/receive buffers and the socket buffers maintained by the TCP/IP stack. Prior work shows that such buffer copies can consume more than 50% of CPU cycles [36]. These copies are a legacy of kernel-userspace isolation, where explicit copying is necessary to protect kernel memory from direct application access. In SURE, where the entire data plane runs in userspace, such an isolation is redundant, and the copy can be avoided.

To this end, SURE integrates Z-stack into the SURE gateway (Fig. 4). Z-stack builds on F-stack [35], a FreeBSD-based TCP/IP stack integrated with DPDK's Poll Mode Driver (PMD [37]). The PMD moves packets between the NIC and memory via DMA entirely in userspace, thereby avoiding kernel-related overhead [38]. Within the Z-stack, packet buffers are passed directly between the protocol layers, with only lightweight header offset adjustments performed along the path (via `rte_pktmbuf_mtod_offset()`). Z-stack further provides two zero-copy APIs, `z_send()` and `z_recv()`.[4] The SURE gateway invokes these APIs to exchange descriptors with Z-stack, rather than copying payload data.

Zero-copy inter-node communication through SURE gateway. The transmit path (from a SURE VM to the SURE gateway) is straightforward. Because the privileged SURE gateway can access the shared memory pool of any security domain, it can directly provide the address of the outgoing shared buffer to the NIC through Z-stack and DPDK. The NIC transmits the payload via DMA without the extra copy.

The receive path (from SURE gateway to a SURE VM) is more challenging. To preserve zero-copy on receive, an incoming packet must be placed directly into the memory pool of the *target* security domain. This cannot be achieved by arbitrarily mixing receive buffers from different domains in the same NIC RX queue: since an RX queue is consumed in FIFO order, the NIC cannot guarantee that a buffer at the head of the queue belongs to the target domain for the arriving packet. Thus, a packet may be incorrectly DMAed into the wrong domain's pool, violating the intended isolation boundary.

To address this issue, SURE partitions NIC receive resources by security domain and configures the NIC to steer packets to the appropriate domain-specific RX queues based on packet-header information, e.g., via NIC flow steering (DPDK's `rte_flow`) together with RSS to balance load among RX queues within the same domain.[5] To scale beyond the limited number of physical RX queues, SURE further leverages SR-IOV to create multiple virtual functions (VFs), each provisioned with dedicated RX queues for a different security domain. This design DMAes the packet directly into a buffer of the target domain's memory pool. The SURE gateway then polls the packet descriptor via the PMD, performs protocol processing on that same buffer, and finally wakes up the target function through SURE's inter-VM event signaling mechanism, passing only the descriptor.[6]

[4] `z_send()` and `z_recv()` are *not* application-facing APIs in SURE. For SURE functions, the communication interface remains the NetIO APIs, i.e., `send()` and `recv()`. The choice between local shared memory communication and remote Z-stack communication is made transparently by the NetIO library. Refer to Appendix-A in the supplementary material for more details.

[5] This design aligns well with DPDK's RX model, where packet buffers are provisioned at an RX-queue granularity: each RX queue is configured with its own mempool in `rte_eth_rx_queue_setup()`, so different queues of the same port can draw buffers from different memory pools.

F. Library-based, Event-driven Sidecar

Following the philosophy of LibOSes, SURE integrates sidecar functionality directly as a library component, embedded within the application and invoked through lightweight function calls. This eliminates the need for a separate sidecar process, while still exposing extensible event-driven execution hooks that can host additional sidecar functions. Similar designs have proven effective in eBPF-based service meshes [19], offering improved resource efficiency compared to traditional container-based sidecars. Within SURE's unikernel, the library-based sidecar provides common service mesh capabilities, such as monitoring and traffic management, with full visibility into the L7 payload. Unlike prior library-based sidecar solutions (*e.g.,* ServiceRouter [40]), which offer little protection between application and sidecar logic, SURE safeguards its sidecar with MPK-based isolation (§III-D).

Event-driven hooks: To preserve transparency with respect to the serverless function, SURE pre-defines two hook points on both the receive (RX) path and transmit (TX) path of its NetIO library. Sidecar functions can be attached to these hooks and are invoked automatically when I/O events occur. Cloud providers can flexibly customize the sidecar by adding or removing functions from these hook sequences, tailoring behavior to events of interest.

Implementation of library-based sidecar: Following the methodology of the popular service mesh solution, Istio [20], each sidecar capability in SURE is encapsulated as a handler function, and multiple handlers are chained into a call sequence. We have implemented a set of common handlers, including request logging, metrics collection, rate limiting, and access control, that collectively provide the monitoring and traffic management features expected of a service mesh.

V. MPK-based Memory Isolation in SURE

Next, we describe how MPK privileges are managed in SURE's unikernel runtime. We then present MPK-based call gates, which allow untrusted user code to interact safely with privileged TCB components and protected shared memory.

A. A Primer on MPK (Memory Protection Key)

MPK is a hardware feature introduced in Intel server CPUs starting with the Skylake microarchitecture [41]. MPK provides lightweight, intra-process memory isolation by associating memory pages with one of up to 16 protection keys (pkeys). Each pkey is represented by a 4-bit identifier stored in the page table entry (PTE) of the page. In SURE, a "process" corresponds to a unikernel VM, so MPK effectively enforces fine-grained protection within each VM.

[6] Additional implementation details of Z-stack are described in our earlier work on Z-stack [39], including the design of `z_send()`/`z_recv()`.

Authorized licensed use limited to: Politecnico di Torino. Downloaded on June 15,2026 at 08:32:37 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2026.3697819

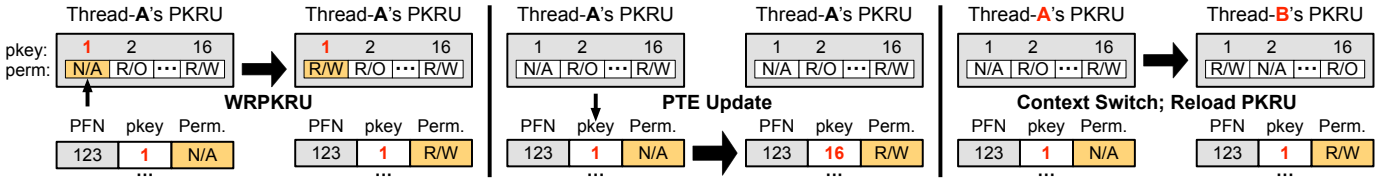


Fig. 5: Access privilege management with (Left) *WRPKRU* and (Middle) PTE update. (Right) `PKRU` is reloaded by thread context switch. Each thread has its own `PKRU` in the thread context to maintain its pkey permissions. (PFN: Page Frame Number.)

Access privileges for all pkeys are tracked in a per-thread 32-bit CPU register called `PKRU` (Protection Key Register User) [42]. For each pkey, `PKRU` maintains two permission bits (Fig. 5): (i) *Access Disable* (*AD*), which disables both read and write access; and (ii) *Write Disable* (*WD*), which blocks writes but allows reads. Thus, Read/Write (0, 0), Read-Only (0, 1), or No-Access (1, ×) [42]. The access privilege of a pkey takes effect once the key is assigned to the PTE of a page. During a context switch, the kernel saves the outgoing thread's `PKRU` value and restores the incoming thread's `PKRU`, ensuring that protection settings remain thread-specific and do not interfere across threads, as shown in Fig. 5 (Right).

B. Access Privilege Management with MPK

Access control with MPK can be managed in two ways.

A1: *WRPKRU*. The first approach uses the x86 instruction *WRPKRU*, which directly modifies the `PKRU` register of the calling thread to change the permissions of an existing pkey, as shown in Fig. 5 (Left). This operation is very lightweight (tens of cycles) and does not require kernel involvement, but it only changes the calling thread's view of the key, leaving other threads unaffected.

A2: PTE Update. The second approach changes access privileges by updating the pkey field in the PTE of the target page (Fig. 5 (Middle)). This effectively re-tags the page with a different pkey, thereby changing their protection at the process-wide level. Concretely, the runtime (1) updates the pkey field in the corresponding PTE, (2) invalidates the stale TLB entry for the modified page,[7] and then (3) allows subsequent access to proceed using the updated translation. While this approach is conceptually similar to inter-VM page remapping, they are quite different. SURE performs this operation entirely within the VM on the first-stage translation (guest page tables), whereas inter-VM remapping requires hypervisor-managed updates to the second-stage translation (*e.g.,* EPT on Intel). They are similar in that both require updating the PTE and invalidating the stale TLB entry before subsequent access can use the updated translation. We discuss this distinction further in Appendix-F of the supplementary material.

1) Performance Difference Between WRPKRU and PTE Update: We conducted a microbenchmark that maps a single page and repeatedly changes its accessibility using either *WRPKRU* or PTE update. We measure the average latency per operation and run millions of iterations to reduce noise. The benchmark pins the execution to one core. We fault the page before timing to avoid skewing the average value.

[7]SURE flushes only the TLB entry corresponding to the modified page rather than performing a full TLB flush.

Our results confirm previous reports [42], [43]: On the Intel Xeon Silver 4314 (2.40 GHz), *WRPKRU* completes in roughly 20-30 cycles (≈**8.3-12.5 ns**), while PTE update costs about 300 cycles (≈**126 ns**). The large gap arises from *WRPKRU* being a user-level instruction that updates only the calling thread's `PKRU` register, whereas PTE update requires the PTE modification and the TLB flush.

TABLE I: Tradeoffs of MPK Access Management Methods in SURE.

Method	Latency	Granularity	Scalability	Usage in SURE
WRPKRU	20-30 cycles	Coarse (per pkey)	Limited (16 keys/thread)	Protecting TCB pages: frequent access, minimal key usage
PTE update	300 cycles	Fine (per page)	Scales with page count	Isolating shared memory pages: inter-VM, fine-grained control

2) Scalability Difference Between WRPKRU and PTE Update: While *WRPKRU* is extremely fast, its scalability is constrained by the hardware limit of 16 pkeys per thread. It is infeasible to assign a unique pkey to every page when a VM may contain thousands or even millions of pages. As a result, *WRPKRU* cannot support fine-grained, per-page privilege management at scale.

In contrast, PTE update operates by updating the PTE itself, allowing any page to be re-tagged with one of the available pkeys. This mechanism avoids the pkey exhaustion problem in *WRPKRU* and makes it possible to enforce per-page protection policies even in large address spaces. However, PTE update is more expensive.

3) Hybrid Access Privilege Management Method: The access control of TCB pages and shared memory pages differs in SURE. TCB pages comprise a relatively small, fixed set of runtime components inside each unikernel (*e.g.,* scheduler, NetIO library, sidecar). These pages are accessed frequently by user code, but isolation only requires a coarse boundary between the trusted TCB and the untrusted user code. For TCB pages, *WRPKRU* is ideal: it toggles per-thread access permission in tens of cycles, providing fast, low-cost protection on critical execution paths. Since each pkey can be associated with a group of pages rather than individual ones, updating a pkey's privilege via *WRPKRU* instantly applies to all pages tagged with that pkey, enabling efficient bulk protection without per-page overhead.

Shared memory pages, in contrast, are dynamically allocated, span large memory pools, and must enforce isolation across multiple VMs. Here, fine-grained control is essential, since only specific pages should be accessible to specific VMs.

Authorized licensed use limited to: Politecnico di Torino. Downloaded on June 15,2026 at 08:32:37 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2026.3697819

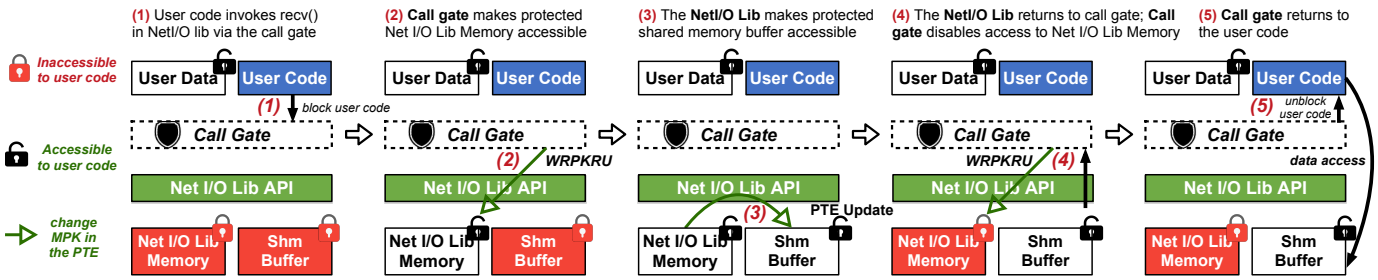


Fig. 6: SURE uses call gates to secure function calls by user code. SURE can dynamically change the access privilege of memory pages.

Although PTE update is slower, it can reassign protection at the individual page level and scale with the size of the pool. Because these pages are accessed less frequently than TCB pages, the higher cost can be amortized across the more involved inter-VM communication.

In practice, SURE combines both methods: *WRPKRU* is used to protect TCB components, where low-latency toggling is critical and key usage is minimal, while PTE update is reserved for shared memory, where fine-grained inter-VM isolation is required and the higher cost can be tolerated.

We configure four pkeys within each thread's `PKRU`. A dedicated T-key protects TCB pages: its permission is set to No-Access by default and is temporarily enabled via *WRPKRU* when privileged execution is required. For shared memory, three keys are used: an RW-key (Read/Write), an RO-key (Read-Only), and an NA-key (No-Access). Their privilege remain static, with PTE update reassigning pages between RW-key/RO-key/NA-key as needed. At unikernel boot, we record the page ranges for TCB code/data (scheduler, NetIO lib, sidecar lib, etc) and for the shared memory pool (data + signal regions), producing a mapping table of (page_addr $\rightarrow$ class $\in$ {TCB, shared memory}). All TCB pages are initially tagged with the T-key, while all shared memory pages are tagged with the NA-key.

C. Secure APIs based on Call Gates

To enforce safe interaction between untrusted user code and privileged TCB components or shared memory buffers, SURE introduces an MPK-based call gate abstraction. A call gate mediates entry into TCB APIs by transparently managing access privileges and execution context. When user code invokes a protected API, the call gate: (i) temporarily enables access to TCB pages by updating the T-key in `PKRU`; (ii) switches execution onto a protected stack and executes the privileged function on behalf of the user; and (iii) re-locks TCB memory before returning control.

For APIs that manipulate shared memory buffers, the call gate coordinates fine-grained privilege changes. When a buffer is allocated or received, its pages are re-tagged from the NA-key to either the RW-key or RO-key using PTE update, granting access to untrusted user code. The RW-key is assigned when the caller requires write permission (typically for producers that generate or modify the buffer's contents), whereas the RO-key is used when the buffer should be shared read-only with one or more consumers (*e.g.,* functions in a branching microservice chain). This design allows SURE to flexibly support microservice patterns such as single-producer/multi-consumer communication, where a producer function updates a buffer and then marks it read-only for downstream functions. Conversely, when a buffer is sent or freed, the pages are reset to NA-key, revoking access.

Fig. 6 shows this process for the NetIO's `recv()` API. ① User code calls `recv()` in the NetIO library through the call gate. ② The call gate uses *WRPKRU* to unlock NetIO Lib (stack) memory, making it accessible, and executes the `recv()` API; ③ The NetIO function uses the descriptor to address the shared memory buffer and makes it accessible by re-tagging its page with the RW-key via PTE update. ④ On return, the call gate re-locks NetIO memory. ⑤ Control returns to the user code, which can now safely access the received buffer.

Other TCB APIs (*e.g.,* in scheduler or page table management) are guarded in the same way. When running user code, all protected pages (*e.g.,* TCB memory or shared memory) are inaccessible. Any unauthorized updates to `PKRU` are prohibited through binary code inspection (see §VI-A). This ensures that only call gates can legitimately modify access rights, providing a secure interface for invoking TCB functionality.

VI. Securing MPK Enforcement in SURE

This section first highlights how unprivileged instructions and the single-address-space unikernel runtime make it difficult to provide isolation guarantees of MPK in SURE. We then present a general defense strategy that fortifies the MPK-based isolation in SURE's unikernel runtime.

A. MPK Privilege Escalation

Although MPK offers lightweight intra-process isolation, it is not a standalone security primitive. Two design aspects create the privilege escalation risk of MPK in a unikernel:

- ***WRPKRU* is an unprivileged, user-level instruction.** Any code running in the address space can execute it (or its equivalent byte sequences) to overwrite the `PKRU` register and change its view of pkey permissions.
- **Unikernels run at supervisor privilege.** In the single-address-space unikernel, a compromised user function can directly tamper with the data structures that determine MPK enforcement: it can call privileged APIs, write to the memory pages that hold PTEs, or overwrite saved thread contexts (which contain the `PKRU`).

Together, these expose several potential concrete escalation scenarios. Below, we enumerate each scenario and indicate why it subverts MPK unless additional controls are applied.

Authorized licensed use limited to: Politecnico di Torino. Downloaded on June 15,2026 at 08:32:37 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2026.3697819

1) Direct execution or injection of WRPKRU: Because *WRPKRU* is unprivileged, the attacker can execute it directly to change the PKRU. In addition, if executable pages can be written at runtime (*i.e.,* write-xor-execute (W⊕X [44]) is not enforced), the attacker can inject machine code containing *WRPKRU* and then execute it in the unikernel.

Impact on SURE: Direct execution or injected *WRPKRU* code can bypass the call gate mediation, breaking the intended access control in SURE's unikernel runtime.

2) PTE tampering (via pkey bits or Page Frame Number): A PTE encodes both the pkey associated with the page and the physical page frame number (PFN) that identifies the physical frame associated with the virtual page (see Fig. 5). The attacker can directly overwrite the PTE's pkey bits (*e.g.,* change an NA-key to an RW-key) or remap the protected physical frame to a PTE (by changing the PFN) that is already assigned with a permissive pkey (*e.g.,* a PTE that the attacker controls and that is tagged RW-key). In either case, after the PTE write and TLB flush, ordinary loads and stores to that protected virtual page/physical frame will succeed despite the intended protection.

Impact on SURE: MPK enforces access based on the pkey in the PTE that the CPU MMU consults on each memory access. When an attacker changes the pkey field or replaces the PFN in the PTE so the virtual address points to a frame with a permissive pkey, the hardware check is bypassed entirely and MPK can no longer enforce the intended isolation.

3) Thread context corruption: A thread context, which contains the PKRU, is saved and later restored during context switches and on interrupt entry/exit. These operations occur frequently within the SURE unikernel due to thread multiplexing and event-driven signaling (*e.g.,* switching between user threads, signal handlers, and I/O routines). When the thread is interrupted, the CPU scheduler stores its thread context on the interrupt stack, which is unprotected memory from user code in a single-address-space unikernel. The attacker can modify the saved PKRU in the interrupt stack when the scheduler or ISR restores the thread context, and the thread would resume with the modified pkey permission.

Impact on SURE: MPK's enforcement relies on correct PKRU values saved and restored during context switching; tampering with saved PKRU provides a persistent privilege escalation that bypasses MPK-based isolation.

4) CPU register manipulation (XSAVE and XRSTOR): Existing x86 CPUs provide instructions *XSAVE/XRSTOR* to save and restore the CPU's extended states, including the PKRU register.[8] The attacker can directly exploit these instructions to persist a thread's PKRU into a writable user-addressable memory in a single-address-space unikernel, and then corrupt the saved PKRU field. A subsequent *XRSTOR* that restores state from the tampered buffer will load the attacker-controlled PKRU into the CPU register.

[8]The *XSAVE/XRSTOR* instruction set was introduced to improve context switch performance and scalability by allowing the OS to selectively save and restore only the active portions of the CPU's extended state (*e.g.,* FPU, AVX, MPK), thereby avoiding unnecessary memory operations as the architectural state continues to expand.

Impact on SURE: This attack leverages *XSAVE* and *XRSTOR* to save and restore PKRU directly from user-managed memory, then changes the thread's pkey rights without executing *WRPKRU* or otherwise interacting with the intended call-gate mediation. The result is functionally equivalent to thread context corruption and it survives until the unikernel runtime overwrites the thread's PKRU during a legitimate save and restore via the call gate.

B. Strategy for Securing SURE

Based on the analysis above, MPK alone is insufficient inside a single-address-space unikernel. The attack surface includes (i) data structures that enforce protection (*e.g.,* PTEs, PKRU in the saved thread contexts) and (ii) instructions that can directly alter PKRU or restore it from attacker-controlled memory (*WRPKRU*, *XSAVE/XRSTOR*). To make SURE more secure while using MPK, we enforce a set of complementary defenses specifically to close these exploits.

- **Binary inspection and W⊕X enforcement.** At unikernel boot, we statically validate that only safe occurrences of *WRPKRU* and *XSAVE/XRSTOR* exist in the binary (same as [43]), ensuring untrusted code cannot directly manipulate PKRU without the call gate. Combined with a strict W⊕X policy, this prevents injection of *WRPKRU*, *XSAVE/XRSTOR* instructions at runtime. Legitimate uses of *WRPKRU* and *XSAVE/XRSTOR* are confined to TCB routines that can be entered only through call gates. This defense blocks attacks §VI-A1 and §VI-A4 within our threat model.
- **Blacklisting TCB Pages.** All data structures that determine protection (PTEs, saved thread contexts) are placed in MPK-protected memory (assigned the T-key) so they are inaccessible for untrusted code. However, even legitimate call-gate operations (*e.g.,* via PTE update) could inadvertently expose these critical pages if not explicitly filtered. To prevent this, SURE maintains a blacklist of guest physical addresses of all TCB pages. Whenever a protected page is about to be accessed by the user code through a call gate, the call gate checks whether the target physical address belongs to the blacklist; if so, the operation is immediately aborted and the unikernel halts. This helps prevent both direct memory access and legitimate API invocation from making these pages writable by user code. Within our threat model, this also blocks privilege-escalation attempts through PTE tampering (§VI-A2) or thread context corruption (§VI-A3).

C. Security Evaluation

To evaluate whether SURE's additional defenses prevent MPK privilege escalation paths in §VI-B, we implement a controlled proof-of-concept (PoC) attack evaluation to compare a baseline MPK configuration against SURE's secured MPK design. In the baseline, memory isolation relies only on MPK domain permissions enforced through PKRU, without SURE's additional protections in §VI-B. MPK privilege escalation paths remain open under this baseline.

Authorized licensed use limited to: Politecnico di Torino. Downloaded on June 15,2026 at 08:32:37 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2026.3697819

TABLE II: Proof-of-concept attack evaluation comparing baseline MPK and SURE's secured MPK design.

Attack scenarios	Baseline MPK outcome	SURE outcome
(1) Intra-domain unauthorized shared memory access	Blocked	Blocked
(2) Intra-unikernel unauthorized TCB memory access	Blocked	Blocked
(3) Unauthorized MPK permission update via *WRPKRU*	Unauthorized access succeeds	Blocked by binary inspection / W⊕X
(4) PTE tampering	Unauthorized access succeeds	Blocked by blacklisting TCB pages
(5) Thread-context corruption	Unauthorized access succeeds	Blocked by blacklisting TCB pages
(6) CPU register manipulation via *XSAVE*/*XRSTOR*	Unauthorized access succeeds	Blocked by binary inspection / W⊕X

Attack scenarios. We implement PoC exploits for six representative attacks derived from SURE's threat model. The first two correspond to ordinary unauthorized accesses that baseline MPK is already designed to prevent when `PKRU` and PTE remain intact: (1) *Intra-domain unauthorized shared memory access*. A compromised function attempts to directly read/write a shared memory buffer that is currently owned by another function without invoking the call gate. (2) *Intra-unikernel unauthorized TCB memory access*. A compromised function attempts to directly access memory belonging to trusted runtime components inside the unikernel, such as sidecar-related or other TCB-resident state, without authorization via the call gate. The remaining four correspond to privilege escalation paths analyzed in §VI-A, which attempt to subvert MPK enforcement in SURE.

Methodology. We assume the attacker has arbitrary code execution within an untrusted function and can invoke any interfaces exposed to that single-address-space unikernel function under the evaluated configuration. For each PoC exploit, we launch an untrusted function (attacker) that deliberately attempts to read or write an unauthorized memory location. We count an attack as successful only if the exploit completes an unauthorized memory access. We count an attack as blocked if the exploit is rejected before execution or stopped before any unauthorized access occurs.

Results. Table II summarizes the results. Both baseline MPK and SURE block the attacks 1 & 2, showing that MPK can prevent unauthorized memory access within the same security domain and within the same unikernel. However, the remaining four MPK-privilege-escalation PoC attacks (3-6) can succeed in the baseline MPK configuration, whenever the attacker can directly manipulate `PKRU`, restore attacker-controlled CPU state, or tamper with protection-critical metadata in the PTE. In contrast, under SURE's configuration, none of the four privilege-escalation PoCs can obtain unauthorized access: unauthorized *WRPKRU* and *XSAVE/XRSTOR* use is rejected by binary inspection, code-injection paths are prevented by W⊕X, and PTE remains inaccessible due to TCB page blacklisting during call-gate validation.

Note that this evaluation does not constitute a completeness proof against all possible MPK bypasses. For instance, 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 [43], [45], [46]. Rather, it shows that SURE blocks several representative MPK privilege-escalation paths in our threat model while preserving the original protection provided by baseline MPK for ordinary unauthorized accesses.

VII. Scalability of SURE

Unikraft's `runu` [14] enables Kubernetes to run SURE unikernels inside VMs as if they are Linux containers. This allows SURE to be integrated with Kubernetes-based serverless platforms such as Knative to leverage its autoscaler, scales based on HTTP request concurrency or requests per second.

A common concern is whether the MPK-based isolation model limits the scalability of SURE, particularly when functions are organized into long chains or when many pages are involved. Since MPK provides only 16 pkeys per thread, naively assigning one pkey per page does not scale. SURE addresses this challenge in two ways: (1) At the chain level, the name-based security domain ensures that each chain is provisioned with its own isolated memory pool, preventing cross-chain interference regardless of the number of chains deployed. (2) Within each function, the hybrid access privilege management method requires only three keys in total. This design avoids allocating keys on a per-page basis, thereby mitigating pkey exhaustion concerns The remaining 12 pkeys are available for potential extensions, such as partitioning TCB components with finer granularity, which we leave as future work.

Beyond memory isolation, SURE's unikernel runtime further improves deployment density. Compared to containerized environments, SURE supports up to 2.6× more concurrent function instances, and up to 30.7× more than full-size VMs (see §IX-E). Unikernels also offer much lower startup latency, enabling fast scaling in reaction to traffic spikes (evaluated in §IX-C).

VIII. Implementation of SURE

We base the development of SURE on Unikraft [11] (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. SURE adds support for the shared memory device and library-based sidecar in Unikraft. We use *ivshmem-plain* to implement the data region (shared memory buffers) and use *ivshmem-doorbell* to implement the signal region for inter-VM signaling. We use QEMU/KVM (8.2.0) as the hypervisor. A small patch to QEMU's *ivshmem* device is required, limited to the *ivshmem-doorbell* path.[9] SURE does

[9]This patch adapts the doorbell handling to better match SURE's inter-VM signaling design, by using per-peer MMIO offsets for event notification.

Authorized licensed use limited to: Politecnico di Torino. Downloaded on June 15,2026 at 08:32:37 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2026.3697819

not replace *ivshmem*; rather, it builds secure communication and isolation mechanisms on top of it, including domain-level shared memory pools, event-driven signaling, and MPK-based memory protection. We implement the strategy in §VI-B to secure the MPK enforcement in SURE's unikernel runtime.

Note that some VMMs, including Firecracker [23] and QEMU's microvm [47], 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 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.

Application porting requirements. SURE functions interact with the shared memory through the NetIO library, whose interface remains socket-like (*e.g.,* `send()`, `recv()`). However, NetIO is not fully POSIX-compliant. In particular, shared memory communication is descriptor-based and tied to pre-allocated shared memory buffers, rather than arbitrary user-provided buffers. As a result, porting a conventional POSIX-socket-based application to SURE mainly requires changes in the data path: buffer preparation, `send()`/`recv()` invocation, and buffer lifetime management. However, the required adaptation is typically localized rather than requiring a full application rewrite. For applications written in higher-level languages, this adaptation can be further reduced through shim layers; our prior experience with X-IO [48] suggests that such an approach can make shared memory communication practical even for large software systems. Refer to Appendix-B in the supplementary material for more details.

IX. Evaluation of SURE

We 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 [49] from Google.

Testbed. Experiments were run on NSF Cloudlab with three nodes, each equipped with a 32-core Intel Xeon Silver 4314 CPU (2.4GHz, MPK-enabled), 128GB RAM, and a 100Gbps NIC. All nodes run Ubuntu 22.04 with Linux kernel 5.15.

A. Microbenchmark Analysis

1) Improvement from shared memory processing: We first evaluate the round-trip latency and throughput between a client and server pair on the *same* node. We choose three message sizes: 64B, 4KB, and 8KB. There is very little variation for SURE for packet sizes ranging from 64B to 1KB. We compare SURE's intra-node shared memory data plane with the following widely used alternatives: (1) Containers; (2) Unikraft unikernels [11]; (3) OSv unikernels [9]. For containers, we use veth pairs connected through the kernel Linux bridge. For Unikraft and OSv unikernels, when using the kernel Linux bridge, connectivity is provided through *vhost-net*; alternatively when using the userspace Open vSwitch [50] for Unikraft and OSv, connectivity is provided through *vhost-user*. SURE and Unikraft both run on standard QEMU. For the OSv baseline, we run it on Firecracker.

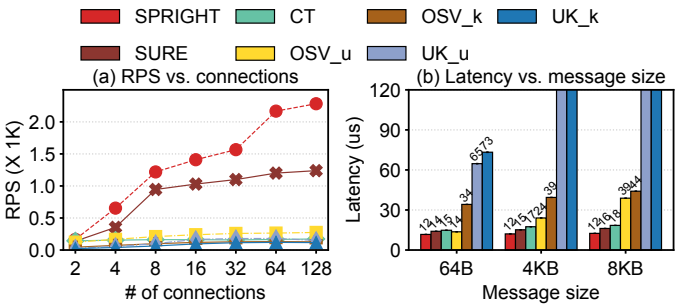


Fig. 7: Intra-node data plane performance. **CT**: container with Linux bridge; **OSv_u**: OSv with userspace OVS; **OSv_k**: OSv with Linux bridge; **UK_u**: Unikraft with userspace OVS; **UK_k**: Unikraft with Linux bridge; (average of 30 repetitions)

To match SURE's reliable data transfer, we use TCP for reliable data transfer with the other alternatives. Note that containers use the host's TCP/IP stack, Unikraft uses *lwip* [51], and OSv's TCP/IP stack is ported from FreeBSD. For throughput measurements, we add a sufficient number of clients to saturate the server, and metrics are collected on the server.

SURE achieves low-latency. We show the latency for a single client-server connection in Fig. 7 (right). 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[10] and has an extra copy[11]).

SURE is more scalable and efficient. We evaluate throughput (requests per second (RPS)) as the number of concurrent connections increases. Fig. 7 (left) 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 more efficient: with more than 16 connections, all alternatives saturate their one assigned CPU core on the server. But SURE has a much higher RPS (and continues to increase with increasing concurrent connections beyond 128).

Comparison against vhost-user. Although *vhost-user* also relies on shared memory and event notification [52], its abstraction is different from SURE's. Under the standard *vhost-user* abstraction, packets passed across multiple VMs in a function chain cannot remain in a single shared buffer visible to all participants. Instead, intermediate VMs typically need to reinject (copy) the data into the communication path to the next function, incurring additional inter-VM copy overhead, as discussed in [53]. In contrast, SURE uses *ivshmem* to let all VMs in the same security domain map the same memory pool directly, which is a more natural method for zero-copy communication across multiple VMs.

MPK in SURE has a limited penalty. To quantify the performance penalty of MPK, we compare SURE against SPRIGHT, which uses pure shared memory without MPK. As shown

[10]https://savannah.nongnu.org/patch/?10111

[11]https://github.com/unikraft/lib-lwip/blob/staging/uknetdev.c#L166-L167

Authorized licensed use limited to: Politecnico di Torino. Downloaded on June 15,2026 at 08:32:37 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2026.3697819

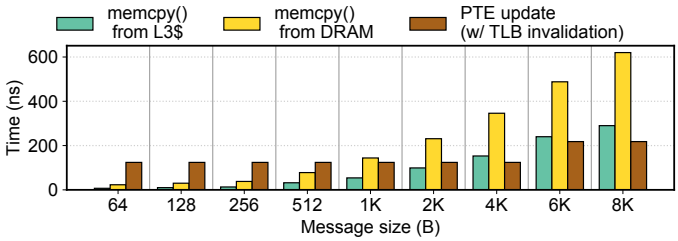


Fig. 8: Cost of copying messages of different sizes from L3 cache or main memory compared to the PTE update in SURE.

in Fig. 7, with a single connection, SURE shows 1.2-1.3× increased latency compared to SPRIGHT. SURE's peak RPS is also 1.8× less than SPRIGHT (with 128 connections). We believe this overhead is still acceptable 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 [42], 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.

2) Cost of updating the PTEs: We compare SURE's MPK-based zero-copy approach against an alternative design that copies messages from protected shared memory into a local user buffer on message reception. In the SURE design, the first access incurs a TLB miss and fetches the updated translation from the page table. With the copy-based design, we use *memcpy()* from *glibc* to copy the message into a local user buffer. Because a message produced by another function on a different CPU core is likely to reside either in the shared L3 cache or in main memory, we evaluate both cases.

As shown in Fig. 8, SURE's approach of updating the PTE (along with the corresponding TLB invalidation) outperforms the copy-based design once the message size is reasonably large. Specifically, SURE's approach of updating PTEs and invalidating the TLB entry is faster than copying messages that are 1 KB or larger when the source data originates from main memory, and faster than copying messages larger than 4 KB when the data is in the shared L3 cache. These results show that although SURE introduces a fixed overhead, that overhead is quickly amortized for larger messages, making the PTE update preferable to copying in those cases. Only for very small messages is copying more efficient. A hybrid design that dynamically chooses between copying and SURE's MPK-based zero-copy design based on message size, similar to [36], is an interesting direction for future work.

3) Improvement with library-based sidecar: We consider an individual container for each sidecar as the baseline to compare against. Each sidecar connects to the user function container over the kernel loopback interface [21]. We use the NGINX proxy as the implementation of the sidecar, as demonstrated in production [54]. 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

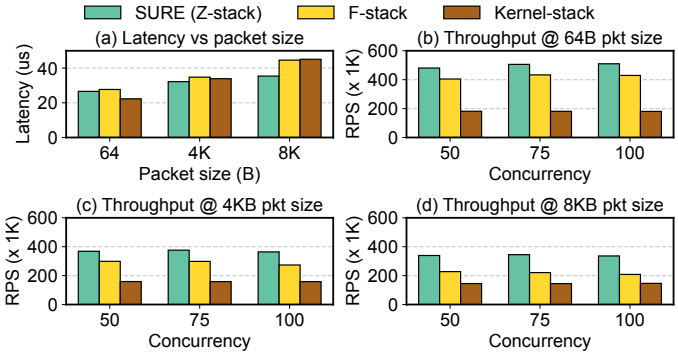


Fig. 9: Latency and throughput comparison of SURE's Z-stack, F-stack, and kernel stack under varying packet sizes and concurrency levels. (a) Latency under varied packet sizes (with 1 connection). (b-d) Requests-per-second (RPS × 1K) at 64B, 4KB, and 8KB packet sizes, measured under increasing concurrency (50, 75, 100).

the throughput, we use concurrent connections to saturate the user function and sidecar. We show the throughput results with 64 connections in Table III. However, the observations are consistent for other values (*e.g.,* 32, 128 connections).

Library-based sidecar shows negligible overhead. Table III compares the CPU cycles spent on the sidecar for different alternatives. The CPU cycle consumption by our library-based sidecar is negligible (only 0.9%) compared to the NGINX sidecar's CPU usage. The higher CPU cycle consumption by the NGINX sidecar is due to the loose coupling between the sidecar and the function container, which results in additional overhead from the kernel's loopback interface. The extra CPU consumption also results in increased networking latency 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 the sidecar entirely disabled ("no SC" in Table III). This is the advantage of our library-based sidecar - maintaining low latency and high throughput, with negligible CPU consumption.

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.

4) Benefit of the zero-copy TCP/IP stack: We compare SURE's Z-stack with F-stack [35] for protocol processing in the SURE gateway. The F-stack gateway incurs a data copy when exchanging payloads with the function chain. We also

TABLE III: Library-based sidecar (LibSC) vs. Individual sidecar (NGINX). "no SC" refers to the setting without the sidecar.

Msg size	Added CPU cycles (× 1K)		Added delay (us)		Throughput (MBytes per second)		
	LibSC	*NGINX*	*LibSC*	*NGINX*	*no SC*	*LibSC*	*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

Authorized licensed use limited to: Politecnico di Torino. Downloaded on June 15,2026 at 08:32:37 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2026.3697819

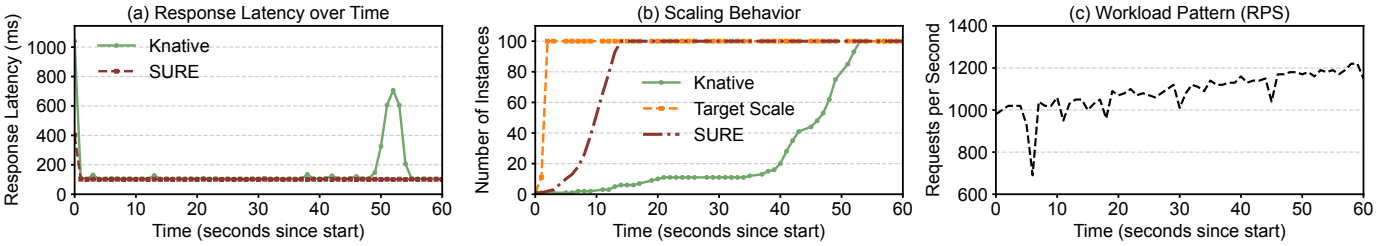


Fig. 10: Dynamic scaling under Azure workload [55]. (a) Response latency over time; (b) number of active function instances; (c) workload pattern in RPS. Each instance handles up to five concurrent requests.

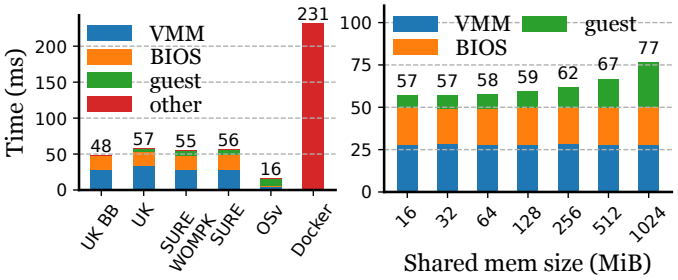


Fig. 11: Boot time: (left) comparison of different runtimes; (right) influence of the shared memory size on SURE with MPK protection.

compare Z-stack with the Linux kernel's protocol stack to evaluate the performance improvement and costs of using the DPDK PMD. For the kernel stack 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.

Z-stack achieves a $\sim$1.2$\times$ RPS improvement and latency reduction, under high traffic load (more than 100 connections) compared to F-stack (Fig. 9). This clearly showcases the advantage of having zero-copy protocol processing. On the other hand, F-stack 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 the kernel stack under very light loads with small packets (*e.g.,* 64B, single connection in Fig. 9 (top 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 kernel stack for larger message sizes.

Polling overhead of Z-stack. Because Z-stack uses DPDK's busy-polling PMD, it incurs a fixed CPU overhead at light load, but becomes more efficient than the interrupt-driven kernel stack at higher load by avoiding interrupt and kernel overheads. Additional quantitative analysis is provided in Appendix-E of the supplementary material.

B. Function Startup in SURE

We compare the function boot time across unikernel alternatives and containers. We use a bare-bones Unikraft unikernel (**UK-BB**) as the baseline (no network interfaces, no support for virtual memory or multithreading, etc.). We also include a Unikraft unikernel suitable for supporting serverless functions (**UK**), equipped with a vhost network interface, virtual memory, and multithreading. We consider SURE with and without MPK enabled (**SURE** and **SURE-WOMPK**, respectively). All Unikraft/SURE variants run on standard QEMU. For OSv, we run it on Firecracker (same as in §IX-A1).

All unikernel alternatives have 8 MB RAM. SURE VMs also attach a 16 MB shared memory region. We also run a 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. 11 (left) shows that unikernel-based solutions boot much faster: SURE is $\sim$4$\times$ faster than Docker containers. Additionally, SURE's shared memory data plane (**SURE** bar) is comparable to a traditional network setup based on virtio/vhost interfaces (**UK**). Adding MPK has a negligible boot-time impact. Most of the unikernel boot time is in the VMM and BIOS, which can be reduced by leveraging a microVM VMM such as Firecracker [23] (Fig. 11 (left) shows that the guest boot time (in green) of SURE (based on Unikraft) is comparable to OSv+Firecracker).

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. 11 (right). 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 (§IX-D).

C. Dynamic Scaling

Experiment setup: We evaluate how rapidly SURE scales functions in response to traffic surges and compare its behavior with Knative's container-based runtime.

We use Knative's official Autoscale Sample App[12], configured as an HTTP server function that sleeps for 100 ms per request before returning a response. This fixed processing time ensures that observed startup latency differences stem from runtime (unikernel vs. container) instead of application logic. Traffic is generated using loadtest,[13] replaying the Azure workload [55]. The invocation pattern is shown in Fig. 10 (c).

[12]https://github.com/knative/docs/tree/main/docs/versioned/serving/autoscaling/autoscale-go

[13]https://github.com/alexfernandez/loadtest.git

Authorized licensed use limited to: Politecnico di Torino. Downloaded on June 15,2026 at 08:32:37 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2026.3697819

We use the default autoscaler in Knative (concurrency-based).[14] The scaling target is set to 5 in-flight requests per instance, ensuring that a single function can contribute at most 5×100 ms $= 500$ ms of queuing delay before requiring it to be scaled out. We perform the experiment on a single node.

Result analysis: Fig. 10 (a) shows the end-to-end response latency as the system reacts to incoming load. We show the latency of all requests completed within a one-second interval. SURE consistently maintains low latency (around 100 ms, equal to the function's processing time) throughout the experiment. A small initial increase appears as cold-start unikernels are launched, but these start within milliseconds and quickly reach steady state. By contrast, Knative exhibits large latency spikes between 48-56 s as request backlogs accumulate while waiting for new containers to be initialized in response to the autoscaling decision.

Fig. 10 (b) shows the scaling behavior. The target scale represents the number of instances requested by the autoscaler to meet the load thresholds. SURE tracks this target reasonably closely as unikernel instances enable fast startup, reaching equilibrium within about 11 seconds. Knative, however, lags significantly due to the slow startup of container runtimes, as shown in Fig. 11 (left). This delay is amplified when many containers are launched concurrently within a short period. During such bursts, each container requires the creation of a new network namespace and the attachment of a veth interface. Since Linux organizes namespaces as a linked list protected by a global lock, these operations serialize under contention, leading to long startup delays [56].

D. Realistic Workload Evaluation

The Online Boutique [49] is a microservices-based online store application from Google, consisting of 10 functions and 6 different function chains in a serverful setup. By default, it uses a container-based runtime and using gRPC to interconnect functions (called "**Container**"). We additionally consider an OSv-based serverful alternative (called "**OSv**"). We evaluate **OSv** on Firecracker [23]. Connectivity between functions in both **Container** and **OSv** is provided through the kernel Linux bridge. **Container** uses veth pairs; **OSv** uses *vhost-net*. For the call graphs of the Online Boutique function chains, refer to [49].

We consider three serverless platforms to compare against: **Knative** as the baseline, two state-of-the-art serverless platforms, **SPRIGHT** [19] and **NightCore** [3]. We port the Boutique microservices to SURE and OSv. We use the Locust [57] load generator with the Boutique's default workload [49]. We disable the default user wait time to generate a heavier workload. Two deployment settings are used: (1) *intra-node*: all functions on one node;, and (2) *inter-node*: Frontend, Checkout, and Recommendation functions (intermediate functions that could become hotspots) on one node; remaining (leaf) functions on a second node. Note, NightCore [3] does not support inter-node communication between functions of the same chain.

1) RPS and Tail Latency: As shown in Fig. 12 (a) (intra-node setup), SURE's RPS is up to 17× and 79× higher than **Container** and **Knative**. Both **Container** and **Knative** become CPU-limited beyond a concurrency of 16 (RPS barely increases). At concurrency 16, SURE's 95%ile latency (shown in Fig. 13 (a)) is 0.39ms, making SURE highly attractive for latency-sensitive microservices. **SPRIGHT** also uses intra-node shared memory processing, but incoming client requests are first 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 (Fig. 13 (a)). **NightCore** performs worse than SURE due to its reliance on the kernel protocol processing and additional queuing in the NightCore engine (akin to our SURE gateway).

Under the inter-node setup (Fig. 12 (b)), SURE still performs much better than the others: up to 6× higher RPS than **Container** and **SPRIGHT**; 19× higher RPS than **Knative**. The zero-copy protocol processing in Z-stack of SURE maintains sub-millisecond latency, even at the 95%ile (CDF shown in Fig. 13 (b)), unlike **SPRIGHT**, whose tail latency is close to **Container**, as shared memory processing does not help for the inter-node setup. This is consistent with the microbenchmark of Z-stack in Fig. 9. **Knative**'s tail latency is 15× higher, and **Container**'s is 4.8× higher than SURE's. Note that SURE and **SPRIGHT** use a static CPU core allocation for the gateway, while the other alternatives support dynamic multi-core scaling. This results in some alternatives (*e.g.,* **Container**) seeing improvement in their performance when switching from the intra-node setup to the inter-node setup.

OSv performs far worse than SURE in both settings. At concurrency 64, **OSv**'s RPS is 46× lower than SURE in the intra-node case, and 25× less than SURE in the inter-node case. This gap is mainly because: *vhost-net* still incurs non-trivial data movement overheads [53], **OSv** still performs full TCP/IP processing in each function, and the kernel Linux bridge also introduces additional software overheads. In contrast, SURE avoids these costs through shared memory communication, consolidated protocol processing in the SURE gateway, and full-userspace isolation enforced with MPK.

2) 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 all CPU cores (as a percentage) divided by Request Per Second (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. 12 (c) and inter-node case in Fig. 12 (d). We can observe that **(i) SPRIGHT** is consistently more CPU-efficient since it uses eBPF-based event-driven shared memory processing, as observed in [19].

(ii) At low concurrency ($\leq$16 for intra-node; $\leq$4 for inter-node), SURE is less efficient than **SPRIGHT** (comes from polling and the use of a CPU for each function), but SURE achieves much higher RPS. SURE is far more efficient

[14] Knative provides two autoscalers: concurrency-based (default) and RPS-based. Since the function's processing time consistently remains at 100 ms, Little's Law dictates that the two approaches produce identical scaling behavior: concurrency = RPS × latency. We therefore use the default one.

Authorized licensed use limited to: Politecnico di Torino. Downloaded on June 15,2026 at 08:32:37 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2026.3697819

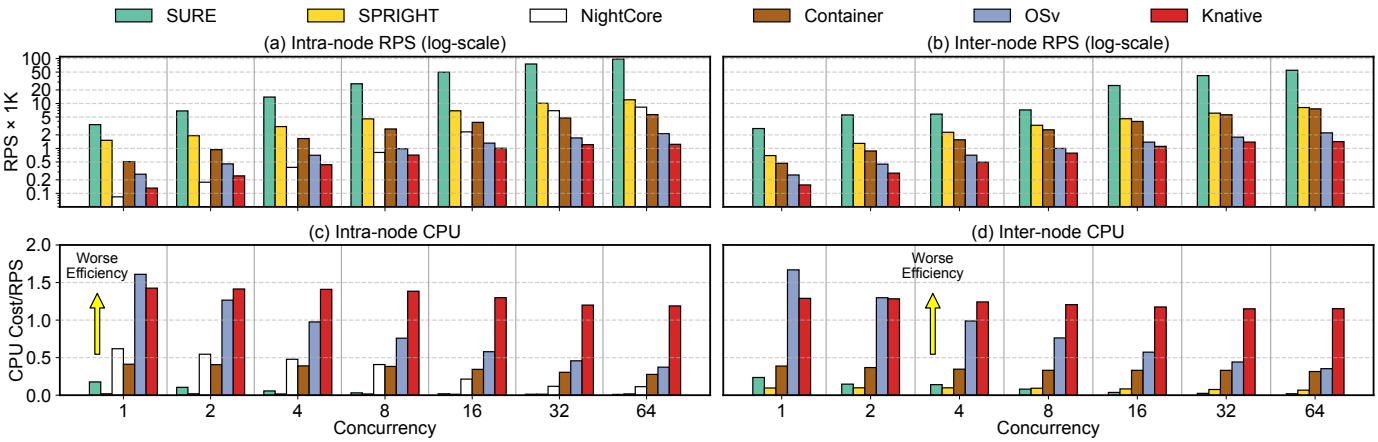


Fig. 12: RPS and CPU efficiency across concurrency levels (1-64). Top row: requests-per-second (RPS × 1K, log scale) for (a) intra-node, and (b) inter-node deployments; bottom row: CPU cost per request (lower is better) for intra-node (c), and inter-node (d). We sum the CPU cost of all components in the system where applicable (*e.g.,* gateway, function, sidecar). NightCore [3] does not support inter-node communication, so it is only evaluated in (a) and (c).

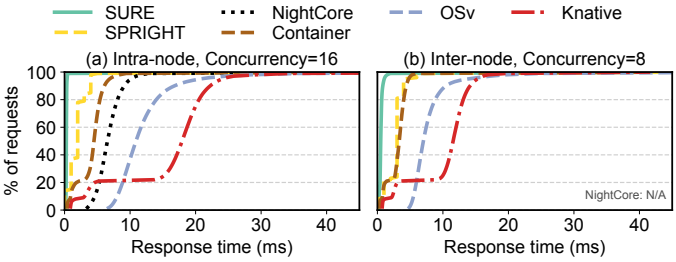


Fig. 13: Response-time CDFs under intra-node (a) and inter-node (b) deployments. NightCore [3] does not support inter-node communication, so it is only evaluated (a).

than **Knative** and **Container**. As concurrency grows, SURE becomes more efficient than **SPRIGHT**. This is because **SPRIGHT** depends on kernel-based networking for inter-node traffic, which in total consumes ≥2 CPU cores (aggregating the CPU usage of the gateways in **SPRIGHT** on the two nodes), for concurrency level more than 4; As we evaluated in §IX-A4, 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 **Knative** and **SPRIGHT**.

(iii) OSv, **Knative** and **Container** are inefficient due to kernel networking. **Knative** is the worst because of heavy-weight per-function sidecars. **OSv** is particularly poor at low concurrency (≤16), which we attribute to the lack of vhost-net acceleration, causing packets to traverse the VMM, which is equivalent to having an individual sidecar. **Container** is slightly better since its deployment does not include the sidecar. However, the cumulative CPU cost of **Container** will be unacceptable since functions in **Container** are always-on (like "Serverful") and occupy CPU/memory resources even with no requests to process.

E. Deployment Density and Scalability

Experiment Setup: To evaluate the deployment density of SURE, we instantiate a large number of online boutique function chains under three runtime environments: the SURE-based unikernel runtime, a containerized runtime using Docker with an Alpine image, and a full-size VM runtime based on Ubuntu Minimal. Each function chain in SURE is provisioned with an isolated security domain and a private memory pool. Every SURE VM is allocated 8 MB of RAM, and each chain is assigned 16 MB of shared memory, consistent with the configuration used in §IX-B. All experiments are performed on a single physical host equipped with 128 GB of RAM.

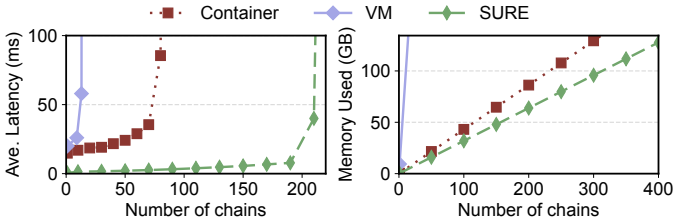


Fig. 14: Deployment density comparison across runtimes. (left) Average latency under increasing number of chains, each serving one user load from locust. and (right) Total memory footprint while all chains are idle (no traffic).

Result Analysis: Fig. 14 (left) shows the average latency as the number of deployed chains increases. Each chain serves one dedicated user from locust (same configuration as in §IX-D). The full-size VM setup quickly exhausts system memory after only 13 chains (∼9590 MB per chain), as indicated by the steep rise in Fig. 14 (right), which measures total memory footprint. The containerized runtime supports up to 86 chains before all 32 CPU cores are fully utilized. Beyond this point, container latency (see Fig. 14 (left)) grows sharply due to severe CPU contention contributed by kernel networking. In contrast, SURE maintains consistently low latency for almost 200 active chains because its shared memory data plane remains lightweight and minimizes CPU interference between chains. SURE scales to 220 active chains, maintaining stable latency before saturating all 32 CPU cores.

Fig. 14 (right) compares the total memory footprint under idle conditions (no traffic). The container setup supports up to 296 chains before exhausting host memory, while SURE's

Authorized licensed use limited to: Politecnico di Torino. Downloaded on June 15,2026 at 08:32:37 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2026.3697819

unikernel runtime, with its streamlined LibOS design, further reduces per-function footprint, supporting up to 400 chains per node (a 30.7× improvement over full-size VMs). These results demonstrate the better scalability of SURE's unikernel design.

X. Related Work

As discussed, prior work [19], [25] is limited to a single node with regard to secure shared memory processing and do not extend their high-performance to inter-node communication. An eBPF-based sidecar, as in [19], is also not viable for an unikernel-based serverless environment. [6], [7], [58] 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 [6], [7], [58] is also a concern.

Unikernel/LibOS Virtualization: Unikernels are very suitable for single-task microservices that need to run isolated, secure tasks with minimal overhead [6]. But they may not be ideal for complex serverless workloads requiring dynamic, multi-process environments or using extensive system services. Several past works have optimized different aspects of unikernels, including system development kits [9], [11], [24], multi-process support [59], and fast startup [12], [22]. SURE can take advantage of these unikernels' startup optimizations [12], [22] to reduce the cold-start penalty of serverless computing. CubicleOS [60] 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 TCP/IP Stack: There are multiple high-performance TCP/IP protocol stack implementations: StackMap [62] offers zero-copy TCP/IP processing by integrating a full-fledged Linux network stack with the interrupt-driven packet handling of netmap [63]. This could be less efficient under high loads compared to a full-fledged TCP/IP stack that uses DPDK's polling-based packet handling [35]; Demikernel [64] supports zero-copy processing through its TCP/IP stack. However, it is not full-fledged. In contrast, 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 [42], [43], [59], [65], [66]. Jenny [66] filters MPK-related syscalls to prevent unauthorized changes to the MPK key. ERIM [43] enforces binary inspection and rewriting to prevent misuse of MPK. libmpk [42] overcomes the limit of the 16 MPK keys by carefully recycling and redistributing keys. EPK [41] extends the supported number of MPK-protected memory domains by using a hardware-based protection key extension – the extended page table (EPT). This can enable MPK-protected shared memory communication between a large number of domains (up to 7680) [41]. These works are complementary to SURE. Other efforts focus on protecting the unikernel (*e.g.,* OCaml in Mirage [10]) and shared memory processing (Rust in RedLeaf [67]) at the language level. They can be used as further enhancements to SURE's unikernel runtime and shared memory data plane.

XI. Conclusions

In this paper we discussed SURE, a serverless computing framework that leverages lightweight unikernels because of their fast instantiation and small memory footprint. Using lightweight unikernels, SURE achieves 4× faster function startup and up to 2.6× higher deployment density than using containers. Compared to full-size VMs, SURE achieves 30.7× higher deployment density, enabling hundreds of concurrent function chains per node.

SURE also eliminates the overhead associated with kernel-based networking and sidecars through a distributed zero-copy data plane that spans multiple nodes. Under a realistic multi-node workload, SURE's data plane delivers up to 19× higher throughput and 15× lower tail latency than the commonly used open-source Knative serverless platform, while outperforming state-of-the-art research systems such as SPRIGHT by achieving 6×-8× higher throughput.

SURE's two-tier isolation design, comprising per-chain memory-pool security domains, and MPK-based call gates, effectively combines page-level protection with scalable, domain-level security boundaries. SURE further safeguards its single-address-space runtime by verifying the function chain's code through binary inspection, W⊕X enforcement, and TCB-page blacklisting, effectively closing MPK privilege escalations that may be overlooked by prior unikernel systems.

References

[1] S. Luo *et al.*, "Characterizing microservice dependency and performance: Alibaba trace analysis," ser. SoCC '21, 2021, p. 412–426. [Online]. Available: https://doi.org/10.1145/3472883.3487003

[2] A. Sahraei *et al.*, "Xfaas: Hyperscale and low cost serverless functions at meta," in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP '23, 2023, p. 231–246. [Online]. Available: https://doi.org/10.1145/3600006.3613155

[3] Z. Jia and E. Witchel, "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*, ser. ASPLOS '21, 2021, p. 152–166. [Online]. Available: https://doi.org/10.1145/3445814.3446701

[4] Q. Chen *et al.*, "Yuanrong: A production general-purpose serverless system for distributed applications in the cloud," in *Proceedings of the ACM SIGCOMM 2024 Conference*, ser. ACM SIGCOMM '24, 2024, p. 843–859. [Online]. Available: https://doi.org/10.1145/3651890.3672216

[5] X. Chai *et al.*, "Fork in the road: Reflections and optimizations for cold start latency in production serverless systems," in *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, 2025, pp. 199–218.

[6] B. Tan, H. Liu, J. Rao, X. Liao, H. Jin, and Y. Zhang, "Towards lightweight serverless computing via unikernel as a function," in *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*, 2020, pp. 1–10.

[7] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, "Seuss: Skip redundant paths to make serverless fast," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3392698

[8] A. Madhavapeddy *et al.*, "Unikernels: Library operating systems for the cloud," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13, 2013, p. 461–472. [Online]. Available: https://doi.org/10.1145/2451116.2451167

Authorized licensed use limited to: Politecnico di Torino. Downloaded on June 15,2026 at 08:32:37 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2026.3697819

[9] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov, "OSv—Optimizing the operating system for virtual machines," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, Philadelphia, PA, Jun. 2014, pp. 61–72. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity

[10] A. Madhavapeddy and D. J. Scott, "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*, vol. 11, no. 11, p. 30–44, dec 2013. [Online]. Available: https://doi.org/10.1145/2557963.2566628

[11] S. Kuenzer *et al.*, "Unikraft: Fast, specialized unikernels the easy way," in *Proceedings of the Sixteenth European Conference on Computer Systems*, ser. EuroSys '21, 2021, p. 376–394. [Online]. Available: https://doi.org/10.1145/3447786.3456248

[12] A. Madhavapeddy *et al.*, "Jitsu: Just-In-Time summoning of unikernels," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, Oakland, CA, May 2015, pp. 559–573. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/madhavapeddy

[13] "Compatibility of Unikraft," https://unikraft.org/docs/concepts/compatibility, 2024, [ONLINE].

[14] "runu OCI runtime," https://unikraft.org/docs/getting-started/integrations/container-runtimes, 2024, [ONLINE].

[15] "NanoVMs," https://nanovms.com/, 2024, [ONLINE].

[16] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, "Attack of the killer microseconds," *Commun. ACM*, vol. 60, no. 4, p. 48–54, mar 2017. [Online]. Available: https://doi.org/10.1145/3015146

[17] F. Parola, S. Qi, A. B. Narappa, K. K. Ramakrishnan, and F. Risso, "Sure: Secure unikernels make serverless computing rapid and efficient," in *Proceedings of the 2024 ACM Symposium on Cloud Computing*, ser. SoCC '24, 2024, p. 668–688. [Online]. Available: https://doi.org/10.1145/3698038.3698558

[18] "Memory protection keys," https://lwn.net/Articles/643797/, 2024, [ONLINE].

[19] S. Qi, L. Monis, Z. Zeng, I.-c. Wang, and K. K. Ramakrishnan, "Spright: Extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing," in *Proceedings of the ACM SIGCOMM 2022 Conference*, ser. SIGCOMM '22, 2022, p. 780–794. [Online]. Available: https://doi.org/10.1145/3544216.3544259

[20] https://istio.io/, 2024, [ONLINE].

[21] X. Zhu *et al.*, "Dissecting overheads of service mesh sidecars," in *Proceedings of the 2023 ACM Symposium on Cloud Computing*, ser. SoCC '23, 2023, p. 142–157. [Online]. Available: https://doi.org/10.1145/3620678.3624652

[22] F. Manco *et al.*, "My vm is lighter (and safer) than your container," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17, 2017, p. 218–233. [Online]. Available: https://doi.org/10.1145/3132747.3132763

[23] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, Santa Clara, CA, Feb. 2020, pp. 419–434. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/agache

[24] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the library os from the top down," vol. 39, no. 1, p. 291–304, mar 2011. [Online]. Available: https://doi.org/10.1145/1961295.1950399

[25] M. Yu, T. Cao, W. Wang, and R. Chen, "Following the data, not the function: Rethinking function orchestration in serverless computing," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, Boston, MA, Apr. 2023, pp. 1489–1504. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/yu

[26] S. Qi, K. K. Ramakrishnan, and M. Lee, "Lifl: A lightweight, event-driven serverless platform for federated learning," in *Proceedings of Machine Learning and Systems*, P. Gibbons, G. Pekhimenko, and C. D. Sa, Eds., vol. 6, 2024, pp. 408–425. [Online]. Available: https://proceedings.mlsys.org/paper_files/paper/2024/file/c2a0e26dd9ee7d57e92bb1c24b39659a-Paper-Conference.pdf

[27] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA, Aug. 2014, pp. 719–732. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[28] D. Dean and A. J. Hu, "Fixing races for fun and profit: how to use access (2)." in *USENIX security symposium*, 2004, pp. 195–206.

[29] J. Z. Yu, S. Shinde, T. E. Carlson, and P. Saxena, "Elasticlave: An efficient memory model for enclaves," in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, Aug. 2022, pp. 4111–4128. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/yu-jason

[30] E. Song *et al.*, "Canal mesh: A cloud-scale sidecar-free multi-tenant service mesh architecture," in *Proceedings of the ACM SIGCOMM 2024 Conference*, ser. ACM SIGCOMM '24, 2024, p. 860–875. [Online]. Available: https://doi.org/10.1145/3651890.3672221

[31] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: attacks and defenses for the vulnerability of the decade," in *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, vol. 2, Jan 2000, pp. 119–129 vol.2.

[32] D. Ray and J. Ligatti, "Defining code-injection attacks," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '12, 2012, p. 179–190. [Online]. Available: https://doi.org/10.1145/2103656.2103678

[33] "Inter-VM Shared Memory device," https://www.qemu.org/docs/master/system/devices/ivshmem.html, 2024, [ONLINE].

[34] "Provisioned Concurrency for Lambda Functions," https://aws.amazon.com/blogs/aws/new-provisioned-concurrency-for-lambda-functions/, 2024, [ONLINE].

[35] "F-Stack," https://www.f-stack.org/, 2024, [ONLINE].

[36] B. Li, T. Cui, Z. Wang, W. Bai, and L. Zhang, "Socksdirect: Datacenter sockets can be fast and compatible," in *Proceedings of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '19, 2019, p. 90–103. [Online]. Available: https://doi.org/10.1145/3341302.3342071

[37] "DPDK Poll Mode Driver," https://doc.dpdk.org/guides/prog_guide/poll_mode_drv.html, 2024, [ONLINE].

[38] J. Hwang, K. K. Ramakrishnan, and T. Wood, "NetVM: High performance and flexible networking using virtualization on commodity platforms," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, Seattle, WA, Apr. 2014, pp. 445–458. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/hwang

[39] A. B. Narappa, F. Parola, S. Qi, and K. K. Ramakrishnan, "Z-stack: A high-performance dpdk-based zero-copy tcp/ip protocol stack," in *2024 IEEE 30th International Symposium on Local and Metropolitan Area Networks (LANMAN)*, 2024, pp. 100–105.

[40] H. Saokar *et al.*, "ServiceRouter: Hyperscale and minimal cost service mesh at meta," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, Boston, MA, Jul. 2023, pp. 969–985. [Online]. Available: https://www.usenix.org/conference/osdi23/presentation/saokar

[41] J. Gu, H. Li, W. Li, Y. Xia, and H. Chen, "EPK: Scalable and efficient memory protection keys," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, Carlsbad, CA, Jul. 2022, pp. 609–624. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/gu-jinyu

[42] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "libmpk: Software abstraction for intel memory protection keys (intel MPK)," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA, Jul. 2019, pp. 241–254. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/park-soyeon

[43] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: Secure, efficient in-process isolation with protection keys (MPK)," in *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA, Aug. 2019, pp. 1221–1238. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner

[44] "Write XOR eXecute," https://en.wikipedia.org/w/index.php?title=W%5EX&oldid=1145060869, 2024, [ONLINE].

[45] V. Kiriansky and C. Waldspurger, "Speculative buffer overflows: Attacks and defenses," *arXiv preprint arXiv:1807.03757*, 2018.

[46] D. Evtyushkin, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev, "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*, ser. ASPLOS '18, 2018, p. 693–707. [Online]. Available: https://doi.org/10.1145/3173162.3173204

[47] "microvm virtual platform," https://www.qemu.org/docs/master/system/i386/microvm.html, 2026, [ONLINE].

[48] S. Qi, H.-S. Tsai, Y.-S. Liu, K. K. Ramakrishnan, and J.-C. Chen, "X-io: A high-performance unified i/o interface using lock-free shared memory processing," in *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*, 2023, pp. 107–115.

Authorized licensed use limited to: Politecnico di Torino. Downloaded on June 15,2026 at 08:32:37 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2026.3697819

[49] "Online Boutique," https://github.com/GoogleCloudPlatform/microservices-demo, 2024, [ONLINE].
[50] B. Pfaff *et al.*, "The design and implementation of open vSwitch," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, Oakland, CA, May 2015, pp. 117–130.
[51] "lwIP - A Lightweight TCP/IP stack," https://savannah.nongnu.org/projects/lwip/, 2024, [ONLINE].
[52] "Vhost-user Protocol," https://qemu-project.gitlab.io/qemu/interop/vhost-user.html, 2026, [ONLINE].
[53] S. Qi, Z. Zeng, L. Monis, and K. K. Ramakrishnan, "Middlenet: A unified, high-performance nfv and middlebox framework with ebpf and dpdk," *IEEE Transactions on Network and Service Management*, pp. 1–1, 2023.
[54] "NGINX Service Mesh," https://www.nginx.com/products/nginx-service-mesh/, 2024, [ONLINE].
[55] M. Shahrad *et al.*, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, Jul. 2020, pp. 205–218. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/shahrad
[56] S. Thomas, L. Ao, G. M. Voelker, and G. Porter, "Particle: Ephemeral endpoints for serverless networking," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20, 2020, p. 16–29. [Online]. Available: https://doi.org/10.1145/3419111.3421275
[57] "Locust: A Modern Load Testing Framework." https://locust.io/, 2024, [ONLINE].
[58] H. Fingler, A. Akshintala, and C. J. Rossbach, "Usetl: Unikernels for serverless extract transform and load why should you settle for less?" in *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, ser. APSys '19, 2019, p. 23–30. [Online]. Available: https://doi.org/10.1145/3343737.3343750
[59] G. Li, D. Du, and Y. Xia, "Iso-unik: lightweight multi-process unikernel through memory protection keys," *Cybersecurity*, vol. 3, pp. 1–14, 2020.
[60] V. A. Sartakov, L. Vilanova, and P. Pietzuch, "Cubicleos: A library os with software componentisation for practical isolation," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21, 2021, p. 546–558. [Online]. Available: https://doi.org/10.1145/3445814.3446731
[61] H. Lefeuvre *et al.*, "Flexos: Towards flexible os isolation," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22, 2022, p. 467–482. [Online]. Available: https://doi.org/10.1145/3503222.3507759
[62] K. Yasukata, M. Honda, D. Santry, and L. Eggert, "StackMap: Low-Latency networking with the OS stack and dedicated NICs," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, Denver, CO, Jun. 2016, pp. 43–56. [Online]. Available: https://www.usenix.org/conference/atc16/technical-sessions/presentation/yasukata
[63] L. Rizzo, "netmap: A novel framework for fast packet I/O," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, Boston, MA, Jun. 2012, pp. 101–112. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo
[64] I. Zhang *et al.*, "The demikernel datapath os architecture for microsecond-scale datacenter systems," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21, 2021, p. 195–211. [Online]. Available: https://doi.org/10.1145/3477132.3483569
[65] M. Sung, P. Olivier, S. Lankes, and B. Ravindran, "Intra-unikernel isolation with intel memory protection keys," in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '20, 2020, p. 143–156. [Online]. Available: https://doi.org/10.1145/3381052.3381326
[66] D. Schrammel, S. Weiser, R. Sadek, and S. Mangard, "Jenny: Securing syscalls for PKU-based memory isolation systems," in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, Aug. 2022, pp. 936–952. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/schrammel
[67] V. Narayanan, T. Huang, D. Detweiler, D. Appel, Z. Li, G. Zellweger, and A. Burtsev, "RedLeaf: Isolation and communication in a safe operating system," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, Nov. 2020, pp. 21–39. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/narayanan-vikram



Federico Parola is a Kernel and Hypervisor Engineer at Amazon Web Services (AWS). He earned his Ph.D. in Computer Engineering from Politecnico di Torino, Italy, in July 2024, where he also completed his B.Sc. and M.Sc. (cum laude). His main interests include the Linux kernel, with a focus on the networking and eBPF subsystems, high-performance packet processing through frameworks such as DPDK, and virtualization of resources through VMs, containers and novel paradigms such as serverless and unikernels.



Shixiong Qi is a tenure-track Assistant Professor in the Department of Computer Science at the University of Kentucky. His research spans cloud computing, serverless computing, high-performance networking, cellular networks, and machine learning (ML) systems. He earned his Ph.D. in Computer Science at the University of California, Riverside. His work has been published in top-tier venues such as SIGCOMM, EuroSys. He earned his B.Sc. in Electronic and Information Engineering from Nanjing University of Posts and Telecommunications in 2015 and his M.Sc. in Communication and Information Systems from Xidian University in 2018.



Anvaya B. Narappa received his B.E. in Electrical, Electronics, and Communications Engineering from Vellore Institute of Technology, India, in 2020. He earned his M.S. in Computer Science from the University of California, Riverside, in 2024. His research interests include Linux kernel networking, high-performance packet processing with DPDK, and unikernels. Anvaya is currently a Systems Engineer at SIG.



K. K. Ramakrishnan Dr. K. K. Ramakrishnan is Professor of Computer Science and Engineering at the University of California, Riverside. Previously, he was a Distinguished Member of Technical Staff at AT&T Labs-Research. Prior to 1994, he was a Technical Director and Consulting Engineer in Networking at Digital Equipment Corporation. Between 2000 and 2002, he was at TeraOptic Networks, Inc., as Founder and Vice President. K. K. is an ACM Fellow, an IEEE Fellow and an AT&T Fellow, recognized for his fundamental contributions on communication networks, including his work on congestion control, traffic management and VPN services. He has published nearly 300 papers and has 183 patents issued in his name. K. K. received his MTech from the Indian Institute of Science (1978), MS (1981) and Ph.D. (1983) in Computer Science from the University of Maryland, College Park, USA.



Fulvio Risso received the M.Sc. (1995) and Ph.D. (2000) in computer engineering from Politecnico di Torino, Italy. He is currently Full Professor at the same University. His research interests focus on high-speed and flexible network processing, edge/-fog computing, software-defined networks, network functions virtualization. He has co-authored more than 100 scientific papers.

Authorized licensed use limited to: Politecnico di Torino. Downloaded on June 15,2026 at 08:32:37 UTC from IEEE Xplore. Restrictions apply.
© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.