rkt-io: A Direct I/O Stack for Shielded Execution

Jörg Thalheim[1,2], Harshavardhan Unnibhavi[1,2], Christian Priebe[3], Pramod Bhatotia[1,2], Peter Pietzuch[3]

[1]Technical University of Munich, Germany [2]The University of Edinburgh, UK [3]Imperial College London, UK

Abstract

The shielding of applications using trusted execution environments (TEEs) can provide strong security guarantees in untrusted cloud environments. When executing I/O operations, today's shielded execution frameworks, however, exhibit performance and security limitations: they assign resources to the I/O path inefficiently, perform redundant data copies, use untrusted host I/O stacks with security risks and performance overheads. This prevents TEEs from running modern I/O-intensive applications that require high-performance networking and storage.

We describe rkt-io (pronounced "rocket I/O"), a direct user-space network and storage I/O stack specifically designed for TEEs that combines high-performance, POSIX compatibility and security. rkt-io achieves high I/O performance by employing direct userspace I/O libraries (DPDK and SPDK) inside the TEE for kernel-bypass I/O. For efficiency, rkt-io polls for I/O events directly, by interacting with the hardware instead of relying on interrupts, and it avoids data copies by mapping DMA regions in the untrusted host memory. To maintain full Linux ABI compatibility, the userspace I/O libraries are integrated with userspace versions of the Linux VFS and network stacks inside the TEE. Since it omits the host OS from the I/O path, does not suffer from host interface/Iago attacks. Our evaluation with Intel SGX TEEs shows that rkt-io is 9× faster for networking and 7× faster for storage compared to host- (Scone) and LibOS-based (SGX-LKL) I/O approaches.

CCS Concepts: • **Security and privacy → Trusted computing**; • **Software and its engineering → Operating systems**.

1 Introduction

Cloud computing offers economies of scale for computational resources combined with the ease of management, elasticity, and fault tolerance. At the same time, it increases the risk of security violations when applications run in untrusted third-party cloud environments. Attackers (or even malicious cloud administrators) can compromise the security of applications [72]. In fact, many studies show that software bugs, configuration errors, and security vulnerabilities pose serious threats to cloud systems, and software security is cited as a barrier to the adoption of cloud solutions [71].

Hardware-assisted *trusted execution environments* (TEEs), such as Intel SGX [35], ARM Trustzone [4], RISC-V Keystone [49, 69], and AMD-SEV [3], offer an appealing way to make cloud services more resilient against security attacks. TEEs provide a secure memory region that protects application code and data from other privileged layers in the system stack, including the OS kernel/hypervisor. Based on this promise, TEEs are now commercially offered by major cloud computer providers, including Azure [54], Google [27], and Alibaba [17].

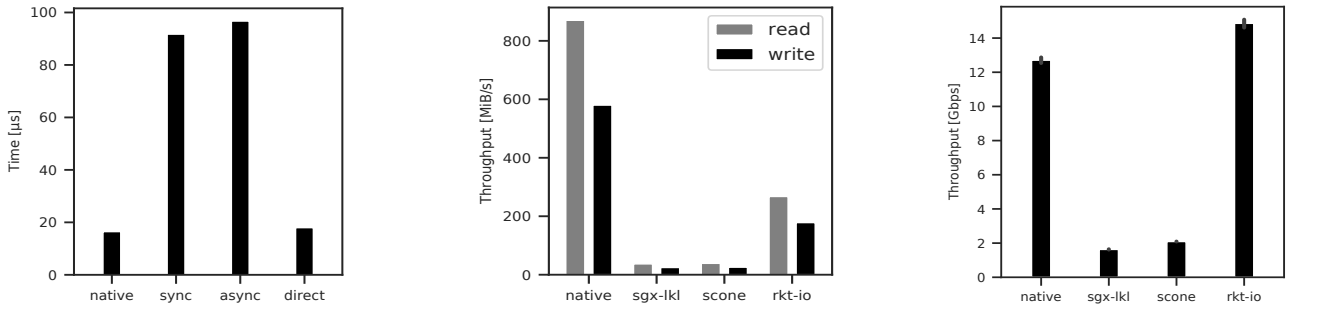
TEEs, however, introduce new challenges to meet the performance requirements of modern I/O-intensive applications that rely on high-performance networking hardware (e.g., >20 Gbps NICs) and storage (e.g., SSDs). Since TEEs are primarily designed to protect in-memory state, they only offer relatively expensive I/O support to interact with the untrusted host environment [20]. Early designs relied on expensive *synchronous world switches* between the trusted and untrusted domains for I/O calls, where a thread executing an I/O operation must exit the TEE before issuing a host I/O system call. This approach incurs prohibitive overheads due to the security sanitization of the CPU state including registers, TLBs, etc.

To overcome this limitation, more recent designs used by shielded execution frameworks (e.g., Scone [5], Eleos [62], and SGX-LKL [66]) employ a *switchless* I/O model in which dedicated host I/O threads process I/O calls from TEE threads using shared memory queues. To avoid blocking TEE threads when waiting for I/O results, these frameworks employ user-level threading libraries inside the TEE to execute I/O calls *asynchronously* [76].

While such *switchless asynchronous* designs improve I/O performance over the strawman *synchronous world switching* design, current frameworks still exhibit significant performance and security limitations: (1) they manage resources inefficiently by requiring dedicated I/O threads outside the TEE, which incurs extra CPU cycles when busy polling syscalls queues. These additional I/O threads also require fine-grained performance tuning to determine their

EuroSys '21, April 26–28, 2021, Online, United Kingdom

ACM ISBN 978-1-4503-8334-9/21/04…$15.00
https://doi.org/10.1145/3447786.3456255



(a) System call latency with `sendto()` **(b)** Storage stack performance with `fio` **(c)** Network stack performance with `iPerf`

Figure 1. Micro-benchmarks to showcase the performance of syscalls, storage and network stacks across different systems

optimal number based on the application threads and I/O workload; (2) they perform additional data copies between the trusted and untrusted domains, and the indirection via shared memory queues significantly increases I/O latency; (3) the untrusted host interface on the I/O path has security and performance issues: the host interface is fundamentally insecure [15, 81], and requires context switches, which are expensive for high-performance network and storage devices; and (4) they lack a universal and transparent mechanism to encrypt data on the I/O path. Instead, they rely on application-level encryption, which is potentially not comprehensive, and incompatible with full VM encryption models.

To overcome these limitations, we argue for a fundamentally different design point where we re-design the I/O stack based on *direct userspace I/O* in the context of TEEs. To exemplify our design choice, we compare the direct I/O approach within TEEs with three alternative I/O approaches, measuring the performance of the `sendto()` syscall with 32-byte UDP packets over a 40GbE link for (i) native (not secured), (ii) synchronous and (iii) asynchronous syscalls within TEEs (secured). As Figure 1a shows, native system calls (16.4 µs) and the direct I/O based approach (17.9 µs) take approximately the same time, while we see higher per-packet processing time for the synchronous (91.7 µs) and asynchronous (96.7 µs) system calls. By bypassing the host I/O support, TEE I/O stacks can avoid performance overheads (and security limitations).

Our design for a TEE I/O stack therefore has the following goals: (a) *performance:* we aim to provide near-native performance by accessing the I/O devices (NICs or SSDs) directly within the TEEs; (b) *security:* we aim to ensure strong security guarantees, mitigating against OS-based Iago [15] and host interface attacks [81]; and (c) *compatibility:* we aim to offer a complete POSIX/Linux ABI for applications without having to rewrite their I/O interface.

To achieve these design goals, we describe *rkt-io* (pronounced "rocket I/O"), an I/O stack for shielded execution using Intel SGX TEEs. The key idea behind the rkt-io design is to combine (a) I/O kernel-bypass libraries (DPDK [24] and SPDK [37]) for direct hardware I/O access with (b) the POSIX abstractions provided by a Linux-based LibOS (LKL [59]) inside the TEEs. This combination results in a high-performance I/O path, while preserving compatibility with off-the-shelf, well-tested Linux filesystems and network protocol implementations inside the TEE. Since the I/O stack runs in the protected domain of the TEE, rkt-io provides improved security, as it does not rely on information from the untrusted host OS.

The design of rkt-io embodies four principles to address the aforementioned limitations of current frameworks:

- rkt-io adopts a *host-independent I/O interface* to improve performance and security. This interface leverages a direct I/O mechanism in the context of TEEs, where it bypasses the host OS when accessing external hardware devices. At the same time, it leverages a Linux-based LibOS (LKL [59]) to provide full Linux compatibility.
- rkt-io favors a polling-based approach for *I/O event handling* since TEEs do not provide an efficient way to receive interrupts on I/O events.
- rkt-io proposes a sensible *I/O stack partitioning* strategy to efficiently utilize resources and eliminate spurious data copies. It partitions the I/O stack by directly mapping the (encrypted) hardware DMA regions into untrusted memory outside the TEE, and runs the I/O stack within the TEE.
- rkt-io provides *universal and transparent encryption* in the I/O stack to ensure the confidentiality and integrity of data entering and leaving the TEE. It supports Layer 3 network packet encryption (based on Linux's in-kernel Wireguard VPN [23]) for networking, and full disk encryption (based on Linux's `dm-crypt` device mapper [22]) for storage.

Our evaluation with a range of micro-benchmarks and real-world applications shows that rkt-io provides better performance compared to Scone (a host-OS based approach) and SGX-LKL (a LibOS-based approach). For example, the read/write bandwidth of rkt-io's storage stack (measured by `fio` [40]) is up to 7× higher (see Figure 1b), and the throughput of rkt-io's network stack (measured by `iPerf` [39]) is up to 9× higher (see Figure 1c).

rkt-io is publicly available (see Artifact appendix A).

2 I/O support in TEEs

TEEs provide the ability to create hardware-assisted protected domains in a process address space, as shown in Figure 2. TEEs protect the confidentiality and integrity of the application's code and data inside the TEE. It is also possible to verify the integrity of the code running inside the TEE via remote attestation. This enables users to run security-sensitive workloads in an otherwise untrusted execution environment.

2.1 Threat model

Our threat model extends the standard threat model for TEEs [5, 7]. As in the prior work, we assume a powerful adversary who has control of the entire system software stack, including the host OS and the hypervisor. In line with previous work, we do not address the physical tampering of the CPU package, denial of service attacks, memory safety, and side channel attacks [12, 28, 29, 86, 90].

For I/O operations, applications running inside the TEE rely on the untrusted host OS for access to I/O devices, such as NICs and SSDs. On the I/O path, an adversary may tamper with the data being exchanged through the untrusted host interface, and compromise the confidentiality and integrity of the application running inside the TEE [10, 14, 15, 50, 88]. More specifically, the host interface may leak sensitive data, also known as interface attacks [81], which can expose the application state to the untrusted host.

In addition, the host interface implementation can be malicious itself, and therefore compromise the security of the application running inside the TEEs e.g., by manipulating the return values of syscalls, also known as Iago attacks [15]. Further, we assume that our I/O stack is resilient to memory safety vulnerabilities [47, 61].

Lastly, an attacker may use software/hardware probing to intercept data on the host's I/O path by DRAM interface snooping, installing malicious hardware with DMA access, or performing cold boot attacks. Only universal end-to-end encryption of data on the I/O path can mitigate these types of attacks.

2.2 Analysis of existing I/O mechanisms

A protected application within the TEE communicates with the outside environment by performing I/O operations for accessing the filesystem or network stack. To support the I/O operations with the untrusted environment, TEEs require a *world switch*, where a thread executing the I/O operation switches between the trusted and untrusted domains, and then issues the syscall to the host OS. I/O operations, when invoked through the synchronous syscall mechanism, add a constant world switch overhead, incurred due to the necessary micro-architectural security-associated sanitizations, such as additional cache/TLB flushes, page permission checks, etc. [20]. For example, a world switch costs ≈10,170 cycles, which is roughly 5× more expensive than a syscall (≈1800 cycles) on our hardware setup (§6).

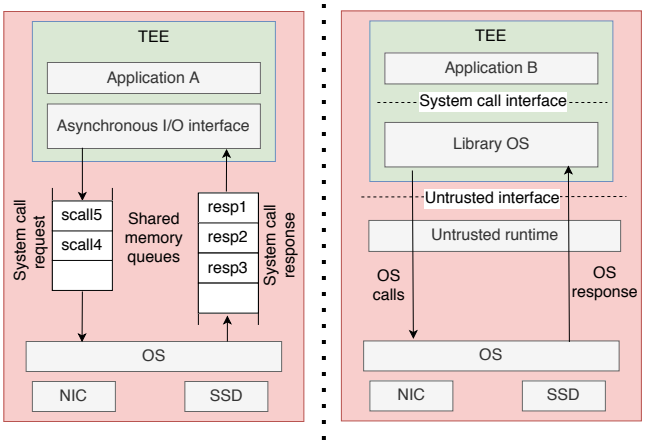


Figure 2. Two possible shielded execution architectures for I/O support in TEEs: *(left)* application A uses a pure host OS based approach, and *(right)* application B uses a LibOS inside the TEE to process the I/O operations. (Regions in green are trusted, whereas red regions are untrusted.)

To avoid the costly world switches between the trusted and untrusted domains, current shielded execution frameworks [5, 7, 36, 62, 66, 85] have adopted *switchless* designs, which can be broadly categorized as (see Figure 2): (i) host-based; and (ii) LibOS-based. We compare these approaches across three dimensions: performance, security, and compatibility.

(1) Host-based frameworks (e.g., Scone [5], Eleos [62], Intel SGX SDK [36]) rely on the host OS for the I/O operations. Among these frameworks, we focus on Scone as the state-of-the-art system for our baseline. Scone improves the I/O performance by leveraging the concept of asynchronous system calls [76]. In the asynchronous model, a set of dedicated I/O threads run (busy polling) outside the TEE to process the syscall requests issued by a thread from inside the TEE via shared memory queues.

(2) LibOS-based frameworks (e.g., Haven [7], Graphene-SGX [85], SGX-LKL [66]) rely on customized LibOSs inside the TEE for handling I/O operations. Note that these LibOSs still use the underlying host OS in the backend (via the untrusted host-based syscall interface) to access the hardware devices. As far as the I/O path is concerned, Haven relies on synchronous mode for I/O operations, where the I/O threads block for the request completion. Whereas, Graphene-SGX provides synchronous syscalls by default, but it also supports asynchronous mode [76]. SGX-LKL uses an *asynchronous* I/O mechanism also and implements the full Linux ABI in its interface, while Graphene provides a subset. Therefore, we focus on SGX-LKL as the state-of-the-art system for our LibOS baseline.

Performance. Both approaches use the resources on the I/O path inefficiently: (1) they rely on dedicated *I/O threads* for issuing the I/O calls. This incurs extra CPU cycles due to the busy polling of the syscalls queues; (2) they require fine-grained tuning to set the optimal number of I/O threads

due to the tight coupling with the application threads and I/O workload; (3) they require additional data copies—data needs to be copied from the TEE to the untrusted host memory for the asynchronous I/O threads, before it is processed by the underlying host OS, requiring another copy; and (4) both approaches incur latency penalties due to the indirection involved on the asynchronous I/O path.

Security. Both approaches depend on the untrusted host OS, which make them vulnerable to Iago attacks [15] and host-interface attacks [81], but they differ with respect to the degree of dependence on the host OS: the host-based approach exposes a wider interface with the untrusted host OS by allowing protected applications to directly issue a large set of syscalls. Although syscall return values are sanitized by network and file systems *shielding layers*, they are susceptible to more attacks due to the increased interactions with the untrusted host OS.

In contrast, the LibOS-based approaches provide better security since they expose only a limited set of syscalls to the untrusted host OS. Furthermore, since the LibOS-based approaches are flexible to adopt, a custom LibOS to fit the application requirements can be developed. Hence a protected application can further improve its security by adopting a LibOS with a smaller TCB.

Compatibility. Host-based approaches can provide POSIX or Linux ABI compatibility, allowing them to support unmodified legacy applications. For instance, Scone can support off-the-shelf filesystems and network stacks based on Linux. On the other hand, LibOS-based approaches offer the possibility of specialization at the cost of limited or no support for the existing filesystem and network stacks. In general, this makes them less amenable for supporting unmodified legacy applications, with a notable exception of SGX-LKL that offers full POSIX/Linux ABI compatibility by using a library version of the complete Linux kernel (LKL [59]).

2.3 Problem statement and approach

In this work, we aim to build a high-performance, secure, and compatible I/O stack for shielded execution. For performance, we want to improve latency and throughput of I/O operations compared to a switchless asynchronous I/O approach. We also want to minimize reliance on the untrusted host for improved security. Lastly, we strive for full compatibility for existing applications by supporting the Linux ABI/POSIX standard.

To achieve these goals, we next summarize our high-level approach and associated *four design principles*.

#1: Host-independent I/O interface. Current host OS- and LibOS-based shielded execution frameworks rely on the underlying host OS for I/O operations. Instead, we argue for a fundamentally different design point, in which we favor a host-independent I/O interface that uses direct I/O with TEEs. A direct I/O approach improves performance and security: compared to a switchless asynchronous syscall mechanism, it reduces the latency and increases the throughput of I/O operations by directly accessing the hardware (NICs and SSDs) and minimizing the number of data copies. Since direct I/O minimizes the host OS interactions as much as possible by accessing the I/O hardware directly from the TEE (i.e., there are no I/O-related syscalls after the initialization phase), it also leads to improved security. We combine this approach with a Linux LibOS (LKL) inside the TEE to provide full Linux ABI compatibility. We primarily target applications not written for SPDK/DPDK. Applications with SPDK/DPDK support have been addressed in the context of enclaves in previous work [6, 83]. The performance in previous work matched those of native SPDK/DPDK applications when encryption was disabled, however missed some features i.e. no TCP/IP stack (only layer2) in Speicher [6] or no filesystem support (only block layer) in Shieldbox [83]. While LibOSs provide such APIs, a naive DPDK/SPDK port then does not offer performance advantages and may be slower than using traditional OS functionality. In this context, the rkt-io design shows the required changes to multiprocessing, threading, timer and I/O event handling, and network polling that are needed to make the best use of hardware resources.

#2: I/O event handling. In the context of SGX, we cannot rely on the interrupt-driven I/O execution because there is no efficient way to receive interrupts or timer events within TEEs. Instead of interrupt-based I/O, rkt-io uses a polling-based approach for handling I/O events in TEEs in which rkt-io explicitly polls I/O response queues for completed requests or new data. Such an approach for I/O event handling is a natural fit with direct I/O libraries (SPDK/DPDK) that combines polling with the run-to-completion model for fast I/O devices, which in turn avoids the performance bottlenecks of interrupt-based execution [8, 42, 63].

#3: I/O stack partitioning. While direct I/O libraries fit better with I/O event handling, their adoption in the context of TEEs presents an interesting challenge: DMA regions for untrusted I/O devices cannot be mapped directly into the TEE as DMA access is prohibited for security reasons. We therefore need a way to efficiently write to and read from a DMA region. rkt-io achieves this by sensibly partitioning the direct I/O stack into two parts: the driver code for I/O stacks runs inside the TEE, and DMA memory regions for I/O devices are outside, as part of the untrusted host memory.

#4: Transparent encryption. Since we cannot trust the host, network or storage hardware, all data leaving the TEE must be encrypted to ensure confidentiality, which is typically handled at the application layer in today's frameworks. Unfortunately, such an approach is error-prone, as applications may not universally encrypt all of its I/O paths, e.g., exposing data through unencrypted legacy network protocols or file systems. Instead, rkt-io supports a transparent and universal mechanism to provide full disk (block layer) and network encryption (Layer-3) by relying on the LibOS.

3 Architecture of rkt-io

Figure 3 shows the high-level architecture of rkt-io. It consists of: (a) a *network stack* that is derived from the Linux kernel exposing a socket API and is backed by the Data Plane Development Kit (DPDK) [24]. Using DPDK, rkt-io gets direct access to the NIC from within the TEE; (b) a *storage stack* that provides a complete filesystem abstraction (e.g., the Linux ext4 filesystem). It uses the Linux VFS layer to interact with the block device layer, which is implemented by the Storage Performance Development Kit (SPDK) [37] over the NVMe protocol to communicate with the SSD; and (c) a *runtime environment* that integrates the storage and network stacks based on the Linux kernel library (LKL), a userspace LibOS port of the Linux kernel [59]. LKL provides Linux system calls as userspace function calls inside the TEE. A modified version of the musl standard C library (libc) [55] exposes a POSIX interface to the application on top of the LKL system call interface.

An application can be built with common toolchains/package managers and put into a Linux ext4 filesystem image. At runtime, a loader sets up the TEE, the I/O stacks, the LKL LibOS, and then mounts the filesystem image as the root file system. After that, the application and its linked libraries are loaded into memory from the root file system, and the application can now use rkt-io's modified musl libc library.

A typical I/O request issued by the application begins with a system call, via the libc API. The system call request is processed by the LibOS within the TEE. Depending on whether the request is made to a network or storage device, the LibOS issues calls to the userspace driver via the network and storage stacks respectively. The userspace drivers add requests to the appropriate request queues (transmission queue (Tx) for NIC; submission queue (Sq) for NVMe), which are mapped into the untrusted DMA memory region. Likewise on the receive path, the drivers continuously poll the completion queues (receive queue (Rx) for NIC; completion queue (Cq) for NVMe), which are also mapped into the untrusted region. The userspace drivers notify the library OS on response completion and can return the data if requested.

Next we explain the two building blocks of rkt-io: (a) the direct kernel-bypass I/O mechanism; and (b) the POSIX abstraction/Linux ABI from the Linux-based LibOS.

Direct I/O libraries. Our I/O stack incorporates direct kernel-bypass I/O libraries in the TEE to avoid system calls and directly access the I/O devices. With fast I/O devices, such as fast NICs and NVMe SSDs, context switches and extra data copies make the OS kernel a performance bottleneck [8, 42, 63]. This has given rise to solutions that manage requests made to the underlying hardware in userspace via kernel-bypass. Our work builds on two popular userspace direct I/O libraries, DPDK [24] and SPDK [37], which support network and storage devices respectively.

A direct I/O approach in the context of TEEs is both advantageous and disadvantageous. On one hand, its

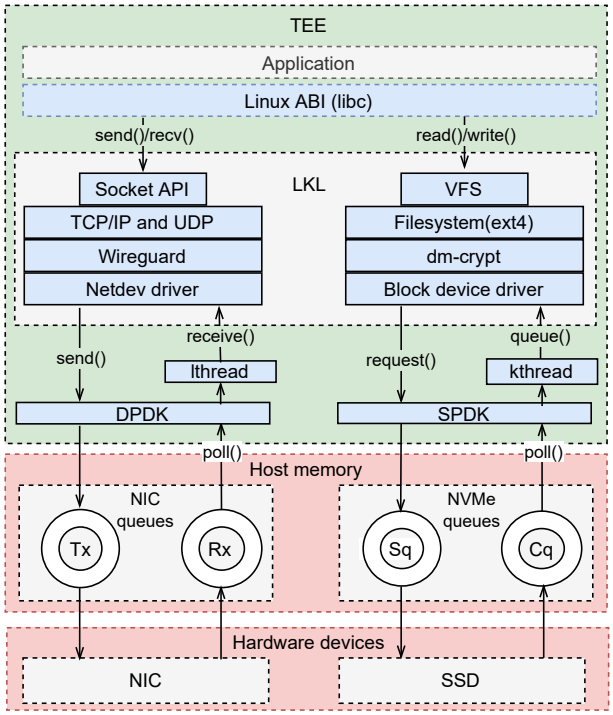


Figure 3. Architecture overview of rkt-io (network stack on left; storage stack on right)

polling-based approach is well-suited for TEEs because interrupts are not permitted within TEEs. Furthermore, polling for completion reduces the total latency, and has been shown to lead to a better design for high-performance I/O devices (NICs and SSDs) [8, 42, 63]; on the other hand, a direct I/O (zero-copy) philosophy is fundamentally incompatible with TEEs, because a DMA memory region cannot be mapped directly inside the TEE due to security restrictions.

To overcome this limitation, rkt-io adopts a split architecture in which driver code for DPDK and SPDK runs inside the TEE, but it maps the DMA memory regions for the NIC/NVMe queues outside the TEE in untrusted host memory. More specifically, the DPDK driver polls the NIC for received packets, which are then explicitly copied into the TEE from the DMA regions. Likewise, the NVMe driver uses its highly parallel asynchronous, lockless and poll-for-completion design to access the underlying SSD. The drivers map hardware queues and PCIe registers into the DMA region, and adds requests and poll responses to distinct queues. Thereby, our design follows a "one-copy" approach that copies the data between the untrusted DMA region and the TEE.

Although DPDK and SPDK help improve the performance and security of applications inside TEEs, it is challenging for developers to use them due to their low-level I/O interfaces. DPDK provides high-speed packet processing capabilities at Layer 2 in the network stack; SPDK offers only a block layer interface (and a rudimentary file system called BlobFS [11]). These low-level interfaces are not sufficient for

most applications, which rather need a full network stack (e.g., TCP/IP) and full filesystem (e.g., ext4) support.

LibOS with Linux ABI. rkt-io uses the Linux Kernel Library (LKL) [59] to provide a mature POSIX implementation with a virtual file system (VFS) layer and a TCP/IP stack. LKL is a complete architecture port of Linux to the userspace, which provides components such as the kernel page cache, work queues, filesystem and network stacks, and crypto libraries.

In our design, the application and the LKL LibOS run in a single virtual address space within the TEE. rkt-io thus avoids user/kernel context switches, as system calls are invoked through functions calls, and it also eliminates data copies between the user/kernel space. By combining LKL with DPDK/SPDK, applications do not need to be modified to use low-level I/O APIs and instead can use POSIX APIs, while taking advantage of the performance and security guarantees offered by rkt-io.

In addition, rkt-io leverages LKL to provide universal and transparent encryption to ensure the confidentiality of data entering and leaving the TEE. rkt-io supports Layer-3 network packet encryption based on Linux's in-kernel Wireguard VPN [23], and full disk encryption based on Linux's `dm-crypt` device mapper [22].

Trusted computing base (TCB). There is an implicit trade-off between the TCB size and the exposed attack surface through the host interface (e.g., for Iago attacks and TEE data leakage). We incorporate DPDK/SPDK within the TEE for improved performance at the cost of a larger TCB. Our design provides better security properties compared to host OS-based designs, which are prone to Iago attacks or host-TEE interface leakage. This is achieved by handling all system calls directly inside the enclave. rkt-io only exposes low-level I/O operations to the untrusted hardware while sanitizing responses. Furthermore, I/O operations can be encrypted and authenticated to make it harder to provide malicious inputs. There is also scope to further minimize/harden the TCB by rewriting parts of DPDK/SPDK, but we consider this beyond this work.

4 Detailed design of rkt-io

We next present the detailed architecture of rkt-io around the four design principles from §2.3.

4.1 Host-independent I/O interface

rkt-io's design aims to provide support for I/O operations while reducing dependencies on the host OS. After boot up of the TEE environment, rkt-io loads the user-provided application and its dependencies into the encrypted memory. It provides its own ABI-compatible variant of the musl libc implementation, which makes system calls against the integrated LKL LibOS — a non-MMU Linux architecture port.

Multi-threading and scheduling is implemented in rkt-io's libc: it implements cooperative userland threads that are scheduled on a fixed number of host OS threads. The userland

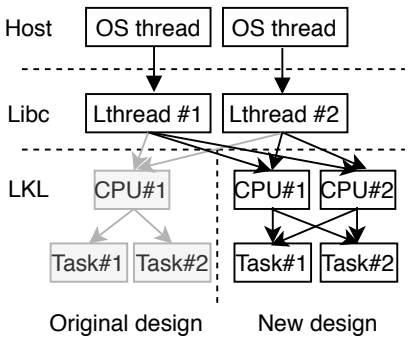


Figure 4. rkt-io SMP architecture

threads yield control and allow other threads to be scheduled on the same host OS thread when they are blocked, e.g., when locks are taken; when a thread sleeps; or when a blocking system call against the LibOS kernel is issued. While pure busy loops thus cannot be preempted, we did not encounter this to be a problem in real-world applications. To build the host-independent I/O interface, we next discuss three main design issues that rkt-io addresses to adapt LKL for high-performance networking and storage.

Symmetric multiprocessing (SMP).

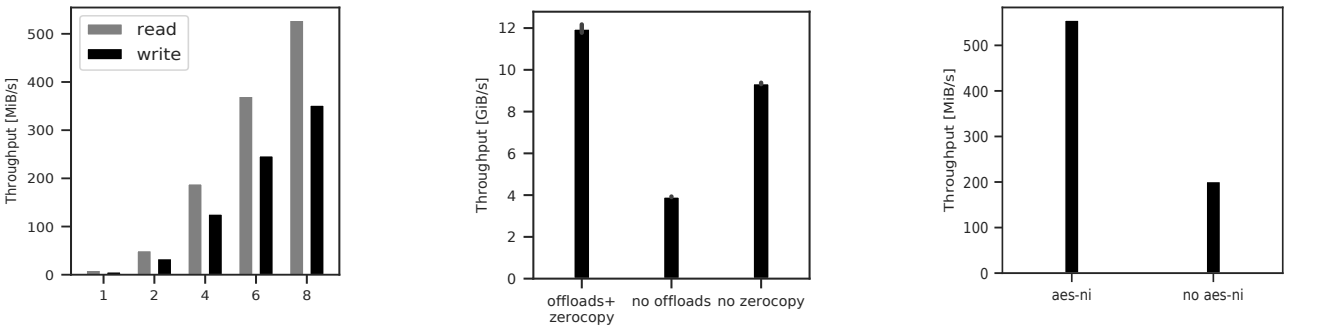
To allow high-performance I/O operations, the I/O stack must be parallel to take advantage of SMP. By default, LKL does not support multi-threading, as shown in Figure 4 (original design on the left). When multiple threads attempt to enter the kernel context, they need to obtain a single lock. This lock protects the data structures associated with a virtual CPU. This creates a bottleneck in the LKL kernel as the backend I/O drivers and most applications are parallel.

To make the kernel scalable, we modify LKL to add SMP support. With that, LKL can provide multiple virtual CPUs as shown in Figure 4 (new design on the right). The threading primitives required by the kernel for SMP are adapted from the native architecture (i.e., x86 in our prototype). This change also introduces additional kernel threads that are needed to handle inter-process interrupts and timer events that are broadcast to multiple virtual CPUs.

To evaluate the effectiveness of our SMP design, we use `fio` [40] with random read/write requests on a 1 GB file while increasing concurrency. Figure 5a shows that the throughput for the storage stack increases linearly with more threads (from 1 to 8 threads) for both read and write requests.

Threads stack management. With an SMP architecture, the number of threads in LKL also increases. To implement threads, LKL uses its OS-specific host interface. In its default implementation for a POSIX-compatible OS, LKL creates a POSIX thread with the architecture's default stack size (8 MB on x86-64). In an environment in which the OS has MMU support, the stack is only backed by physical memory as it grows in size.

rkt-io, however, has no MMU support and must therefore pre-allocate stack memory. This results in a significant



(a) Effectiveness of the SMP design with `fio` and increasing number of threads

(b) `iPerf` throughput with optimizations

(c) Effectiveness of hardware-accelerated crypto routines

Figure 5. Micro-benchmarks to showcase the effectiveness of various design choices in rkt-io

memory overhead when implementing SMP, as more threads consume proportionately higher physical memory. This is an issue given that TEE technologies such as SGX have limited physical memory (≈94 MB in x86 SGX enclaves) that are usable without costly paging operations.

To solve this issue, we reduce the kernel thread stack size to 8 KB, which is the same stack size that the Linux kernel uses for *x*86_64. For DPDK/SPDK, this stack size, however, is too small due to its heavy use of inlined functions. Nested inlining causes more local variables to be stored on the stack, as variable lifetimes increase, which we therefore reduced by removing inline compiler directives.

This approach for thread stack management is extremely effective: each CPU allocates kernel workers and each subsystem spawns dedicated threads (filesystem, diskmapper, TCP/IP, etc.). We observe 155 kernel threads for a single-threaded application with 8 virtual LKL cores. By switching from 8 MB to 8 KB stacks, we save 1.2 GB of memory.

Event scheduling timer. An I/O stack relies on timer events for several periodic tasks, e.g., to flush out dirty pages, to schedule TCP re-transmissions, etc. In our experiments, the original timer support in LKL is too slow for our design because it would only schedule 1–3 events per second. The slow timer is not an issue for the native LKL storage and network drivers, because they only delegate I/O requests to threads that execute host system calls, making it less reliant on periodically scheduled tasks. In rkt-io, however, timer events are used to periodically poll I/O devices, which makes them a bottleneck for scheduling tasks. Therefore, we need to design a new timer implementation to meet the scheduling requirements on the direct I/O path.

Originally, LKL implements a one-shot timer interface in which the kernel registers functions to be called after a certain time by creating a thread per event. The thread sleeps for a given time interval before invoking the kernel callback. rkt-io implements a periodic timer instead. With a frequency of 50 Hz, it calls a generic interrupt function, and the kernel checks which tasks must be executed within this tick. To do so, a single thread is created once, which performs a sleep system call in a loop before notifying the kernel. With this new design, the polling mode I/O stack becomes able to handle I/O events at a high rate.

4.2 I/O event handling

We design our I/O stack based on polling, and therefore we must re-design the LibOS's network and filesystem interfaces to support this. We achieve this by mapping registers and DMA memory regions into rkt-io's virtual address space.

Rather than relying on conventional interrupts to get I/O completion events, which would force context switches and exit the TEE, rkt-io uses polling. While polling can consume more CPU cycles than interrupt handling, especially in I/O-intensive applications, interrupts become a bottleneck. There is a recent trend in OS design to switch to hybrid polling/interrupt approaches to meet the performance requirements of network and storage hardware [25, 92]. We next explain our polling-based design, which we find most suitable for both block (SSDs) and network (NICs) devices.

Block device polling. Figure 3 (right) shows the data path when applications access the filesystem. System calls issued by the application are first dispatched by the virtual filesystem and then delegated to the actual filesystem (in our experiments, ext4). When the filesystem reads/writes data from the underlying block device, the data is cached in the page cache.

The SPDK driver puts incoming requests in the NVMe queue. When queuing requests, the driver also polls for completed requests in the corresponding completion queue. If there are outstanding requests, it schedules a polling task. The polling task periodically polls the completion queue until all outstanding requests are acknowledged and notifies the kernel about each completed item.

In an SMP environment, single hardware queue pairs can easily become a bottleneck due to lock contention, which is caused by multiple threads trying to issue requests concurrently. To overcome this problem, the NVMe standard allows the creation of multiple request/response queue pairs. This allows I/O requests to be issued in parallel, while improving

data locality in NUMA systems. We assign one queue pair per virtual LKL CPU and bind one polling thread to each.

As described in §4.1, LKL has the concept of virtual CPUs, which are protected by locks, so that only one thread can access them at a time. Due to this, rkt-io does not need additional locks around its own queues, as they are already protected by the CPU locks. One challenge when introducing multiple queues is to not increase the CPU overhead due to polling: too much polling on a particular queue steals CPU cycles from the application or from other queues; while not enough polling increases latency and decreases throughput. rkt-io puts the polling threads to sleep if no outstanding requests are due.

An advantage of rkt-io's design with one queue-pair per CPU is that it can also safely poll without locks in the request function, because the actual poll thread cannot run at the same time on the same CPU. This reduces context switches, as the request function is often called from the application thread during a system call.

Network device polling. Similar to the storage stack, the network stack also relies on polling. Figure 3 (left) shows the data path for networking. An application can use the full POSIX socket API with all extensions, as supported by Linux. New data sent by the application is stored in a kernel-side socket buffer, and the socket buffer is placed in a software queue. On a software interrupt, buffers from this queue are passed to the DPDK-based network driver, which puts the data into the NIC's transmission queue. Packets are received by a dedicated polling thread.

While implementing the network stack, we experimented with different setups on how to manage polling. Our first design used multiple queues for sending and receiving. This approach, however, makes network throughput worse: each receiving queue must be polled by a dedicated polling thread, which takes too many CPU cycles away from the application and increases latency. Likewise, for sending queues, rkt-io needs to poll the completion status, as the Linux kernel uses this information to adjust its TCP window size.

Ultimately, we decide on having a single thread dedicated to polling a single queue, which is faster, because the cost of context switches exceeds the cost of polling. To reduce the overhead of scheduling the polling thread, we move it out of the kernel scheduler into the underlying userland scheduler. Before it starts polling, it acquires an LKL CPU lock to get ownership, so it can safely access Linux kernel data structures. After it has processed every received packet, it releases the ownership of the CPU lock.

Bufferbloat mitigation through eager queue cleanup. To counter bufferbloat [13], network congestion avoidance for the TCP stack in Linux measures how many packets are still queued by the NIC. In the original DPDK driver design, however, old packets are not freed as soon as packets are sent, but when new packet buffers override the old entries in the send ring buffer. This is too late for Linux, where the critical

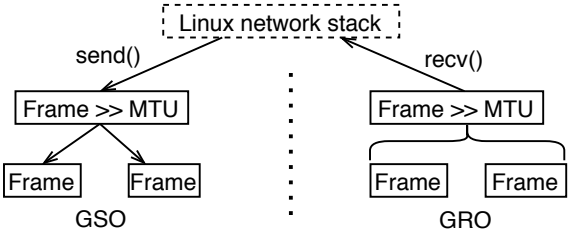


Figure 6. Generic segmentation offload and generic-receive offload

threshold is around 0.5 MB while a send queue in a NIC is significantly larger (i.e. 8 MB for our NIC). As a result, connections are throttled to a rate below 1 Gbps on a 40 Gbps NIC.

To counter that, we redesign the queue cleanup algorithm to free old send buffers when new packets are queued for sending or when the Linux network stack's threshold for a TCP connection is exceeded. In our experiments, this boosts `iPerf` throughput from 300 Mbps to 25 Gbps without encryption and 15 Gbps with encryption. This is close to the native performance for `iPerf`, which reach 26Gbps per second without encryption. `iPerf` does not achieve the full line rate in both cases since it runs singled threaded.

NIC offloading support. To achieve high throughput when processing packets, the offloading mechanism available in modern NICs must be used by a high-performance network stack. When network streams are sent, they need to be divided into smaller Ethernet frames with a maximum transfer unit (MTU) of commonly 1500 bytes. The smaller Layer-3 packets (i.e TCP/IP) need new updated headers to reflect the changed size, sequence number and checksum. On the receiver side, it is also typically too expensive to traverse the whole network stack for each packet. Optimizations in either software or hardware are required to re-assemble the payload from smaller TCP packets into larger buffers. In addition, the payload and network headers are often not stored continuously in memory. This needs to be communicated to the NIC to avoid having to copy parts to a new continuous buffer.

To address the problem of assembling and disassembling network streams, we use generic segmentation offloading (GSO) and generic receive offloading (GRO). GSO, as shown in Figure 6 (left), requires implementation changes in DPDK to allow for the segmentation of large network streams into smaller Ethernet frames in the hardware. For GRO, as shown in Figure 6 (right), we rely on a software solution, as described in §5.2, to avoid entering the network stack for each small packet by aggregating them in larger buffers. Furthermore, rkt-io configures the NIC to offload checksum computation for network headers. We also make use of DPDK's segmentation support to make the NIC read headers and payload chunks from different memory locations, thus avoiding copying them to a new buffer.

To evaluate the effectiveness of these offloading techniques, we run `iPerf` with and without them, as shown in Figure 5b,

with a single TCP stream. As for all network benchmarks, TLS is enabled in `iPerf`. With offloading, we obtain a 3× higher throughput, making DPDK's performance comparable to NIC-enabled offloading in the Linux kernel.

4.3 I/O stack partitioning for TEEs

Since storage and network devices cannot directly access TEE memory, their DMA memory regions need to be mapped outside of the TEE. Conversely, the POSIX API forces the kernel to make a copy of the data passed in a system call because applications expect the memory to be re-usable. A naive implementation would therefore do two copies: one from the application buffer to the kernel and one from the kernel to the NIC DMA region.

rkt-io reduces the number of copies to one, by copying data for sending directly from the application buffer to the hardware's DMA region. In systems that have access to an MMU, usually from a privileged ring, this can be achieved by re-mapping pages in virtual memory. rkt-io, however, runs in unprivileged userspace and instead extends the Linux kernel memory allocator to support memory allocations in both encrypted TEE memory as well as unencrypted DMA memory (see §5.1).

One-copy for networking. This support in the Linux kernel memory allocator allows rkt-io to allocate the data part of a socket buffer (short skb) in the NIC DMA memory. In turn, rkt-io makes use of DPDK's external buffer support [21] (see §5.2) to transfer packets to the NIC without an extra copy.

To evaluate the performance improvements of a one-copy data path, we use the same `iPerf` benchmark as in §4.2 with the result shown in Figure 5b. When comparing all optimizations enabled with disabling the copy optimization in the receive/send path, we see a 21% improvement (11.6 Gbps vs. 15 Gbps).

One-copy for storage. Similarly, the NVMe device needs data to be written to a special DMA memory region in which NVMe queues are allocated. To avoid extra copies of the encrypted pages from the disk encryption layer, rkt-io allocates those pages in the DMA memory outside of the TEE. When transferring pages from or to the NVMe device, rkt-io uses the gather-scatter API of SPDK. This API allows to pass I/O vectors instead of continuous buffers, which is needed to pass multiple scattered kernel pages directly to the hardware. This optimization results in a throughput improvement of 7% for the block device.

4.4 Transparent encryption

Since all data that leaves the TEE must be protected to avoid information leakage to the host, rkt-io implements both transparent encryption of network traffic using a Layer 3 virtual private network (VPN) as well as full disk encryption.

For network protection, we find that many network-facing applications already support transport encryption using TLS. This is the preferred way, as it provides high-throughput and low protocol overhead thanks to highly optimized TLS stacks, such as OpenSSL [87]. If an application does not support TLS, rkt-io also supports the Wireguard VPN [23] to encrypt network packets on Layer 3. It is integrated into the LibOS as a tunnel, and it encapsulates encrypted IP packets into the UDP protocol using the ChaCha20 [48] stream cipher before forwarding them to the NIC.

For storage protection, we use the Linux `device mapper` crypto target that gives full disk encryption. It is set up to use AES 256-bit in XTS cipher mode before passing encrypted pages to the underlying block device.

Both network and block-based encryption are optional and can be disabled. For example full disk encryption can be only enabled for certain devices and network routes can be configured to bypass the wireguard VPN.

Hardware acceleration. The LKL architecture by default only provides slow generic routines for AES encryption or cryptographic hashing, which makes full disk encryption slow. Optimized routines must be loaded from kernel modules, depending on which CPU extensions are available (i.e. AES-NI on Intel x86). rkt-io ports the crypto modules from the respective native CPU architecture (i.e. x86) to speed-up block disk encryption. Therefore, we implement kernel module loading support (see §5.3).

Figure 5c shows the throughput before and after enabling hardware-accelerated cryptographic routines for sequential writes to a 10 GB file. Enabling acceleration increases throughput by 2.8×.

5 Implementation of rkt-io

The implementation of rkt-io is based on SGX-LKL [66]. SGX-LKL provides the musl libc abstraction, userland threading and the integration with LKL. rkt-io extends SGX-LKL to support the direct I/O network and storage stacks. It also re-designs several components for improved I/O performance, as introduced in §4 and explained in further detail below.

5.1 Runtime environment

Driver setup. DPDK/SPDK configure the system to map hardware queues into rkt-io's virtual address space. This is a privileged action that requires root permissions. rkt-io delegates this task to a dedicated `setuid` binary, so that the actual SGX enclave can run with regular user privileges. For this to work, we use DPDK's multi-process feature in which the privileged process acts as a primary and communicates over shared memory with the enclave, which runs as a secondary process. In addition, DPDK/SPDK needs root access to resolve its own virtual addresses to physical addresses for communication with the hardware. We delegate this task to another `setuid` binary, which provides this service over a pipe.

Hugetables. DPDK/SPDK allocate the memory used for communication with the hardware as huge pages (either 4 MB

or 1 GB instead of 4 KB on x86). rkt-io uses this huge memory region as a page cache, which is why it allocates as many huge pages as possible. We find that, with the default 1 GB page size recommendation of DPDK, this was not possible. The host OS page allocator causes memory fragmentation, and it cannot find sufficiently many unused continuous 1 GB physical pages (only 4–5 GB of pages on a system with 32 GB RAM in our experiments). Instead, we modify DPDK to allocate 4 MB pages and align them continuously in memory by moving DPDK's metadata structure to a different offset. This way the page allocator that runs in our LibOS can treat this memory as a one continuous chunk.

Page allocator. We extend the Linux page allocator to use DPDK/SPDK memory. The Linux kernel expects page data structures and cannot work with external buffers. We therefore re-used page flags used in the NUMA architecture to differentiate between memory allocated in the TEE and memory allocated in the DMA memory region. On top of that, we can also identify pages based on their addresses for extra security checks, e.g., whether the memory comes from protected TEE memory or the unencrypted DMA memory region.

We extend LKL to register DPDK/SPDK memory in its own "NUMA" zone on bootup. By default, the kernel never allocates memory in these zones except when a special flag (`GFP_SPDK_DMA`) is passed to the page allocator function. We also add DMA memory support for `kmalloc`, which is the kernel's `malloc` equivalent. It builds on top of the page allocator by adding additional caches for different size classes. We added new cache data structures for the DMA memory region and make kmalloc select them if the `GFP_SPDK_DMA` flag is set.

5.2 Network stack

rkt-io implements a new network device driver to integrate DPDK into the Linux network stack. We make several modifications to DPDK itself to improve the performance. To improve the TCP send performance and make DPDK competitive with the native Linux kernel driver, we implement Generic Segmentation Offload (GSO) for the Intel 40-Gigabit Ethernet NIC family (called i40e in DPDK/Linux). On the receiving side, we implement Generic Receiving Offload (GRO) using the kernel `napi_gro_receive()` function. This shortcuts parts of the network stack as packets get summarized into larger streams, without traversing the full stack for each packet.

By default, DPDK comes with its own allocator for packet buffers. To avoid copying when transferring packets from Linux's socket buffer to the NIC, we make use of DPDK's external buffer support using the *rte_pktmbuf_attach_extbuf()* function. For receiving packets, DPDK does not offer support for external buffers, so we modify DPDK to allocate Linux socket buffers rather than its own packet buffers. This modification was only implemented for the Intel NIC driver (i40e) since the allocation happened within the DPDK driver. This limits our current prototype to this specific class of devices, but this can be extended in the same way to all device drivers supported by DPDK [79].

Devices in DPDK cannot be shared with other enclaves. However many data-center NICs often come with a feature called SR-IOV to make one device appear as multiple physical interfaces, which can then be assigned to different enclaves.

5.3 Storage stack

rkt-io implements a multi-queue block device driver using the `blk-mq` [19] interface to integrate SPDK as a block device into LKL. The driver uses the NVMe interface of SPDK, hence we can support any block storage device that implements the NVMe protocol. In our prototype we mainly tested PCIe-connected devices. rkt-io assigns one queue-pair per CPU that is used when doing requests in our driver's `queue_rq` implementation to avoid locks between different CPUs. In the same function, we also poll for outstanding requests. If there are outstanding requests, a dedicated polling kernel thread for this queue is woken up.

An important performance optimization to speed up disk encryption is to load native x86 kernel modules with hardware-accelerated optimized crypto routines. As these modules contain assembly code, that is not position-independent, we need support for kernel module loading in LKL to allow relocations at runtime. rkt-io builds the kernel crypto modules from the normal x86 port with a small patch (56 LOCs) to align the kernel module initializer struct between the two architectures. At setup time, it loads the kernel modules via a syscall, and the modules are linked into the rkt-io binary. The binary as a whole is signed and the signatures is checked when setting up the enclave whereas modules are loaded from enclave memory. Enclave pages are mapped as writeable, but the OS cannot modify them due to the SGX integrity checks.

The modules itself rely on x86 CPU feature checks using the `cpuid` instruction to determine available CPU extensions. We modify LKL to load this information during the bootup based on `cpuid` information from outside of the TEE (`cpuid` is an illegal instruction inside SGX enclaves).

Currently each NVMe disk can only be accessed by one enclave at the time. Since rkt-io already makes use of SPDK multi-process features it is possible with little modification to allow multiple processes, each using their own NVMe namespace with their own request/completion queue.

6 Evaluation

Our experimental evaluation is based on four real-world applications (see Figure 7): SQLite, Nginx, Redis and MySQL.

Testbed. We perform our experiments on two machines with SGX support that have an Intel Core i9-9900K CPU with 8 cores (16 hyper-threads), 64 GiB of memory (caches: 32 KiB L1; 256 KiB L2; 16 MiB L3). The I/O devices are an Intel XL710

Ethernet controller for 40 GbE QSFP+ (rev/. 02), and a P4600 NVMe 2 TB drive. The host OS is Linux version 5.7.12.

Baselines. We compare our performance against the overall performance of these applications across three systems: (i) native Linux (unsecured version), (ii) Scone (a host OS-based approach), and (iii) SGX-LKL (a LibOS-based approach). The applications are compiled against musl libc.

For the native benchmarks, we use ext4 as a filesystem using `device mapper` for encryption with the aes-xts-256 cipher. We use the same configuration inside the TEE for SGX-LKL and rkt-io. SGX-LKL accesses the block device as a file through the host interface, while rkt-io accesses it via SPDK. For Scone, we enable fileshield [74] and store the data on ext4 without encryption.

All network benchmarks have TLS enabled (Redis, MySQL and Nginx). Both the native and Scone benchmarks access the network through the host socket interface, while rkt-io uses DPDK. To connect SGX-LKL to the native network adapter, we use a TAP interface that is bridged with the native NIC.

For Scone, we use recommended tuning parameters for the I/O-heavy workloads: two threads running inside the TEE with a system call queue assigned to each thread. Each system call queue has 7 I/O threads running outside the TEE.

6.1 SQLite database

Methodology. We evaluate SQLite [78] using its default configuration with journal mode set to delete and full synchronization. We use the Speedtest benchmark [78] shipped with SQLite to perform 15k transactions. We then configure the benchmark to perform 5k transactions, each with insert, update and delete operations. We report the throughput as transactions per second for each operation. We compare the results across the four system configurations, as shown in Figure 7a.

Results. rkt-io performs 2.0–2.8× better than SGX-LKL and 2.4–3.4× better than Scone. However, the performance of rkt-io is lower than for the native run (outside TEE) by 1.7×, 2.3× and 1.8× for insert, update and delete operations, respectively.

Our profiling of the experiments shows that the writes in a transaction are cheaper compared to creating/opening/flushing/unlinking the journal/WAL files. For such an I/O pattern, rkt-io and SGX-LKL have an advantage over Scone: they can directly access inodes from the LibOS inode cache, as they implement the filesystem themselves, while Scone must perform host system calls.

On the other hand, even though the writes performed by SQLite itself are small (4 KiB) compared to the other operations completed around them, the writes still need to be flushed to the disk to provide crash consistency. This is where the polling-based approach of rkt-io falls behind the native execution, as rkt-io spends more CPU cycles on polling to wait for the I/O completion.

6.2 Nginx web server

Methodology. We evaluate Nginx [58] in a client/server configuration using two machines. We use the `wrk` HTTP benchmarking tool [89] to request a 3 MB file (average page size according to [32]) via HTTPS. The benchmarking tool is setup as a client process running on another machine, for 30 seconds using 16 threads and 100 concurrent connections. We then report the throughput and latency of the server on this workload as requests per second and milliseconds, respectively. We compare the results across the four system configurations, as shown in Figure 7b and Figure 7c.

Results. rkt-io incurs a lower average per request latency than SGX-LKL (2.7×) and Scone (2.3×) as well as a higher throughput than SGX-LKL (3.4×) and Scone (2.3×). There is still a performance gap compared to the native non-secure run, which has 2.5× higher throughput and 2.4× lower latency.

Our profiling of the web server shows that, while serving the requests, Nginx spends 92% of the time in the kernel to process network packets, while the rest of the time is spent mostly in userspace encryption. This benchmark therefore shows the differences in the network stacks: in SGX-LKL, network packets must traverse the network stacks in the host and LKL. The host then has two extra network devices that the network packets must pass (the TAP interface and the bridge), and their respective firewalls.

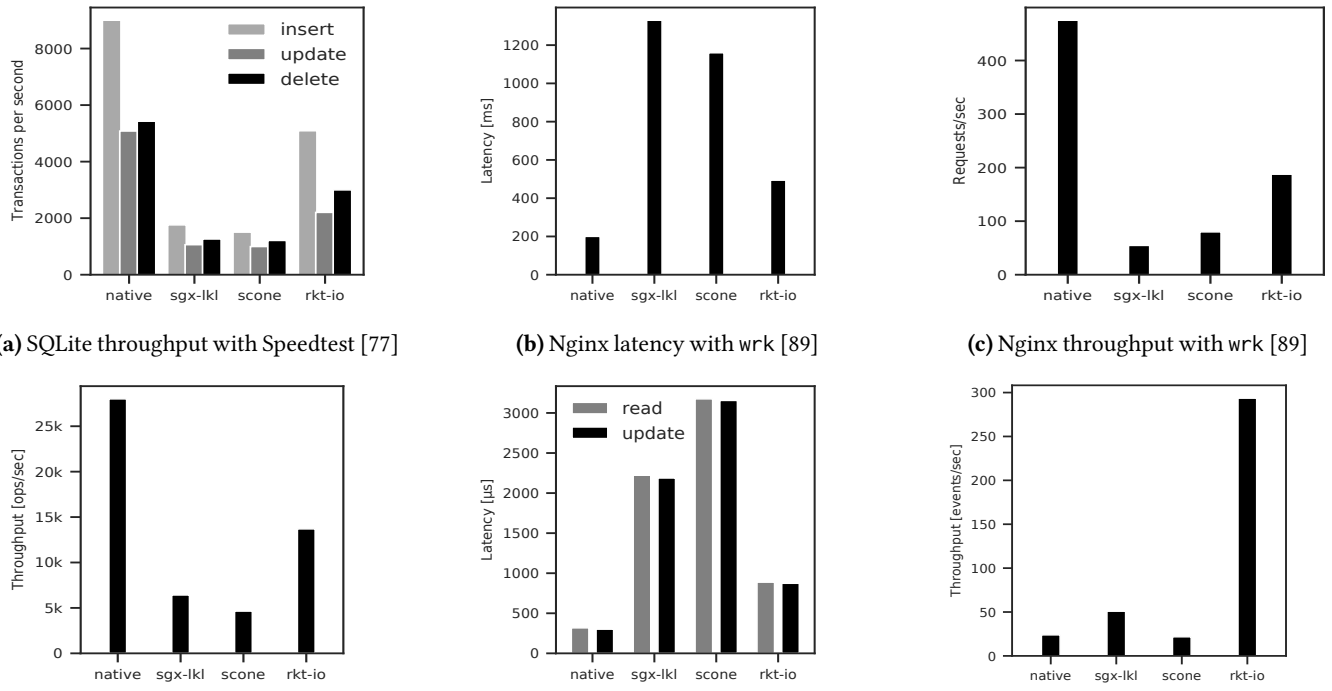
Scone can transfer network packets to the native host interface directly, but it still spends more time copying data from the TEE to its system call queue and from the system call queue to the host than rkt-io, which interacts with the NIC directly. The gap between native and SCONE measured in our paper diverge from the original publication of Scone [5]. In their evaluation they saturated a 10GbE NIC, in our evaluation however we employed 40GbE cards which moves the bottleneck more to the network stack.

Compared with the `iPerf` benchmark we see a bigger gap between rkt-io and native. This is because in the Nginx benchmarks streams are smaller, and therefore, NIC offloadings are not as efficient as the send buffers are shorter.

6.3 Redis key-value store

Methodology. We evaluate Redis [68] with the YCSB benchmark [18], which runs on another client machine. The key-value store is loaded with 100k key-value pairs. We use workload A of YCSB for a total of 10k operations using 16 threads. We then report the throughput and latency for the read/update operations on the key/value store in terms of operations per second and milliseconds, respectively. We compare the results across the four system configurations as shown in Figure 7d and Figure 7e.

Results. rkt-io's throughput is better compared to Scone (2.9×) and SGX-LKL (2.1×); however, it is slower than native execution (2.0×). The average latency per operation follows



(a) SQLite throughput with Speedtest [77]

(b) Nginx latency with `wrk` [89]

(c) Nginx throughput with `wrk` [89]

(d) Redis throughput with YCSB (A) [18]

(e) Redis latency with YCSB (A) [18]

(f) MySQL throughput with sysbench [80]

Figure 7. Performance of four real-world applications (SQlite, Nginx, Redis, and MySQL) on three secure systems (Scone, SGX-LKL and rkt-io) and native Linux (no security)

a similar trend, with the Redis server running on rkt-io with 2.8–2.9× lower latency than native execution. However, rkt-io is faster than Scone (3.6×) and SGX-LKL (2.5×).

Our profiling shows that the workload of this benchmark is also network-bound, similar to Nginx, however, we measure that only 50% of the time is spent in the network stack, and the remaining majority of the time is spent on TLS encryption. This alone would suggest that SGX-based solutions should be closer to native execution performance, because they require less interaction with their I/O stacks; however, this is not the case. We see an increase in the activity of the enclave paging kernel thread (i.e. a 3× increase in CPU usage for rkt-io when compared to the Nginx benchmark with Redis). Paging of enclave pages to unprotected memory is a well-known source of performance overhead in SGX [6] due to encryption and context-switch overhead. Therefore, the runtime difference is caused by increased paging when Redis' in-memory data structures are accessed.

6.4 MySQL database server

Methodology. We evaluate MySQL [56] with the SysBench benchmarking tool [80]. The benchmarking tool is setup on another machine as a client to generate the OLTP workloads. We then compare the throughput of the server serving OLTP requests, in events per second. We compare the results across the four system configurations as shown in Figure 7f.

Top 5	Syscall	Count	Time(μs)	Total (%)
#1	futex	64	4.20e+07	69.4
#2	read	24728	9.40e+06	15.5
#3	select	9	8.99e+06	14.8
#4	fsync	436	6.03e+04	0.1
#5	write	8243	3.48e+04	0.06

Table 1. Top-5 syscalls in MySQL native execution

Results. rkt-io's throughput is better than native, SGX-LKL and Scone by 12.2×, 5.7× and 13.5× respectively.

Our off-cpu analysis [60] shows that MySQL spends a significant time performing table locks with `futexes`. To illustrate our analysis, we show the time spent by the native MySQL execution for the top-5 syscalls in Table 1. Since both SGX-LKL and rkt-io do scheduling in the userspace, they are faster in handling these locks as they can switch without a CPU context switch to a different thread. Therefore, rkt-io performs better than native execution. (Note that Scone also relies on the host OS for handling I/O operations, but the implementation details of `futexes` in Scone are not available).

Furthermore, Table 1 shows that the top-2 and top-3 most used system calls are network-related followed by filesystem syncs. Since rkt-io network/disk throughput is higher than SGX-LKLs as shown in `iPerf` (Figure 1c) and `fio` benchmarks (Figure 1b), rkt-io processes more number of transactions compared to SGX-LKL.

7 Related work

I/O support for shielded execution. With the adoption of TEEs in cloud environments, shielded execution frameworks, such as Haven [7], Scone [5], Graphene-SGX [85], Panoply [75], and SGX-LKL [66], are used to deploy applications with strong security properties. These frameworks provide OS functionality and associated run-time libraries to support unmodified legacy applications in TEEs. They promote portability, programmability and performance for shielded execution, and have been used to implement a wide-range of secure systems for storage [6, 45], data analytics [73, 94], data management [67], distributed systems [82], FaaS [84], file storage [2], network functions [64, 83], decentralized ledgers [51], content delivery networks [30], machine learning [46], etc.

Current shielded execution frameworks primarily rely on existing OS functionality (i.e., syscalls to host OS or a LibOS inside the TEE) for I/O operations, which differs from rkt-io's design of providing a separate direct I/O stack within the TEE for storage and networking. Scone [5], SGX-LKL [66], and Eleos [62] use switchless asynchronous I/O calls to mitigate I/O bottlenecks in the TEEs. This avoids expensive TEE world switches, and the use of I/O threads outside the TEEs improves I/O performance through asynchronous syscalls [76]. Since these approaches rely on the host OS to handle I/O operations via dedicated I/O threads outside the TEE, it suffers from performance and security limitations. In terms of performance, it reduces the number of available threads for application execution, requires extra copies of the data and syscall arguments, and significantly increases I/O latency; since the host OS is responsible for performing I/O operations, it is also susceptible to Iago [15] and host interface attacks [81].

To overcome the limitations of switchless asynchronous I/O mechanisms, ShieldBox [83] uses Intel DPDK [24] as a user mode driver to support secure middleboxes based on the Click modular router. Likewise in the storage domain, Speicher [6] accesses persistent storage (SSDs) through Intel SPDK [37] within the TEE to provide a secure persistent KV store. Since these systems try to address I/O bottlenecks, the corresponding I/O stacks are designed to operate at the lowest layer, and thus are incompatible with legacy applications that require POSIX network and file support. More specifically, Shieldbox only targets Layer-2 networking without TCP/IP support; in contrast, rkt-io provides secure network (IP) and transport protocols. Similarly, Speicher operates at the block layer without filesystem support, but rkt-io's I/O stack supports off-the-shelf filesystems (e.g., ext4, xfs, etc.) available in the Linux kernel. Finally, rkt-io adopts a holistic design to provide both network and storage support in an integrated I/O stack.

High-performance I/O stacks. To meet the performance needs of I/O-intensive applications and leverage high-performance hardware, a range of I/O stacks have been proposed for networking (e.g., mTCP [41], netmap [70], StackMap [91], Sandstorm [52], and TAS [43]) as well as storage (e.g., Decibel [57], i10 [33], DiskMap [53], ReFlex [44], and PASTE [31]).

Our work builds on the designs of high-performance I/O stacks, especially I/O stacks bypassing the OS kernel. rkt-io differentiates compared to these I/O stacks in two aspects: (a) it supports secure I/O operations directly within TEEs, whereas these I/O stacks would require non-trivial changes for retro-fitting their architecture in the context of TEEs; and secondly, (b) it supports full Linux compatibility, whereas these I/O stacks exports new APIs for the network/storage stacks, which might require significant re-writing of existing applications to adopt the new APIs.

Efficient LibOS design. LibOSs can improve application performance, while ensuring portability [9, 16, 26, 65]. Advances in high-performance networking and storage in data centers has led to a resurgence of LibOSs support for latency-sensitive applications: Arrakis [63], Demikernel [93] and IX [8] adapt a kernel-bypass design that splits functionality across control and data paths in order to support I/O-intensive applications that use high-performance NICs and SSDs. In the same spirit, rkt-io favors an host-independent I/O interface based on LKL to avoid the OS on the critical I/O path for improved performance (and also for security). In contrast to these systems, we need to address additional fundamental challenges to make the direct I/O compatible in the context of TEEs. Since the DMA region cannot be directly mapped in the TEEs, our design requires additional "one-copy" instead of "zero-copy" to read/write data in the TEE. On the down side, these systems do not support POSIX compliant APIs.

8 Conclusion

In this paper, we presented the design and implementation of rkt-io, a direct I/O stack for shielded execution targeting high-performance networking and storage. Our I/O stack strives to overcome the performance and security limitations of switchless asynchronous I/O designs adopted in the host OS- and LibOS-based shielded execution frameworks. This design goal is achieved by our sensible co-design of the kernel-bypass direct I/O libraries with the LibOS (LKL) running in the trusted domain of TEEs. Thereby, our I/O stack facilitates switchless direct I/O with improved performance and security, while preserving the rich POSIX environment to support off-the-shelf filesystems and network stacks. We have implemented rkt-io as an integrated I/O stack, and extensively evaluated it using a wide-range of micro-benchmarks and unmodified real-world applications. Our evaluation shows the effectiveness of the individual system design components and overall approach; for instance, our network and storage stacks are 7–9× faster compared to Scone (host-based) and SGX-LKL (LibOS-based) based on `iPerf` and `fio` benchmarks, respectively.

Source code availability. Our project is publicly available for the research community—see Artifact appendix A.

Acknowledgements. We thank our shepherd, Hakim Weatherspoon, and the anonymous reviewers for their helpful comments. We are also thankful to Maurice Bailleu, Le Quoc Do, and Dimitra Giantsidi for their help during the project. This work was supported in parts by a UK RISE Grant from NCSC/GCHQ at the University of Edinburgh, UK.

References

[1] Nix package manager. https://nixos.org/download.html. Last accessed: March, 2021.

[2] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee. OBLIVIATE: A data oblivious filesystem for intel SGX. In *25th Annual Network and Distributed System Security Symposium (NDSS)*, 2018.

[3] AMD. AMD Secure Encrypted Virtualization (SEV). https://developer.amd.com/sev/. Last accessed: Mar, 2021.

[4] ARM. Building a secure system using trustzone technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf. Last accessed: Mar, 2021.

[5] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[6] M. Bailleu, J. Thalheim, P. Bhatotia, C. Fetzer, M. Honda, and K. Vaswani. SPEICHER: Securing lsm-based key-value stores using shielded execution. In *17th USENIX Conference on File and Storage Technologies (FAST)*, 2019.

[7] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[8] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[9] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

[10] A. Biondo, M. Conti, L. Davi, T. Frassetto, and A.-R. Sadeghi. The guard's dilemma: Efficient code-reuse attacks against intel SGX. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[11] BlobFS: Blobstore Filesystem. Last accessed: Mar, 2021.

[12] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.

[13] Bufferbloat project. Last accessed: Mar, 2021.

[14] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[15] S. Checkoway and H. Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[16] D. R. Cheriton and K. J. Duda. A caching model of operating system kernel functionality. In *Proceedings of the 6th Workshop on ACM SIGOPS European Workshop*, 1994.

[17] A. Cloud. Alibaba Cloud's Next-Generation Security Makes Gartner's Report. https://www.alibabacloud.com/blog/alibaba-clouds-next-generation-security-makes-gartners-report_595367. Last accessed: Mar, 2021.

[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud computing (SoCC)*, 2010.

[19] J. Corbet. The multiqueue block layer. https://lwn.net/Articles/552904/, 2013. Last accessed: Mar, 2021.

[20] V. Costan and S. Devadas. Intel SGX Explained, 2016.

[21] R. Darawsheh. mbuf external buffer and usage examples. In *Proceedings of the DPDK Userspace, Dublin*, 2018.

[22] dm-crypt/device encryption. Last accessed: Mar, 2021.

[23] J. A. Donenfeld. WireGuard: Next Generation Kernel Network Tunnel. https://www.wireguard.com/papers/wireguard.pdf. Last accessed: Mar, 2021.

[24] Data plane development kit (DPDK). Last accessed: Mar, 2021.

[25] E. Dumazet. Busy Polling: Past, Present, Future. In *netdev 2.1 Montreal*, 2017.

[26] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

[27] Introducing Google Cloud Confidential Computing with Confidential VMs. https://cloud.google.com/blog/products/identity-security/introducing-google-cloud-confidential-computing-with-confidential-vms. Last accessed: Mar, 2021.

[28] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security*, 2017.

[29] M. Hähnel, W. Cui, and M. Peinado. High-resolution side channels for untrusted operating systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017.

[30] S. Herwig, C. Garman, and D. Levin. Achieving keyless cdns with conclaves. In *29th USENIX Security Symposium (USENIX Security)*, 2020.

[31] M. Honda, G. Lettieri, L. Eggert, and D. Santry. PASTE: A network programming interface for non-volatile main memory. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[32] Tracking page weight over time. https://discuss.httparchive.org/t/tracking-page-weight-over-time/1049. Last accessed: Mar, 2021.

[33] J. Hwang, Q. Cai, A. Tang, and R. Agarwal. TCP = RDMA: Cpu-efficient remote storage access with i10. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[34] Product page of Intel SSD DC P4600 Series. https://ark.intel.com/content/www/us/en/ark/products/series/96947/intel-ssd-dc-p4600-series.html. Last accessed: Mar, 2021.

[35] Intel Software Guard Extensions (Intel SGX). https://software.intel.com/en-us/sgx. Last accessed: Mar, 2021.

[36] Intel SGX SDK. Last accessed: Mar, 2021.

[37] Intel Storage Performance Development Kit. http://www.spdk.io. Last accessed: Mar, 2021.

[38] Product page of XL710 network card family. https://www.intel.com/content/www/us/en/products/docs/network-io/ethernet/network-adapters/ethernet-xl710-brief.html. Last accessed: Mar, 2021.

[39] iPerf - The ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr/. Last accessed: Mar, 2021.

[40] Jens Axboe. Flexible I/O Tester. https://github.com/axboe/fio. Last accessed: Dec, 2020.

[41] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. MTCP: A Highly Scalable User-Level TCP Stack for Multicore Systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementationi (NSDI)*, 2014.

[42] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture(ISCA)*, 2015.

[43] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. Anderson. Tas: Tcp acceleration as an os service. In *Proceedings of the Fourteenth EuroSys Conference (EuroSys)*, 2019.

[44] A. Klimovic, H. Litz, and C. Kozyrakis. Reflex: Remote flash ≡ local flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[45] R. Krahn, B. Trach, A. Vahldiek-Oberwagner, T. Knauth, P. Bhatotia, and C. Fetzer. Pesos: Policy enhanced secure object store. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys)*, 2018.

[46] R. Kunkel, D. L. Quoc, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer. Tensorscone: A secure tensorflow framework using intel SGX. *CoRR*, 2019.

[47] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer. SGXBOUNDS: Memory Safety for Shielded Execution. In *Proceedings of the 12th ACM European Conference on Computer Systems (EuroSys)*, 2017.

[48] A. Langley. rfc7539: ChaCha20 and Poly1305 for IETF Protocols. https://tools.ietf.org/html/rfc7539. Last accessed: Mar, 2021.

[49] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song. Keystone: an open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*, 2020.

[50] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[51] J. Lind, O. Naor, I. Eyal, F. Kelbert, E. G. Sirer, and P. Pietzuch. Teechain: A secure payment network with asynchronous blockchain access. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[52] I. Marinos, R. N. Watson, and M. Handley. Network stack specialization for performance. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, 2014.

[53] I. Marinos, R. N. Watson, M. Handley, and R. R. Stewart. Disk|crypt|net: Rethinking the stack for high-performance video streaming. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2017.

[54] Microsoft Azure. Azure confidential computing. https://azure.microsoft.com/en-us/solutions/confidential-compute/. Last accessed: Mar, 2021.

[55] musl: an implementation of the C standard library. https://musl.libc.org/. Last accessed: Mar, 2021.

[56] MySQL. https://www.mysql.com/. Last accessed: Mar, 2021.

[57] M. Nanavati, J. Wires, and A. Warfield. Decibel: Isolation and sharing in disaggregated rack-scale storage. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[58] Nginx Web Server. https://www.nginx.com/. Last accessed: Mar, 2021.

[59] O. Purdila and L. A. Grijincu and N. Tapus. Lkl: The linux kernel library. In *9th RoEduNet IEEE International Conference*, 2010.

[60] Off-CPU Flame Graphs. Last accessed: Mar, 2021.

[61] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2018.

[62] M. Orenbach, M. Minkin, P. Lifshits, and M. Silberstein. Eleos: ExitLess OS services for SGX enclaves. In *Proceedings of the 12th ACM European ACM Conference in Computer Systems (EuroSys)*, 2017.

[63] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[64] R. Poddar, C. Lan, R. A. Popa, and S. Ratnasamy. Safebricks: Shielding network functions in the cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[65] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the library os from the top down. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[66] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. Pietzuch. Sgx-lkl: Securing the host os interface for trusted execution, 2019.

[67] C. Priebe, K. Vaswani, and M. Costa. EnclaveDB: A Secure Database using SGX (S&P). In *IEEE Symposium on Security and Privacy*, 2018.

[68] Redis. https://redis.io/. Last accessed: Mar, 2021.

[69] RISC-V. Keystone Open-source Secure Hardware Enclave. https://keystone-enclave.org/. Last accessed: Mar, 2021.

[70] L. Rizzo. Revisiting network I/O APIs: The Netmap Framework. *Communications of the ACM*, 2012.

[71] N. Santos, K. P. Gummadi, and R. Rodrigues. Towards Trusted Cloud Computing. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[72] N. Santos, R. Rodrigues, and B. Ford. Enhancing the os against security threats in system administration. In *Proceedings of the 13th International Middleware Conference (Middleware)*, 2012.

[73] F. Schuster, M. Costa, C. Gkantsidis, M. Peinado, G. Mainar-ruiz, and M. Russinovich. VC3 : Trustworthy Data Analytics in the Cloud using SGX. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[74] Scone file protection. https://sconedocs.github.io/SCONE_Fileshield/. Last accessed: Mar, 2021.

[75] S. Shinde, D. Le Tien, S. Tople, and P. Saxena. PANOPLY: Low-TCB Linux Applications with SGX Enclaves. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[76] L. Soares and M. Stumm. FlexSC: Flexible System Call Scheduling with Exception-less System Calls. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[77] speedtest. https://www.sqlite.org/speed.html. Last accessed: Mar, 2021.

[78] SQLite. https://www.sqlite.org/. Last accessed: Mar, 2021.

[79] Supported Hardware. Last accessed: Feb, 2021.

[80] sysbench. https://github.com/akopytov/sysbench. Last accessed: Mar, 2021.

[81] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[82] B. Trach, R. Faqeh, O. Oleksenko, W. Ozga, P. Bhatotia, and C. Fetzer. T-lease: A trusted lease primitive for distributed systems. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC)*, 2020.

[83] B. Trach, A. Krohmer, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer. ShieldBox: Secure Middleboxes using Shielded Execution. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, 2018.

[84] B. Trach, O. Oleksenko, F. Gregor, P. Bhatotia, and C. Fetzer. Clemmys: Towards secure remote execution in faas. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR)*, 2019.

[85] C.-C. Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2017.

[86] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[87] Vlad Krasnov. How "expensive" is crypto anyway. https://blog.cloudflare.com/how-expensive-is-crypto-anyway, 2017. Last accessed: Mar, 2021.

[88] Weichbrodt, Nico and Kurmus, Anil and Pietzuch, Peter and Kapitza, Rüdiger. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In *Computer Security – ESORICS*, 2016.

[89] wrk. https://github.com/wg/wrk. Last accessed: Mar, 2021.

[90] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[91] 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)*, 2016.

[92] T. Yates. Linux kernel: Introduction of hybrid polling in the blk-mq subsystem. https://lwn.net/Articles/735275, 2017. Last accessed: Mar, 2021.

[93] I. Zhang, J. Liu, A. Austin, M. L. Roberts, and A. Badam. I'm not dead yet! the role of the operating system in a kernel-bypass era. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2019.

[94] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

A Artifact appendix

A.1 Abstract

This artifact contains the LibOS and scripts to reproduce the experiments and figures from the Eurosys 2021 paper—"rkt-io: A Direct I/O Stack for Shielded Execution" by J. Thalheim, H. Unnibhavi, C. Priebe, P. Bhatotia, P. Pietzuch. rkt-io provides a direct userspace I/O stack for network and storage, specifically designed for TEEs that combines high-performance, POSIX/Linux ABI compatibility and security.

A.2 Artifact repository

All the project source code including the instructions on how to evaluate and build the software is available in the following git repository: https://github.com/Mic92/rkt-io

A.3 Hardware requirements

We require the following hardware setup to reproduce our experimental results.

- Intel NIC supported by i40e driver: In rkt-io, we perform some driver optimizations that require some refactoring in DPDK to reduce the memory copy. In particular, we implement an allocation function that allocates kernel socket buffer instead of DPDK's mbuf. Hence, we modify the low-level i40e Intel NIC driver to support this new allocation scheme. However, we did not apply these changes to other drivers. Consequently, one needs the same hardware to reproduce the paper results. Our NIC is XL710 [38].
- NVMe block device: We need a free NVMe block device. We use an Intel DC P4600 2TB NVMe drive [34]. Note that this device will be *reformatted* during the evaluation.
- Intel CPU with SGX support: Most new consumer CPUs have SGX support. However, some server processors of the Xeon family do not support SGX.
- A second machine acting as a client for generating the workloads. The client machine needs a similar capable NIC (i.e., the same bandwidth). The client machine does not require to have an NVMe drive.

A.4 Software requirements

We require the following software configuration to reproduce our experimental results.

- Operating system: Linux
- Nix [1]: For reproducibility, we use the Nix package manager to download all the build dependencies. We have fixed the package versions to ensure reproducible evaluation.
- Python 3.7 or newer for the script to reproduce the evaluation.

A.5 Applications

In our evaluation, we run the following applications and benchmarks.

- Sqlite [78] with its Speedtest benchmark [77].
- Nginx [58] using the wrk HTTP benchmark [89].
- Redis [68] with the YCSB benchmark [18].
- MySQL [56] with the SysBench benchmarking tool [80].

In addition, we use `iPerf` [39] and `fio` [40] for the micro-benchmarks.

A.6 Methodology

In our artifact evaluation, we reproduce the following results from the paper:

- Figure 1: Micro-benchmarks to showcase the performance of syscalls, storage, and network stacks across different systems:
  1. System call latency with `sendto()`
  2. Storage stack performance with `fio`
  3. Network stack performance with iPerf
- Figure 5: Micro-benchmarks to showcase the effectiveness of various design choices in rkt-io:
  1. Effectiveness of the SMP design w/ `fio` with increasing number of threads
  2. `iPerf` throughput w/ different optimizations
  3. Effectiveness of hardware-accelerated crypto routines
- Figure 7: Performance comparison of four real-world applications (SQlite, Nginx, Redis, and MySQL) for four configurations: native Linux (no security), and three secure systems: Scone, SGX-LKL and rkt-io.
  1. SQLite throughput w/ Speedtest
  2. Nginx latency w/ wrk
  3. Nginx throughput w/ wrk
  4. Redis throughput w/ YCSB (A)
  5. Redis latency w/ YCSB (A)
  6. MySQL OLTP throughput w/ sys-bench

For Figure 1 and Figure 7 we compare our project against SGX frameworks SGX-LKL [66] and Scone [5].

Evaluation workflow. The evaluation script `reproduce.py` first builds rkt-io, SGX-LKL and Scone, and thereafter, it runs all the experiments. This script only depends on Python and Nix as referenced in the software requirements. All other dependencies will be loaded through Nix. If the script fails at any point it can be restarted—after the restart, it will continue with the incomplete builds or experiments. Each command it runs will be printed during the evaluation along with environment variable set. In addition to some default settings, the evaluation also requires machine-specific settings. The script read these settings from a file containing the hostname of the machine + '.env'. An example of the configuration file is included in the repository.

The evaluation script is executed as follows:

```
$ python reproduce.py
```

After the build is finished, it will start the evaluation and generate all plots. The plots are stored in `./results`.