RAKIS: Secure Fast I/O Primitives Across Trust Boundaries on Intel SGX

Mansour Alharthi
Georgia Institute of Technology
Atlanta, United States

Fan Sang
Georgia Institute of Technology
Atlanta, United States

Dmitrii Kuvaiskii
Intel Corporation
Hillsboro, United States

Mona Vij
Intel Corporation
Hillsboro, United States

Taesoo Kim
Georgia Institute of Technology
Atlanta, United States

Abstract

The use of Intel® Software Guard Extensions (SGX) offers robust security measures for shielding applications in untrusted environments. However, the performance overhead experienced by IO-intensive applications within SGX limits widespread adoption. Prior approaches have proposed the use of userspace kernel-bypass libraries such as Data Plane Development Kit (DPDK) inside SGX enclaves to enable direct access to IO devices. However, these solutions often come at the cost of increasing the Trusted Computing Base (TCB) size, expanding the attack surface, and complicating deployment. In this paper, we introduce RAKIS, a comprehensive system that securely enables SGX enclave programs to leverage fast IO Linux kernel primitives without modifying user applications. RAKIS prioritizes the security of its TCB components by employing rigorous software testing and verification methods, embodying a security-by-design approach. Importantly, RAKIS achieves performance advantages without sacrificing TCB size or introducing deployment intricacies and demonstrates significant improvements in benchmark tests with a 4.6x increase in UDP network throughput compared to state-of-the-art SGX enclave LibOS (Gramine-SGX). To demonstrate the practical applicability of RAKIS, we evaluate its performance on four real-world programs showcasing an average performance improvement of 2.8x compared to Gramine-SGX across all workloads.

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

Keywords: Secure enclaves, express data path, iouring, trusted execution environments, system security, Intel SGX

EuroSys '25, March 30-April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1196-1/25/03
https://doi.org/10.1145/3689031.3696090

ACM Reference Format:
Mansour Alharthi, Fan Sang, Dmitrii Kuvaiskii, Mona Vij, and Taesoo Kim. 2025. RAKIS: Secure Fast I/O Primitives Across Trust Boundaries on Intel SGX. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3689031.3696090

1 Introduction

The evolution of modern computing has introduced new paradigms and challenges. A salient development is the increased adoption of Trusted Execution Environments (TEEs), led by Intel Software Guard Extensions (SGX). SGX enables user-level code to allocate protected memory regions, or enclaves, safeguarding them from even privileged adversaries, including the Operating System (OS) and hypervisor. While groundbreaking, SGX grapples with issues particularly regarding untrusted and slow IO operations. The dependence on an untrusted kernel for crucial IO functionalities, such as networking and filesystem operations, leads to a surge in costly SGX enclave exits. These exits, induced by syscalls and context switches, significantly impair the performance of IO-intensive SGX applications, causing up to a 5x decrease in performance [42]. Given the substantial cost of syscalls for SGX enclave programs, the elimination of syscalls for enclave programs should be prioritized.

Boosting application performance by bypassing kernel mediation (e.g., syscalls) has been a compelling academic focus for over a decade. Dune [4] serves as an exemplary initiative in this realm, functioning as a type-2 hypervisor to shift applications into a privileged mode. This facilitates the direct execution of sensitive operations, eliminating the need for expensive syscalls. Arrakis [32] further advances this space by reducing the OS's role to serving only control-plane functionalities and harnessing the power of the Input-Output Memory Management Unit (IOMMU) to grant user-space applications direct and secure access to IO devices. This strategy significantly enhances latency and throughput for data-intensive, IO-demanding applications. More recently, the Linux kernel has embraced this trend by introducing new IO interfaces that offer reduced reliance on syscalls, such as eXpress Data Path

(XDP) [19] and `io_uring` [16]. These Fast IO Kernel Primitives (FIOKPs) have emerged as specialized interfaces that enhance performance by eliminating the overhead associated with syscalls and context switches.

The challenge, therefore, lies in the seamless integration of FIOKPs within SGX enclaves, thereby enhancing performance without compromising the robust security guarantees intrinsic to SGX. More specifically, securely enabling FIOKPs in SGX brings a unique set of challenges: 1) *Untrusted kernel features:* FIOKPs are often accompanied with userspace libraries (e.g., `libxdp` [43] for XDP and `liburing` [25] for `io_uring`) that assume a trusted OS while this assumption does not extend to enclave programs; 2) *Incompatible IO interfaces:* The absence of a one-to-one mapping between regular IO syscalls such as `send()/recv()` and FIOKP interfaces necessitates modifications to enclave programs, thereby undermining existing efforts [1, 3, 6, 34, 36] to support executing unmodified programs inside SGX; 3) *Usability vs. enlarged TCB:* FIOKPs alone provide only lower-level services (e.g., layer-2 network packets in XDP) and require additional functionalities to support a full stack of services (e.g., ARP, IP/UDP), which expands the TCB and creates a larger attack surface.

RAKIS. In this paper, we present RAKIS, a comprehensive end-to-end system for securely enabling FIOKPs within SGX enclaves, seamlessly integrating them with unmodified user applications. RAKIS takes a security-first approach by performing rigorous security checks on the shared untrusted data of FIOKPs. To mitigate potential threats posed by a malicious OS tampering with data passed into the enclave, RAKIS employs rigorous software testing and verification methods in all code related to FIOKPs within the SGX environment to ensure their correctness even when operating on untrusted data. Furthermore, RAKIS enables high-level functionalities (e.g., layer-3 networking) on top of FIOKPs inside enclaves. In order to support unmodified user applications, RAKIS preserves regular syscall interfaces and seamlessly integrates into existing enclave syscall abstraction layers. Additionally, RAKIS maintains a small footprint, thereby ensuring a minimum increase in TCB size.

To demonstrate the real-world capability and applicability of RAKIS for enclave applications, we have enabled it on two prominent FIOKPs: XDP for providing performant UDP networking and `io_uring` for providing exit-less TCP networking and file IO syscalls. The process of implementing these two FIOKPs in RAKIS is described in detail, including a security analysis of RAKIS-enabled XDP and `io_uring`. Further highlighting the benefits of integrating RAKIS with enclave programs, we have conducted extensive benchmark performance evaluations revealing an average UDP networking performance gain of 4.6x compared to a state-of-the-art SGX library OS. We have also evaluated the completeness of RAKIS's support for different syscalls across a selection of 4 real-world workloads: Memcached [29], Redis [35], Curl [8] and MCrypt [28] and showed that RAKIS can support execution of practical workloads with 2.8x average performance improvement compared to a state-of-the-art SGX library OS.

Contributions. This paper makes the following contributions:

- **FIOKPs for SGX.** RAKIS represents the first comprehensive end-to-end system designed to securely enable Linux kernel fast IO primitives for unmodified SGX applications, offering a unique blend of performance optimization, ease-of-deployment and minimal TCB.
- **Security-by-design.** RAKIS employs comprehensive testing mechanisms, encompassing both model checking and dynamic fuzzing techniques, to ensure robustness across RAKIS's trusted components.
- **Implementation.** RAKIS has been integrated with two FIOKPs: XDP and `io_uring`, demonstrating its applicability and potential to enhance the performance of real-world, IO-intensive applications within SGX enclaves.

2 Background

2.1 Intel SGX

SGX [15] is an instruction set that enables the creation of a secure enclave, providing a protected execution environment for user-level software. The enclave memory is encrypted to ensure confidentiality and integrity of the data stored within it. After the enclave is initialized, a user program can enter the enclave using the `EENTER` instruction, allowing a context switch from untrusted code to trusted code. Access to hardware and other OS resources is restricted within the enclave. When the enclave program needs to make syscalls, it must copy the syscall data to untrusted memory, exit the enclave using the `EEXIT` instruction, perform the syscall, re-enter the enclave, and copy the result back from untrusted memory. These operations impose a significant overhead, with the enclave exit alone requiring a minimum of 8200 CPU cycles, as reported by Weisse et al. [42].

2.2 Running Unmodified Applications in SGX

Running unmodified user applications inside SGX enclaves has been a topic of extensive academic research since the introduction of SGX. One of the primary challenges in this context is to provide secure access to host OS syscalls from within the enclave. To address this challenge, various works proposed the use of a Library Operating System (LibOS) inside enclaves [1, 3, 6, 34, 36]. Figure 1 provides the generic architecture of LibOSs proposed by previous works. In essence, the LibOS would serve as an intermediary layer that abstract and handle syscalls within the enclave environment. Depending on the specific scenario, the LibOS employs different strategies to handle syscalls. In some cases, it can completely emulate the syscall within the enclave itself. However, for

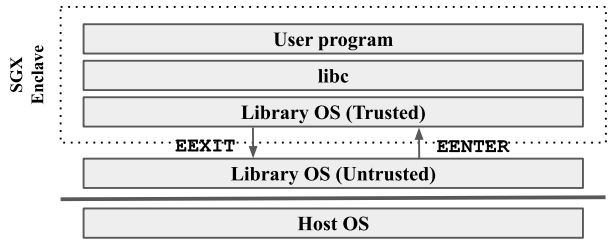


Figure 1. An example of a SGX LibOS architecture. The LibOS acts as an intermediary to provide unmodified user programs with access to host OS services from within the enclave.

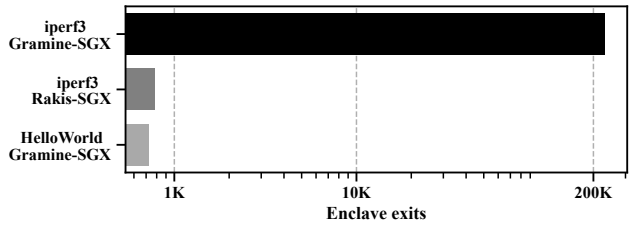


Figure 2. Comparison of enclave exits required when running the iperf3[17] tool on Gramine LibOS [6] and RAKIS within SGX enclaves. The x-axis represents the log-scaled count of enclave exits at the conclusion of the network test, with the baseline depicted by the HelloWorld program. RAKIS utilizes FIOKPs to eliminate the need for enclave exits for IO operations.

majority of syscalls, particularly those related to IO operations, the LibOS would exit the enclave, perform the required syscall on behalf of the user program in the untrusted host environment, and then re-enter the enclave to verify the results and provide them back to the user program running inside the enclave. While this mechanism permits unmodified enclave user programs to leverage host OS syscalls, it imposes costs associated with enclave exit operations. To demonstrate its frequency, Figure 2 illustrates the count of enclave exits required to execute the network benchmark test presented in our evaluation, using Gramine LibOS [6] within SGX enclaves.

2.3 eXpress Data Path

XDP is a Linux kernel primitive for high performance network packets processing. Essentially, XDP is an eBPF framework that enables running eBPF scripts on incoming network packets. The user-provided XDP programs attach to network interfaces, with the ability to inspect and perform arbitrary alterations to the received network packets. The return code of the XDP program determines how the received packet is handled, allowing for decisions such as dropping the packet, passing it up the normal Linux kernel network stack, or redirecting it to userspace bypassing the Linux kernel network stack. XDP is the cornerstone feature that enabled introducing a new socket type called XDP Socket (XSK). XSKs are specifically designed for a one-to-one binding with a single Network Interface Card (NIC) queue. The loaded XDP eBPF program orchestrates the redirection of incoming packets from NIC queues to their respective bound XSKs, bypassing the kernel stack. Therefore, XSKs deliver layer-2 network data frames before any kernel processing with up to 5x the throughput of raw networks packets `AF_PACKET` [18].

XSK's data structures. XSKs necessitate the allocation of a packets buffer area known as UMem. The UMem area is divided into frames, where each frame can hold a single network packet. Furthermore, XSKs necessitate a minimum of four producer/consumer rings that enable the transfer of ownership of UMem frames between the user and the kernel in a lockless manner. These rings consist of the Fill Ring (xFill) and Receive Ring (xRX) rings for the receive routine, as well as the Complete Ring (xCompl) and Transmit Ring (xTX) rings for the send routine. The entries of all four rings are UMem offsets pointing to the UMem frame being sent or received.

XSK's setup. The end result of an XSK setup is for packets arriving at a specific NIC queue to be placed in a user allocated UMem buffer. The UMem area is allocated by the user and is passed to the kernel using the `setsockopt` syscall. To create the four rings, the user issues the `setsockopt` syscall for each ring, specifying the respective flag option. For network packets to start flowing into the UMem area, the user must attach an XDP eBPF program to a NIC queue to redirect network packets toward the newly created XSK. The loaded XDP program has flexibility in determining which packets are directed to the XSK based on various factors such as header values of the network packet.

XSK's operation. The operation of XSK sockets revolves around the utilization of four XSK rings for the purpose of sending and receiving network packets within UMem frames. The user is responsible for allocating UMem frames between the send and receive routines, with all UMem frames initially owned by the user. For receiving, two rings are used: xFill and xRX. In xFill, the user acts as the producer, and the kernel is the consumer. Conversely, in the xRX ring, the roles are reversed, with the kernel acting as the producer and the user as the consumer. To receive network packets, the user allocates UMem frames from the UMem area and produces them in the xFill. As network packets arrive, the kernel consumes UMem frames from xFill, places incoming packets into the consumed UMem frames, and then produces the UMem frames in xRX for the user's consumption. For the send routine, xTX and xCompl are used. The user is the producer in the xTX and the consumer in the xCompl, with the kernel assuming the opposite role in both rings. To send network packets, the user allocates a UMem frame, copies the layer-2 data frames into it, and then produces these UMem frames in xTX. The kernel consumes the UMem frames from xTX and performs the transmission of the network packets. Upon completion of the transmission, the kernel produces the UMem frames in the xCompl. This allows the user, acting as the consumer in xCompl, to claim the ownership of UMem frames to use

Ring	Purpose
xFill †	Supply kernel with UMem frames for incoming packets
xRX ‡	Receive populated UMem frames from kernel
xTX †	Request kernel to transmit UMem frames
xCompl ‡	Pass UMem frames to user after transmit is complete
iSub †	Submit asynchronous IO requests to the kernel
iCompl ‡	Provide status information for I/O operations

Table 1. A Summary of the rings data structures of two kernel IO primitives: XSK (top four rings) and `io_uring` (bottom two rings). Rings marked with † are rings where the user is producer, while rings marked with ‡ are rings where the user is consumer. For all rings, the kernel assumes the opposite role.

them for the next send/receive. A summary of the four XSK rings along with each ring's purpose is provided in Table 1.

2.4 `io_uring`

`io_uring` [16] is a versatile Linux kernel primitive that offers an asynchronous interface for performing various types of IO operations. It can be utilized for a wide range of syscalls such as `read`, `send` and `poll` for asynchronous IO operations.

`io_uring`'s data structures. Two shared producer/consumer rings are at the core of `io_uring`'s functionality, which establish the communication channels between the user and the kernel. These rings consist of the Submission Ring (iSub) and the Complete Ring (iCompl), as shown in Table 1. Each iSub entry is an instance of a structure specifying the syscall and its arguments, while an iCompl entry is an instance of a structure mainly containing the return code.

`io_uring`'s operation. The operation of `io_uring` is centered around the use of the iSub and iCompl to request IO operations from the kernel. To initiate an IO operation, the user populates the necessary details within an iSub entry. If the IO operation requires a user buffer, such as in the case of `read` or `write`, the user must provide a pointer to a buffer as part of the iSub entry. Once the submission queue entry is prepared, the user produces a reference to that entry in iSub for processing by the kernel. The kernel, assuming the consumer role in iSub, processes the user request. Upon completion of the requested IO operation, the kernel generates an entry in iCompl containing the operation return code.

3 Overview

Threat Model. The main objective of RAKIS is to provide fast IO primitives within SGX enclaves, ensuring adherence to SGX security design and threat model, with minimal TCB expansion. RAKIS does not trust any data outside enclave memory, this includes any data used to control the operation of FIOKPs. RAKIS achieves equivalent security guarantees as previous works without relying on enclave exits to handle syscalls. RAKIS considers robustness and resilience as part of its threat model, aiming to prevent crashes caused by consumption of invalid data from untrusted memory.

Goals. Design goals of RAKIS are threefolds:

1) Trustworthiness. RAKIS must rigorously handle untrusted values to thwart malicious control flow manipulations within the enclave. Furthermore, RAKIS assumes full responsibility for managing untrusted memory, ensuring that user programs primarily operate on trusted enclave memory. Furthermore, RAKIS must be subjected to thorough program testing and auditing, especially for code components interacting directly with untrusted host OS through shared untrusted memory.

2) Minimal TCB. Vanilla FIOKPs only offer low-level services (e.g., delivering layer-2 network packets) that cannot meet the requirement of modern programs (e.g., layer-4 networking). However, comprehensively supporting lacked kernel functionalities for FIOKPs inside enclaves forms a wide and complex attack surface. Therefore, RAKIS must achieve its goals with minimal TCB increase.

3) Usability. It is crucial to make FIOKP interfaces compatible with existing I/O syscall interfaces to ensure compatibility with unmodified user applications. By prioritizing usability, RAKIS strives to provide a seamless and transparent integration while harnessing the advantages of FIOKPs within SGX enclaves.

Architecture. The architecture of RAKIS is illustrated in Figure 3. RAKIS comprises four distinct modules, each assigned specific responsibilities and tasks within the system. The first module is the FastPath Module (FM) (§4.1). It serves as the interface between the enclave and the untrusted host OS. Its purpose is to deliver data via FIOKPs into the enclave. Importantly, it achieves this by utilizing only the shared untrusted memory without requiring any enclave exits. The second module is the Service Module (SM) (§4.2), which builds on top of the FM. Given the vast functionality gap between what is delivered by FIOKPs and what is expected by unmodified user applications utilizing regular IO syscalls, the SM steps in to bridge this disparity. It provides RAKIS with the essential functionalities to seamlessly hook into the regular syscall API. The third module is the Monitor Module (MM) (§4.3), situated outside the enclave. Its primary role is to oversee the FIOKP data structures within shared memory and issue the needed syscalls to operate the FIOKPs on behalf of the FM. Lastly, the Testing Module (TM) plays a crucial role in ensuring the robustness and security of the system. The design and implementation of the TM is elaborated upon in (§5), as it stems from our security analysis.

IO integration As shown in Figure 3, RAKIS integrates two FIOKPs: XDP and `io_uring`. RAKIS leverages the XDP FIOKP to facilitate exitless UDP IO inside SGX enclaves. For incoming packets, RAKIS configures the host OS's XDP primitive to direct UDP packets into shared untrusted memory. Subsequently, RAKIS's FM securely retrieves these packets from the shared memory, transporting them into the enclave's trusted memory. Once inside the enclave, these packets undergo processing through a UDP/IP stack, priming them for user consumption. For outgoing packets, RAKIS interfaces

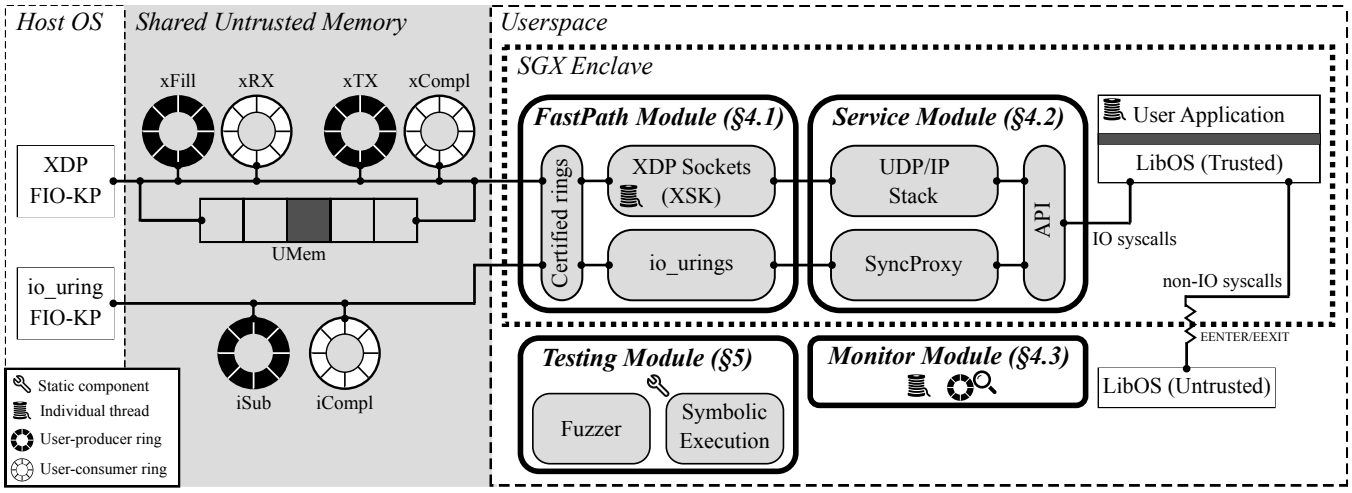


Figure 3. RAKIS architecture: FastPath Module (FM) (§4.1) and Service Module (SM) (§4.2) are inside the SGX enclave. The Monitor Module (MM) (§4.3) exclusively runs outside the enclave. The Testing Module (TM) (§5) serves as a static component that offers testing tools.

with the trusted component of a LibOS to handle UDP `send` syscalls via its UDP/IP stack and the XDP primitive. Similarly, for TCP and file IO functionalities, RAKIS leverages the power of `io_uring`. To streamline this process, RAKIS integrates with the trusted component of a LibOS, enabling it to handle IO syscalls without necessitating enclave exits.

4 Design and Implementation

4.1 FastPath Module (FM)

The FM is the central component of RAKIS, serving as the main module responsible for incorporating and supporting FIOKP capabilities. It runs inside the SGX enclave and interacts directly with the host OS using untrusted memory for FIOKP operations. FMs take the role of the user in FIOKP operations, facing an untrusted host OS. The FM abstracts all untrusted interactions with the host OS, including control values and data structures necessary for FIOKPs, and therefore alleviates higher-level components from the burden of processing on untrusted non-enclave memory.

Validating the initialization data. The FM's initialization values are derived from user configuration variables and values returned by the initialization syscalls of the FIOKP. These values are expected to remain constant throughout the lifetime of the FM. Additionally, RAKIS ensures the consistency of syscall returned values by deriving them independently from the host OS whenever possible. For instance, instead of relying on the host OS-provided ring mask value, RAKIS calculates it based on the user-provided ring size. The top rows of Table 2 present the collection of initialization values employed by RAKIS, along with the corresponding checks that are performed on them to validate their correctness.

Validating the runtime data. While SGX offers protections for enclave memory, manipulating the enclave program can compromise these protections, especially if enclave data is maliciously copied outside or overwritten with untrusted content. Thus, a key objective of the FM is to certify the interactions carried out by FIOKPs across the trust boundaries with the host OS. To carry out those interactions, they rely on three data categories that reside in shared untrusted memory: *1) User data values*, such as network packets and IO operations status codes, *2) memory offsets* that point within agreed-upon buffers, and *3) ring control values*, such as the producer and consumer values of the shared rings. The FM verifies all three data categories to facilitate secure FIOKP interactions. The bottom rows of Table 2 provide a list of the data items falling within each of the data category.

Validating the shared rings. At initialization, FIOKPs shared ring structures are setup by the host OS in the request of the FM. In particular, these rings involve the host OS either producing the data (i.e., the producer) consumed by the FM (i.e., the consumer), or vice versa. The producer and the consumer of a ring structure share three ring control variables: the ring size, the producer value and the consumer value. For the FM, the ring size is an immutable user configuration that is copied to trusted memory at RAKIS's startup. However, in runtime, the consumer and producer variables must reside in the shared untrusted memory for direct access by both actors. Yet, inherent to the design of ring data structures, and based on the role of FM within the ring, the producer and consumer values serve distinct purposes: one acts strictly as write-only, while the other is exclusively read-only. For instance, in a ring where FM acts as the producer, the producer value in shared untrusted memory is exclusively write-accessible, whereas the consumer value must be read and verified before utilization. Conversely, if FM operates as the consumer within the ring, the consumer value in shared untrusted memory becomes exclusively write-only, while the producer value requires reading and subsequent verification

Stage	Category	Data Items	Checks	Fail Action
Initialization	File descriptors	XSK fd, `io_uring` fd	fd >= 0	Abort startup
	Memory pointers	XSK rings & UMem `io_uring` rings	1) Non-overlapping. 2) Exclusively in untrusted memory.	Abort startup
Operation	User data values	Incoming network packets & read file contents	No checks / Left for application-level protocols i.e. TLS	-
		IO operations status codes	Return code is expected for the requested operation	Return -EPERM
	Memory offsets	UMem frame `io_uring` iCompl	1) UMem frame correctly owned by routine. 2) Offset & size fully points within UMem/iCompl.	Refuse and advance consumer
	Untrusted ring control values	Producer value in rings where RAKIS is consumer	0 <= (Producer^{u} - Consumer^{t}) <= size^{t}	Do not update trusted producer
		Consumer value rings where RAKIS is producer	0 <= (Producer^{t} - Consumer^{u}) <= size^{t}	Do not update trusted consumer

Table 2. Enumeration of untrusted data utilized by RAKIS's FM to facilitate the operation of XDP and `io_uring` FIOKPs. Each untrusted data item has a trusted version stored within the enclave memory. Untrusted data are copied to trusted memory prior to applying any check. Successful completion of the checks allows the data to update its trusted version and be utilized by RAKIS for its operation. In the event of failed checks, RAKIS will perform the action specified in the Fail Action column. (^{u}) denotes an untrusted value. (^{t}) denotes a trusted value.

before use. Hence, to protect against manipulation of the producer and consumer values, RAKIS adopt a secure-by-design approach by maintaining trusted versions of all ring control variables inside trusted memory. Consequently, before updating the trusted version of a read-only control value, the retrieved value is subjected to distinct security checks, based on FM's role in the ring, as shown in Table 2.

Enabling the XDP primitive. XSK relies on the use of four RAKIS-certified rings and a data buffer called the UMem. The UMem and the four rings must reside in the shared untrusted memory for host OS access. Moreover, RAKIS can manage multiple XSKs, each associated with a separate FM and requiring dedicated rings and UMem.

Initialization of XSK. The initialization process for XSK is intricate, involving a series of at least 14 syscalls to setup the four shared rings, allocate the UMem area, and load the XDP eBPF program. During the startup of RAKIS, initialization routines for XSKs are executed outside the SGX enclave. After each XSK is initialized, each XSK FM acquires five memory pointers that point to the four shared rings and the UMem. The FM verifies that the five pointers are non-overlapping and that they reside exclusively in shared untrusted memory.

UMem frames allocator. During operating of XSK, it is the FM's responsibility to manage the ownership of UMem frames. Initially, all UMem frames are owned by the FM. To send or receive network packets, the FM and the host OS exchange ownership of UMem frames using xTX and xCompl for the send routine, and xFill and xRX for the receive routine. A critical observation of XSK operation is that the FM must anticipate consuming only the UMem frames that it had previously produced within the corresponding ring in the same routine. To ensure the integrity of the FM UMem frame pool and prevent potential malicious manipulation by the untrusted host OS, the FM must verify the ownership of the consumed frames before adding them to its UMem frame pool for future allocations. This verification step is crucial in preventing scenarios where the host OS could deceive the FM into populating its UMem frame pool with invalid and overlapping frames. By confirming ownership, the FM safeguards against unexpected paths in the UMem frame allocator and prevents multiple entities from erroneously claiming ownership of the same UMem frame. To that end, RAKIS maintains a map to track the ownership of each UMem frame, as well as the specific routine in which it is currently being utilized. In the event that RAKIS encounters an invalid or unexpected UMem frame from either xCompl or xRX, it can swiftly detect this through its ownership tracker. In such cases, RAKIS takes appropriate action by advancing the ring consumer value and refusing to process the problematic UMem frame. This meticulous validation mechanism ensures the robustness of the UMem frame allocator and enhances RAKIS's resilience against potential malicious activities orchestrated by the host OS.

Quality of service assurance. With a properly configured XSK, incoming packets are placed in UMem frames that were provided by the FM in the xFill ring. However, if the FM fails to produce sufficient UMem frames to accommodate incoming packets, the excess packets will be dropped due to memory constraints. This puts emphasis on efficient management of XSKs, as delays or inefficiencies can directly result in packet drops. To address this, we assign a distinct SGX enclave thread to each XSK managed by RAKIS. Each XSK FM thread is tasked with transferring incoming packets to trusted memory and subsequently invoking the UDP/IP stack for processing, readying the received packets for user access.

Enabling the `io_uring` primitive. `io_uring` utilizes two RAKIS-certified shared rings: iSub and iCompl, to submit and receive IO requests. Similar to XSK's, the `io_uring` initialization routine is completed upon RAKIS's startup, yielding two pointers to iSub and iCompl. RAKIS employs the `io_uring` primitive to handle five syscalls: `send` and `receive` for TCP sockets, along with `read`, `write`, and `poll`. When IO requests arrive at the `io_uring` FM, it swiftly submits them to the host OS via the iSub. Once the IO task concludes, the host OS generates a corresponding entry in the iCompl for FM's consumption. To optimize performance and avoid potential contention issues, the `io_uring` FM runs in the same thread as the IO requester. In multi-threaded user programs, RAKIS creates a separate `io_uring` FM for each user thread where each `io_uring` FM operates independently.

Implementation. We wrote our two FMs in a total of 2K lines of C code. To implement RAKIS's certified rings, we developed a dedicated C file that accepts pointers to untrusted ring values as input. This code module provides accessors equipped with checks to maintain valid ring states. Both the producer and consumer values, represented as 32-bit unsigned integers (`u32`), can cyclically wrap around upon reaching the maximum `u32` value. However, when applied as they are, the checks outlined in Table 2 have limitations in addressing scenarios where the producer value wraps around before the consumer value does. Thus, our implementation handles this edge case by introducing supplementary checks that enforce the same constraints while accommodating scenarios involving wrapping around. In addition, we intentionally refrained from using the official userspace libraries associated with XDP and `io_uring`, namely `libxdp` and `liburing`. These libraries are not designed with SGX trust assumptions in mind, potentially introducing wide attack surfaces if contained inside SGX enclaves, as discussed in §5. Moreover, these libraries are significantly larger in size, with `liburing` being over 35K lines of code and `libxdp` (with its dependance on `libbpf`) being over 130K lines of code. The extensive size of these libraries is attributed to their broad applicability. For instance, `libbpf` supports over 25 BPF commands, RAKIS uses only two BPF commands to enable XDP sockets. Similarly, `liburing` supports over 40 `io_uring` operations, whereas RAKIS uses only 8 for file and TCP IO and polling. Therefore, we chose to develop custom low-level libraries within RAKIS tailored to our specific requirements and threat model.

4.2 Service Module (SM)

The SM acts as an intermediary layer, extending the capabilities of low-level FIOKPs to fulfill the functional requirements of user-level syscall interfaces. The SM exclusively processes trusted in-enclave buffers, relaying on FMs to handle untrusted buffers.

UDP/IP stack To enable RAKIS to employ XSKs for UDP IO, the UDP/IP stack functions as an essential bridge. It connects the functionalities of XSK FMs, which handle layer-2 data frames, with syscalls like `recv` and `send`, which deliver application-level data. The UDP/IP stack is equipped with multi-threading capabilities, allowing for simultaneous usage of the stack with minimal contention. Broadly, two distinct thread types interact with the UDP/IP stack: XSK FM threads and user threads. The XSK FM threads utilize the UDP/IP stack to process incoming packets. After the XSK FM securely copies layer-2 data frames into trusted memory, the UDP/IP stack takes over for processing. Based on packet validity, the stack either routes these packets to a UDP socket queue for user consumption or drops them. On the other hand, user threads interface directly with the UDP/IP stack to serve UDP sockets syscalls like `recv` and `send`. For the `recv` syscall, user threads aim to retrieve data from the relevant UDP socket queue. For the `send` syscall, user threads utilize the stack to encapsulate the user data before transferring it directly to the XSK UMem for transmission by the host OS.

SyncProxy. The SyncProxy enables RAKIS to use `io_uring` FIOKP to serve five syscalls: `send` and `receive` for TCP sockets, along with `read`, `write`, and `poll`. The SyncProxy receives those IO syscalls synchronously from the API submodule, and acts as an intermediary to forward them to an `io_uring` FM. Since the user expects the results synchronously, the SyncProxy blocks and wait for the user request to complete as signaled by the `io_uring` FM.

API. The API submodule integrates with existing SGX LibOSs to reroute IO syscalls for processing by RAKIS, thereby bypassing the need to send them to the untrusted part of the LibOS via enclave exits. The primary role of the API submodule is to interface with the UDP/IP stack and SyncProxy to handle the IO syscalls directed to RAKIS. In addition, the API submodule plays a crucial role in coordinating `poll` syscalls involving different IO providers. Consider a `poll` syscall encompassing two file descriptors: one for a UDP socket overseen by RAKIS's UDP/IP stack and another for a TCP socket handled by the host OS. In this context, exclusively monitoring events on one socket, neglecting the other, might introduce delays in processing events from the overlooked socket. To address this, the API submodule initiates the `poll` syscall for both file descriptors, directing them to their respective IO providers. Following this, the API submodule engages in a busy-wait mechanism, monitoring events on both sockets via the UDP/IP submodule for the UDP socket and the SyncProxy submodule for the TCP socket. Upon the occurrence of an event on either socket, the API submodule aggregates the outcomes and forwards them to the user.

Implementation. For the UDP/IP stack, we based our implementation on LWIP [27]. LWIP is a comprehensive TCP/IP network stack that supports a plethora of network protocols beyond the scope of RAKIS's requirements. Consequently, our initial task involved refining LWIP by eliminating unnecessary functionalities, retaining only those essential for processing UDP/IP packets. This refinement significantly downsized LWIP from its original size of over 80K LoC to

under 5K LoC. Notably, LWIP's support for multi-threading presented challenges; its reliance on a global lock became increasingly problematic as thread counts rose, resulting in contention issues. Addressing this, we transitioned the stack from a singular global lock to a more efficient system using multiple smaller read/write locks on the stack's shared state. Regarding the SyncProxy, we designed it as a per-thread pass-through stub that forwards requests directly to `io_uring` FMs. Lastly, we opted to use Gramine [6] as the LibOS to intercept and forward IO syscalls to RAKIS.

4.3 Monitor Module (MM)

While FIOKPs aim to minimize dependency on syscalls to mitigate the costly context-switch overhead, total elimination of syscalls for XDP and `io_uring` remains unfeasible. In FIOKPs rings where the kernel serves as the consumer, syscalls become essential to prompt the kernel to process user requests. Although FIOKPs introduce flags such as `XDP_USE_NEED_WAKEUP` for XDP and `IORING_SETUP_SQPOLL` for `io_uring` to reduce the frequency of syscalls, they do not eradicate them entirely [16, 19]. Rather than burdening all FMs with enclave exit overhead for these infrequent syscalls, the MM efficiently oversees all RAKIS's FIOKPs with a dedicated thread consistently operating outside the enclave. However, this decision mandates synchronization between the MM and each FIOKP FM to guarantee the MM's prompt syscall execution on behalf of the FMs. To address this synchronization challenge, the MM continually monitors the shared FIOKP rings in untrusted memory. Specifically, it observes the producer value within the FIOKP rings where RAKIS acts as the producer. Upon recognizing that an FM has progressed its producer value, the MM invokes a syscall, prompting the host OS to address RAKIS's requests. Thus, by monitoring theses FIOKP rings, the MM avoids the need for direct interactions with individual FMs to determine the timing and nature of the syscalls. Furthermore, the design of XDP and `io_uring` allows syscalls to instruct the kernel about user requests without blocking and without immediately executing the requested IO operations. Instead, both XDP and `io_uring` employ dedicated kernel routines, scheduled in response to the syscall, to handle user requests from the shared rings [20–22]. Benefiting from these non-blocking syscalls, the MM can efficiently oversee multiple FIOKPs using one thread.

Implementation. The MM thread is spawned during the final phase of RAKIS's initialization. Operating outside the enclave, the MM thread is provided with a set of pointers that point to the shared untrusted producer values of the rings where RAKIS acts as the producer. These encompass xFill and xTX rings within XSK, as well as iSub of `io_uring`. Additionally, the MM thread is provided with the file descriptors of the XSKs and `io_urings` to be used in the syscalls. During runtime, the MM thread continuously monitors the producer values of each ring, triggering the `recvfrom` syscall when an XSK FM progresses the value of xFill's producer, the `sendto` syscall when an XSK FM progresses the value of xTX's producer, and the `io_uring_enter` syscall when an `io_uring` FM progresses the value of iSub's producer.

5 Security Analysis

In this section, we delve into pivotal security considerations for RAKIS. To set the context, we highlight potential vulnerabilities when settling with user libraries accompanying FIOKPs, like `libxdp`[43] and `liburing`[25], to use within SGX enclaves. Particularly, these libraries are crafted with assumptions that trust the host OS. Consequently, their deployment within SGX enclaves could enable a malicious host OS to alter enclave program behavior or access enclave data.

`libxdp` case study. `libxdp` provides the function `xsk_prod_nb_free`[44] to check the number of free elements to produce in a XSK ring. The function essentially determines the count by utilizing the consumer value read from shared memory provided by the host OS. However, the function does not verify that the calculated number of free elements must be less than or equal to the ring size. Such oversight could unintentionally cause anomalies in user programs using `libxdp` within an SGX enclave, possibly resulting in buffer overflows. On the other hand, RAKIS implements rigorous checks on all ring control values, ensuring a consistently valid ring state irrespective of the values supplied by the host OS.

`liburing` case study. Similarly, `liburing` exhibits analogous concerns. For instance, the function `io_uring_queue_init` within `liburing` is intended as a utility to initialize the `io_uring` context. However, owing to `liburing`'s lack of familiarity with the trust assumptions specific to SGX enclaves, it overlooks a crucial validation step: ensuring that pointers within the `io_uring` context exclusively reference shared untrusted memory areas rather than enclave memory. This omission paves the way for a malicious host OS to compromise SGX confidentiality by extracting enclave-protected data, as shown in Appendix A. Conversely, RAKIS rigorously verifies all initialization values before accepting them, recognizing they were provided by an untrusted host OS.

RAKIS's TCB. When utilizing RAKIS, it's crucial to highlight its minimal impact on the TCB size. Among RAKIS's components, the FM and the SM are integrated into the SGX enclave's TCB. The FM component, while only consisting of 2K LoC, expands the attack surface due to its direct interactions with the untrusted host OS via untrusted memory. Even though FMs contain runtime checks on all data read from untruted memory, programming errors could still lead to incomplete runtime checks. Therefore, we model the FMs interactions with the untrusted host OS in a set of constraints and assumptions. We then verify that our FM code adheres to our model, as detailed in §5.1. On the other hand, the SM consists of 6K LoC. While the SM does not directly interface with the host OS, its interaction with untrusted data calls for rigorous testing. Notably, within the SM's architecture, we

focus our testing predominantly on the UDP/IP stack due to its size and pivotal role in data processing. As a result, we statically test RAKIS's UDP/IP stack using fuzzing techniques to guarantee its proper handling of untrusted data (§5.2). The distinction between our testing efforts for the FM and the SM is based on the following factors: model checking is more thorough and feasible for the FM, which has a small code size and lower complexity. The SM, however, is larger and more complex, making model checking infeasible. Therefore, we opted for fuzzing the SM instead. Lastly, the MM is entirely outside RAKIS's TCB and does not interact with any of RAKIS's trusted components. Given that the MM's functionality only affects data availability, it is excluded from this security analysis. Based on our security analysis of RAKIS's design, we crafted the TM to rigorously test the trusted components of RAKIS. The TM comprises two distinct binaries tailored for our testing endeavors: the verification binary, which executes our model checking atop the FM, and a fuzzing harness binary integrated with the SM UDP/IP stack for fuzz testing. The TM does not affect RAKIS's performance, as it is only used for testing and is not included in the production version of RAKIS.

5.1 Verifying the FM

The primary goal of modeling the FMs interactions with the host OS is to uphold the integrity of the FM's state, especially during the processing of untrusted data for FIOKP operations. We aim to ascertain that any value obtained from untrusted memory do not result in an unexpected state within the FM. Consequently, our model checking efforts aspire to reduce the host OS to a remote adversary, thereby limiting the attack surface to only the exchanged IO data.

Scope. We model all interactions between the FMs and the host OS that pertains to managing the XSK and `io_uring` primitives. For the XSK primitive, this encompass managing the rings to send and receive network frames, as well as verifying the memory location and limits of consumed XSK UMem frames. For the `io_uring` primitive, our model covers the ring operations needed to perform the various IO operations as well as verifying the memory location and limits of all consumed untrusted data.

Specifying FIOKPs rings. FMs within RAKIS rely exclusively on RAKIS-certified rings to manage FIOKP rings located in shared untrusted memory. RAKIS-certified rings must consistently maintain valid ring control variables within enclave trusted memory, irrespective of the values present in the shared untrusted memory. Since untrusted memory is overseen by an untrusted OS, our model cannot hinge on the consistency of untrusted values. Thus, our model ensures that for RAKIS's rings R, where Pt represents the trusted producer value, Ct denotes the trusted consumer value, and St signifies the trusted ring size, the following constraint remains valid following any ring operation on either FIOKP primitive:

$$\forall R : \{Pt, Ct, St\},\ 0 \leq (Pt - Ct) \leq St \tag{1}$$

Specifying untrusted memory access. FMs frequently access untrusted shared memory to read and write IO data. Often, the memory address for these operations relies on untrusted index provided by an untrusted host OS. For example, to receive network frame, the XSK FM reads the index of a UMem frame from the xRX ring. However, all untrusted memory accesses by the FM are performed on memory slots within array-like memory objects (i.e., UMem areas and ring data structures), which have immutable base addresses, element counts, and element sizes. Therefore, we include constraints in our model to ensure that all untrusted memory accesses referenced by an untrusted index points to a memory slot within a predefined memory object which falls completely within a larger memory segment designated as untrusted memory.

Implementation. The FM's model checking employs symbolic execution techniques. To facilitate this process, as part of the TM, we have crafted a verification binary that leverages the KLEE [5] symbolic execution engine to symbolically explore all the program states of the FM. We designate the shared untrusted memory, which the FM would use for FIOKP operations, as input and, consequently, mark them as symbolic. As these symbolic values traverse the various control paths of the FM, emulating interactions with the host OS, KLEE evaluates them to ensure the resulting program state aligns with our model. The verification binary begins by activating the FM using its initialization functions. Following this, the verification binary utilizes FM functions to simulate various IO tasks. For XSK operations, the binary triggers the network send and receive routines, managing the symbolic XSK rings. In contrast, for `io_uring`, the binary initiates the file read and write procedures over the symbolic `io_uring` rings. Throughout these IO tasks, we incorporate `klee_assert` statements, which are symbolic validation checks used to ensure that the FM's trusted ring control values consistently adhere to the constraints of our model, regardless of the utilization of symbolic untrusted values. Ring state assert statements are injected before and after any ring operation. The memory access assert statements are injected before any memory access to any untrusted memory object. Particularly, KLEE explores all program paths of the FM and validates that there are no paths that violate any of the constraints we specified in our model.

Limitations. Due to limitations of KLEE, true interactions with the host OS to operate FIOKPs are not feasible. This is because any value extending beyond KLEE's tested process boundaries loses its symbolic representation (i.e., concreteized). Thus, as a workaround, our verification binary treats all values that would be acquired from the host OS, including initialization values, as symbolic, allowing us to examine all potential paths within the FM regardless of the concrete value

supplied by the host OS. Additionally, KLEE lacks the capability to differentiate between trusted and untrusted memory accesses. This limitation hinders our ability to confirm that ring operations exclusively utilize trusted data and that accesses to untrusted ring control values are strictly limited to read-only or write-only operations. As part of our future endeavors, we aim to enhance the symbolic engine's awareness of trust boundaries.

5.2 Fuzzing the UDP/IP stack

The XSK FMs supply the UDP/IP stack with incoming network packets, representing them as pointers and lengths within the enclave's trusted memory. Thus, while the stack avoids direct interaction with untrusted memory, it remains vulnerable to manipulated packets from the host OS. To enhance the stack's security, we streamlined its functionalities, significantly reducing potential attack vectors. Following this, we conducted a fuzzing campaign on the UDP/IP stack to proactively identify and address vulnerabilities.

Scope. We narrow our fuzzing efforts exclusively to the UDP/IP stack within SM. As detailed in §4.2, the UDP/IP stack features dual entry points: one interfacing with users for handling UDP IO syscalls and another interfacing with the host OS to handle incoming network packets. Prioritizing protection against potential threats from the untrusted host OS, our fuzzing focuses specifically on incoming packets from the host OS as the primary input source.

Implementation. As part of the TM, we developed our fuzzing harness binary and employed AFL++[12], state-of-the-art fuzzing tool, to execute it. The fuzzing binary initializes the UDP/IP stack, then reads packets from `stdin` via AFL++ for processing. To enhance our code coverage and broaden the program states accessible to our fuzzer, the fuzzing binary interacts with the stack, mimicking user actions to receive packets, configure network sockets, and echo received packets using the send routines. We launched a 144 CPU-hour fuzzing campaign across three CPU cores. This effort yielded an 84% line coverage and 76% branch coverage. Upon analyzing our fuzzing code coverage, we recognize the potential to enhance it by incorporating additional emulated user interactions, exploring varied stack states related to incoming data processing. Importantly, we emphasize that this fuzzing effort is ongoing and a continuous commitment.

Limitations. A limitation of AFL++ is the absence of support for multi-threaded fuzzing. This restricts our ability to test the stack's parallel processing of incoming packets and detect potential race conditions. Moving forward, we aim to expand our fuzzing strategies by incorporating tools equipped to identify inter-thread vulnerabilities.

6 Evaluation

We conducted all of our performance experiments in five different test environments: *Native* executes the test program natively on the host OS, serving as the baseline of our experiments results. *Gramine-SGX* and *Gramine-Direct* execute the test program within SGX enclaves and outside SGX enclaves, respectively, leveraging Gramine [1] , a state-of-the-art library OS. These two configurations emphasize the performance impact of enclave exits during syscalls. Rakis-*SGX* and Rakis-*Direct* run the test program inside and outside of SGX enclaves, respectively, utilizing RAKIS. These two configurations underscore the performance benefits of employing FIOKPs with unmodified user programs that use standard POSIX syscalls (i.e., `send` and `recv`), inside and outside SGX enclaves. We sought to incorporate additional baselines that use fast IO primitives like DPDK in our performance evaluation. To our knowledge, [39] is the only project known to use DPDK in SGX enclaves. However, we were unable to meet their hardware requirements. Additionally, we could not identify any projects that run unmodified programs (that use standard POSIX syscalls) on top of XDP sockets or `io_uring` for non-SGX baselines. Lastly, all of our experiments were conducted on a single machine equipped with Intel(R) Xeon(R) Gold 6312U CPU @ 2.40GHz with 48 cores all on a single NUMA node. The machine have 64GB of RAM and one NIC with two ethernet interfaces which are wired in a loopback configuration with link capacity of 25Gbps.

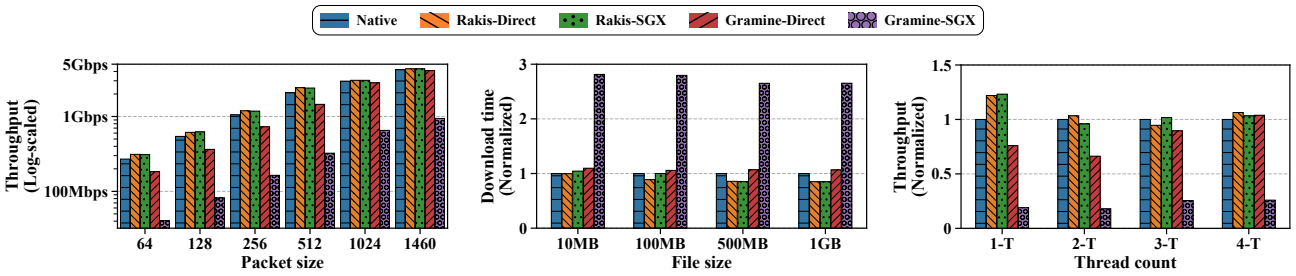
6.1 UDP IO Evaluation

RAKIS employs the XDP FIOKP to provide fast UDP IO for unmodified user program which use regular syscall interfaces such as `recv` and `send`. In our UDP IO evaluation, we ensured the same amount of memory was allocated for packet handling across all test environments: The Linux Kernel was set with a 16MB UDP send buffer and a 2K NIC queue length. Likewise, RAKIS's XSKs were configured with 16MB UMems and 2K for the size of each XSK ring. For the iperf3 and curl experiments, we employed a single XSK. For the Memcached experiment, we utilized four XSKs each overseen by its respective XSK FM thread.

iperf3. We employed iperf3 (v3.13) [17] to measure the UDP socket throughput of RAKIS. In our experiment setup, the iperf3 client operated natively within its own Linux network namespace, whereas the iperf3 server was run under the five settings previously outlined. We conducted 10-second UDP throughput tests, utilizing packet sizes up to 1460 bytes to align with standard Maximum Transmission Unit (MTU) values. The transmission rate was set at 25Gbps, reflecting the peak capability of our physical NIC.

Results of our iperf3 experiment are shown in Figure 4(a). Gramine-SGX experiences an average throughput reduction of 78% compared to Gramine-Direct (83% vs native). This reduction occurs due to the necessity of exiting the enclave for each UDP IO syscall. Gramine-Direct incurs an average overhead of 25% relative to native execution. This overhead

[1] We used Gramine-v1.5, the latest release at the time of our experiments.



(a) iperf3 (b) Curl (c) Memcached

Figure 4. Evaluating UDP IO performance: iperf3 network test, Curl download duration, and Memcached throughput across Native, RAKIS-Direct, RAKIS-SGX, Gramine-Direct, and Gramine-SGX test environments.

becomes particularly evident with smaller packet sizes, highlighting the amplified LibOS overhead due to frequent calls. Conversely, RAKIS-Direct leverages an XSK to achieve an average 11% increase in UDP throughput compared to native execution, all without necessitating modifications to user programs. RAKIS-SGX experiences no overhead relative to RAKIS-Direct since both leverage shared memory for IO, thereby eliminating enclave exit costs.

Curl. Curl [8] is a command-line tool used for transferring data over various network protocols. Recently, experimental support for the QUIC protocol over UDP has been added to Curl [9]. In our experiment, we set up a web server running natively on the same machine to serve files of sizes ranging from 10MB to 1GB via the QUIC protocol. Subsequently, we evaluated the total time to download the files using Curl. Figure 4(b) shows that Gramine-SGX, on average, exhibits 2.5x longer download times than native execution. Conversely, RAKIS achieves comparable download times to native execution, whether inside or outside SGX enclaves.

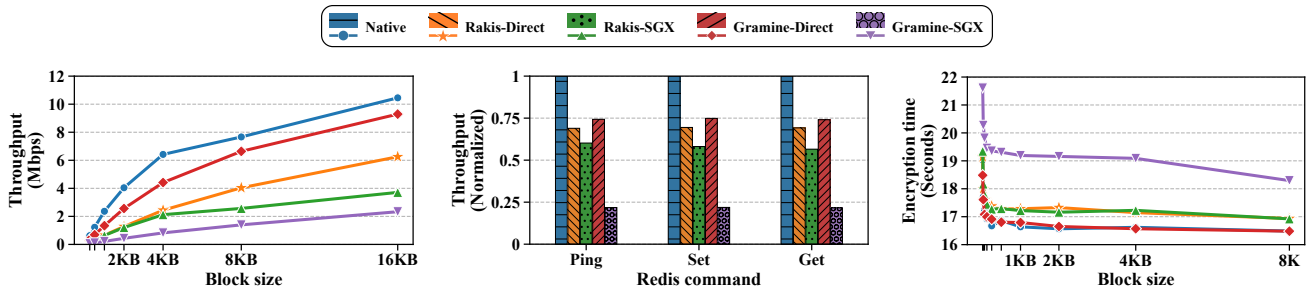
Memcached. Memcached [29] is a high-performance, distributed memory caching system that stores and retrieves data in key-value pairs. Memcached supports communication with clients over UDP protocol. To benchmark our Memcached deployment, we employed `memaslap`, a load generation and benchmark tool specifically designed for Memcached servers. In our experiment, the Memcached server operated within our five test environments, while the benchmark client functioned natively within its dedicated Linux network namespace. We set up the benchmark client with four threads and 32 simultaneous connections. Meanwhile, we conducted the experiment multiple times, adjusting the Memcached server to utilize varying thread counts. The results of our Memcached experiment are presented in Figure 4(c). RAKIS consistently matches native execution performance for Memcached across varying thread counts, both inside and outside the enclave. In addition, RAKIS-SGX achieves an impressive 4.6x average throughput enhancement compared to Gramine-SGX. Interestingly, the results show a slight performance advantage of Gramine-Direct over Native when Memcached runs with four threads. This advantage arises because Gramine-Direct can handle some `futex` syscalls internally without a syscall context switch, thus eliminating the associated overhead. This behavior is analogous to RAKIS, which also avoids context switches for IO syscalls, which is particularly beneficial in workloads with frequent IO syscalls, minimal userspace processing, and no thread or lock management, such as iperf3, Curl, and single-threaded Memcached.

6.2 TCP and File IO Evaluation

RAKIS employs the `io_uring` FIOKP to enable TCP and file IO for unmodified user program. Our evaluation comprises benchmarks for file writing, a TCP-intensive program, and a file-intensive program.

fstime. In order to evaluate the file write throughput in RAKIS, we utilized the `fstime` tool from the UnixBench suite [41]. The test involved performing repeated writes to a single file using the `write` syscall with a provided block size for a specified duration. We conducted the test multiple times, with block sizes increasing exponentially, and then took the average of the results from five test iterations. Our experiment results are depicted in Figure 5(a). Gramine-Direct incurs added overhead due to LibOS handling, which tends to decrease with increased data sizes due to fewer calls. Compared to native execution, both RAKIS-Direct and RAKIS-SGX experience overheads due to utilization of asynchronous `io_uring` operations. Specifically, when employing `io_uring` for synchronous write operations, it necessitates waiting for another thread to execute the task instead of allowing synchronous execution within the same thread. In addition, for larger block sizes, RAKIS-SGX exhibits increased overhead relative to RAKIS-Direct, attributable to memory copy operations from encrypted enclave memory to shared untrusted memory as detailed in previous works [42]. However, RAKIS-SGX maintains a 2.8x performance advantage over Gramine-SGX due to the cost associated with enclave exits in Gramine-SGX.

Redis. Redis [35] is an open-source, in-memory data structure store. Redis clients communicate with the Redis server over TCP. For benchmarking performance, we employed a built-in benchmarking client called `redis-benchmark`, focusing on three Redis commands: Ping, Set, and Get. As



(a) fstime (b) Redis (c) MCrypt

Figure 5. Evaluating TCP and file IO performance: fstime file write test, Redis throughput, and MCrypt encryption time across Native, RAKIS-Direct, RAKIS-SGX, Gramine-Direct, and Gramine-SGX test environments.

RAKIS does not currently support `epoll` syscall, we compiled Redis to use the `select` syscall instead. We ran the `redis-benchmark` tool natively with one thread and 50 parallel connections. In Figure 5(b), we show the normalized throughput of Redis across our five test environments. On average, RAKIS-SGX surpasses Gramine-SGX by 2.6x in performance. Nevertheless, when compared to native execution, both RAKIS-Direct and RAKIS-SGX demonstrate an average overheads of 30% and 40%, respectively.

MCrypt. MCrypt [28] is a command-line tool used for encrypting files with various encryption algorithms. For our experiment, we encrypt a 1GB file utilizing varying file read block sizes. As shown in Figure 5(c), on average, RAKIS exhibits 3% overhead compared to native execution, while demonstrating a 10% reduction in execution time compared to Gramine-SGX.

7 Discussion

Data protection. Like prior works that enable IO syscalls via enclave exits [1, 3, 6], RAKIS lacks inherent mechanisms to ensure the confidentiality or integrity of user data during IO operations. User applications can depend on application-layer protocols such as Transport Layer Security (TLS) for these protections. However, RAKIS integrates a UDP/IP stack within the enclave, enabling the integration of layer-3 tunnels like Wireguard [10]. This capability paves the way for efficiently running programs needing protection over layer-3 network packets, such as software switches, within SGX enclaves without reliance on host OS trust.

Resource utilization. In total, RAKIS requires the use of a minimum of two additional user threads: one inside the enclave to process incoming network packets on a XSK (FM), and another outside the enclave to ensure the host OS handles RAKIS's IO requests promptly (MM). For memory, RAKIS maintains an in-enclave packet queue to buffer incoming packets before transferring them to the user-provided buffer. Outside the enclave, RAKIS manages the UMem for XSK, along with the `io_uring` and XSK ring structures. These memory parameters can be configured based on the user's workload. Compared to previous works, RAKIS does not incur additional resource requirements. For example, the design of Rkt-IO [39] incurs higher resource requirements by incorporating DPDK inside SGX enclaves, which necessitates additional CPU cores and memory to manage SGX-LKL [34] for filesystem and TCP/IP operations. This increase in resource demand is due to DPDK heavy runtime requirements, even for single-threaded user applications running within the enclave.

TCP Stack Considerations. RAKIS employs `io_uring` primitive for TCP IO operations. While it's feasible to incorporate a TCP stack into RAKIS's network stack to utilize the XDP primitive for TCP IO, we chose not to pursue this avenue. Introducing a TCP stack inside the SGX enclave would considerably enlarge the TCB of RAKIS due to the substantial size and complexity of the TCP stack, conflicting with RAKIS's design objectives. Moreover, the established maturity of the Linux kernel's TCP stack, coupled with its access to hardware acceleration in modern NICs, significantly enhances its TCP performance. Attempting to replace this optimized stack within the enclave would entail substantial resource and performance compromises.

Deployment Simplicity. One standout feature of RAKIS is its ease of deployment. Specialized hardware is not necessary; all that is needed is a recent kernel equipped with XDP and `io_uring` capabilities. RAKIS's setup is streamlined, necessitating only essential networking parameters such as MAC address, IP address, and the specific NIC queue ID for XSK configuration. With its user-friendly setup, performance benefits, and uncompromised security, RAKIS offers an ideal blend for widespread adoption.

SGX future. SGX remains the only technology that provides a secure execution environment with a small TCB, particularly for scenarios where the OS and VMM are untrusted. Although Confidential Virtual Machines (CVMs) are gaining popularity, Intel SGX continues to be the preferred choice for solutions requiring strong security. For example, IBM employs SGX for the e-prescription system for the entire

German population [30], Signal uses it for their secure messenger [7] and FlashBots runs blockchain blocks builders within enclaves [13].

8 Related work

High-performance IO for SGX enclave programs. Several works have proposed approaches to provide high-performance IO in SGX enclaves [2, 33, 39, 40]. These works involve pulling entire userspace kernel-bypass libraries, such as DPDK [11] and SPDK [38], into the enclave to enable direct IO access for enclave programs. However, they suffer from several drawbacks: *1) Significant increase in TCB size.* DPDK consists of 1.9M LoC, while SPDK consists of 300K LoC. This results in a large TCB footprint. *2) Difficulty in deployment.* Both DPDK and SPDK have specific hardware requirements that must be met for successful deployment. *3) Inclusion of unnecessary components.* DPDK, for example, requires heavy OS features such as thread scheduling and direct IO access, which may not be needed by the enclave program. Finally, a study by Lefeuvre et al. [24] focused on secure and efficient IO within SGX enclaves, emphasizing attributes like security-by-design, high performance, and independence from specific TEE implementations. We believe that RAKIS's design aligns closely with the study's insights.

Shielded execution for unmodified programs. Frameworks such as [1, 3, 6, 34, 36, 37] are utilized to facilitate the deployment of applications within SGX enclaves without modifications. These frameworks provide a LibOS that would serve as an intermediary layer that abstract and handle syscalls within the enclave environment.

Switchless enclave syscalls. Frameworks like [6, 31, 34, 34, 42] address IO bottlenecks in TEEs by employing switchless asynchronous IO calls. This technique eliminates the need for expensive SGX enclave exits and leverages IO threads outside the enclave to enhance IO performance through asynchronous syscalls.

Use cases for shielded execution. Since its introduction, SGX enclaves have been utilized in diverse applications such as secure storage [23], data analytics [45], decentralized ledger [26] and content delivery networks [14]. In all these use cases, the IO overhead stemming from enclave exits posed challenges, leading to workarounds like external IO worker threads or simply accepting it as a trade-off for shielded execution. We anticipate that RAKIS can advance current practices, fostering greater adoption of SGX enclaves.

9 Conclusion

In conclusion, this paper introduces RAKIS, an end-to-end system designed to securely enable Linux FIOKPs within SGX enclaves. RAKIS seamlessly integrates with unmodified user applications, maintaining standard syscall interfaces while effectively minimizing the expansion of the TCB size. Through extensive benchmarking and real-world evaluations, RAKIS demonstrates notable performance enhancements in network and file IO within SGX enclaves compared to Gramine.

10 Acknowledgment

We thank the anonymous reviewers, and our shepherd, Christian Dietrich, for their constructive comments and valuable suggestions that helped improve the quality of this paper. This research was supported, in part, by the NSF award CNS-1749711, ONR under grant N00014-23-1-2095, DARPA V-SPELLS N66001-21-C-4024, and gifts from Facebook, Mozilla, Intel, VMware and Google.

References

[1] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 689–703. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov

[2] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. 2019. SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 173–190. https://www.usenix.org/conference/fast19/presentation/bailleu

[3] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 267–283. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/baumann

[4] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Hollywood, CA.

[5] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[6] Chia che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 645–658. https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai

[7] Graeme Connell. 2022. Technology Deep Dive: Building a Faster ORAM Layer for Enclaves. https://signal.org/blog/building-faster-oram/.

[8] Curl. 2023. A command line tool for transferring data with URLs. https://curl.se/.

[9] Curl. 2023. HTTP/3 with Curl. https://curl.se/docs/http3.html.

[10] Jason A Donenfeld. 2017. Wireguard: next generation kernel network tunnel.. In *NDSS*. 1–12.

[11] DPDK. 2023. Data Plane Development Kit (DPDK). http://www.dpdk.org.

[12] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.

[13] Chris Hager and Frieder Paape. 2023. Block Building inside SGX. https://writings.flashbots.net/block-building-inside-sgx.

[14] Stephen Herwig, Christina Garman, and Dave Levin. 2020. Achieving Keyless CDNs with Conclaves. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 735–751. https://www.usenix.org/conference/usenixsecurity20/presentation/herwig

[15] Intel 2023. Intel® Software Guard Extensions Programming Reference. https://www.intel.com/content/dam/develop/external/us/en/documents/329298-002-629101.pdf.

[16] io_uring. 2023. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf.

[17] iperf3. 2023. A TCP, UDP, and SCTP network bandwidth measurement tool. https://github.com/esnet/iperf.

[18] Magnus Karlsson. 2023. AF_XDP Sockets: High Performance Networking for Cloud-Native Networking Technology Guide. https://networkbuilders.intel.com/docs/networkbuilders/af-xdp-sockets-high-performance-networking-for-cloud-native-networking-technology-guide.pdf.

[19] Linux Kernel. 2023. AF_XDP - Linux Networking Documentation. https://www.kernel.org/doc/html/v5.10/networking/af_xdp.html.

[20] Linux Kernel. 2023. io_uring_enter in io_uring/io_uring.c (Linux v6.7-rc8). https://elixir.bootlin.com/linux/v6.7-rc8/source/io_uring/io_uring.c#L3688.

[21] Linux Kernel. 2023. xsk_recvmsg in net/xdp/xsk.c (Linux v6.7-rc8). https://elixir.bootlin.com/linux/v6.7-rc8/source/net/xdp/xsk.c#L901.

[22] Linux Kernel. 2023. xsk_sendmsg in net/xdp/xsk.c (Linux v6.7-rc8). https://elixir.bootlin.com/linux/v6.7-rc8/source/net/xdp/xsk.c#L856.

[23] Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, and Christof Fetzer. 2018. Pesos: Policy Enhanced Secure Object Store. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 25, 17 pages. https://doi.org/10.1145/3190508.3190518

[24] Hugo Lefeuvre, David Chisnall, Marios Kogias, and Pierre Olivier. 2023. Towards (Really) Safe and Fast Confidential I/O. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems* (Providence, RI, USA) *(HOTOS '23)*. Association for Computing Machinery, New York, NY, USA, 214–222. https://doi.org/10.1145/3593856.3595913

[25] liburing. 2023. The io_uring library. https://github.com/axboe/liburing.

[26] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gün Sirer, and Peter Pietzuch. 2019. Teechain: A Secure Payment Network with Asynchronous Blockchain Access. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 63–79. https://doi.org/10.1145/3341301.3359627

[27] LwIP. 2023. A small independent implementation of the TCP/IP protocol suite. https://github.com/lwip-tcpip/lwip.

[28] MCrypt. 2023. Encryption tool. https://mcrypt.sourceforge.net/.

[29] Memcached. 2023. A distributed memory object caching system. https://memcached.org/.

[30] Intel newsroom. 2021. Intel SGX Protects German Electronic Patient Records. https://www.intel.com/content/www/us/en/newsroom/news/intel-sgx-protects-german-electronic-patient-records.html.

[31] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS Services for SGX Enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 238–253. https://doi.org/10.1145/3064176.3064219

[32] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, Colorado.

[33] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. 2018. SafeBricks: Shielding Network Functions in the Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 201–216. https://www.usenix.org/conference/nsdi18/presentation/poddar

[34] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A. Sartakov, and Peter Pietzuch. 2020. SGX-LKL: Securing the Host OS Interface for Trusted Execution. arXiv:1908.11143 [cs.OS]

[35] Redis. 2023. The open source, in-memory data store used by millions of developers as a database, cache, streaming engine, and message broker. https://redis.io/.

[36] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and Efficient

Multitasking Inside a Single Enclave of Intel SGX. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 955–970. https://doi.org/10.1145/3373376.3378469

[37] Shweta Shinde, Dat Le, Shruti Tople, and Prateek Saxena. 2017. Panoply: Low-TCB Linux Applications with SGX Enclaves. https://doi.org/10.14722/ndss.2017.23500

[38] SPDK. 2023. Intel Storage Performance Development Kit. http://www.spdk.io.

[39] Jörg Thalheim, Harshavardhan Unnibhavi, Christian Priebe, Pramod Bhatotia, and Peter Pietzuch. 2021. Rkt-Io: A Direct I/O Stack for Shielded Execution. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 490–506. https://doi.org/10.1145/3447786.3456255

[40] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. 2018. ShieldBox: Secure Middleboxes Using Shielded Execution. In *Proceedings of the Symposium on SDN Research* (Los Angeles, CA, USA) *(SOSR '18)*. Association for Computing Machinery, New York, NY, USA, Article 2, 14 pages. https://doi.org/10.1145/3185467.3185469

[41] UnixBench. 2023. The original BYTE UNIX benchmark suite. https://github.com/kdlucas/byte-unixbench/tree/master.

[42] Ofir Weisse, Valeria Bertacco, and Todd Austin. 2017. Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 81–93. https://doi.org/10.1145/3079856.3080208

[43] xdp tools. 2023. Library and utilities for use with XDP. https://github.com/xdp-project/xdp-tools.

[44] xdp tools. 2023. xsk_prod_nb_free in xdp-tools/headers/xdp/xsk.h at libxdp. https://github.com/xdp-project/xdp-tools/blob/a7b0903ca29b5e58090a3cbb118f59446055b367/headers/xdp/xsk.h#L92.

[45] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 283–298. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zheng

```
1  int io_uring_queue_init(unsigned entries, struct io_uring *ring,
2      unsigned flags);
3
4  struct io_uring_sqe* io_uring_get_sqe(struct io_uring *ring) {
5    struct io_uring_sq *sq = &ring->sq;
6
7    // pointer dereference potentially pointing inside enclave
8    unsigned int head = IO_URING_READ_ONCE(*sq->khead);
9    unsigned int next = sq->sqe_tail + 1;
10
11   if (next - head <= sq->ring_entries) {
12     struct io_uring_sqe *sqe;
13
14     sqe = &sq->sqes[sq->sqe_tail & sq->ring_mask];
15     sq->sqe_tail = next;
16     return sqe;
17   }
18
19   return NULL;
20 }
```

Figure 6. Code in liburing where a malicious host OS can commence data exfiltration from enclave memory

A Case study: liburing with untrusted OS

In Figure 6, within the function `io_uring_queue_init` defined in Line 1, `liburing` initializes an `io_uring` context without verifying host OS provided values. This absence of validation means that the host OS can provide deceptive or malicious values during setup, potentially undermining the system's integrity. Later, within the function `io_uring_get_sqe` that is exported to be used by `liburing` users, there's a direct dereference of the pointer `sq->khead` (Line 8), which is provided by the host OS during initialization of the `io_uring` context. If `liburing` were running inside SGX, and the `sq->khead` pointer referenced enclave memory, an untrusted host OS can exploit the trust of `liburing` to read sensitive enclave data. The vulnerability becomes evident in Line 11: By observing the difference between the producer and consumer indices (`next` and `sq->khead`), combined with the knowledge of `ring_entries` which holds the ring's size, the host OS can effectively deduce the specific content pointed to by `sq->khead` by observing if the enclave program produces an IO request as a result of returning an `sqe` in Line 16. This deduction leverages the predictable behavior and structure of the shared ring, allowing the host OS to indirectly access enclave data based on observed ring patterns and values.

B Artifact Appendix

B.1 Abstract

RAKIS integrates fast IO primitives within SGX enclaves to reduce enclave exits and significantly boost IO performance. Our prototype implements two Linux kernel IO primitives for use within SGX enclaves: AF_XDP [19] and io_uring [16]. We demonstrate RAKIS's performance benefits through evaluations on four real-world applications and two benchmarking tools.

B.2 Description & Requirements

B.2.1 How to access. RAKIS is publicly available on Github (https://github.com/sslab-gatech/RAKIS) and Zenodo (https://zenodo.org/records/13800030).

B.2.2 Hardware dependencies.

- Intel CPU with SGX [15] support.
- Two free Ethernet interfaces wired in loopback configuration.

B.2.3 Software dependencies. As RAKIS is a fork of Gramine [6], it shares the same software dependencies. For details, refer to https://gramine.readthedocs.io/en/stable/devel/building.html.

B.2.4 Benchmarks. In our experiments, we used two benchmarking tools: iperf3 [17] and fstime [41], along with four real-world programs: Memcached [29], Curl [8], Redis [35], and MCrypt [28].

B.3 Setup

We provided the exact commands in our repository's `README.md` file to set up the environment and reproduce our results. These commands configure the network between the two Ethernet interfaces, place them in separate network namespaces, and set up the NIC queues where XDP programs will be attached.

B.4 Evaluation workflow

We use `Makefiles` to automate the reproduction of our paper's results, with targets prefixed by `eurosys-reproduce` to initiate each experiment. The repository also contains a hierarchical structure of `README.md` files, starting with a top-level file that includes a section titled "Eurosys artifact reviewers", which provides guidance on the reproduction process and details for navigating the artifacts.

B.4.1 Major Claims.

- (C1): RAKIS utilizes XDP IO primitive to achieve an average 11% increase in UDP throughput in iperf3 benchmark while running in SGX enclave compared to native execution. This is proven by experiment (E1) whose results are illustrated in Figure 4(a).
- (C2): In the Curl benchmark using the QUIC protocol, RAKIS eliminates the enclave-exit overhead and exhibits comparable download times to native execution. This is proven by experiment (E2) whose results are illustrated in Figure 4(b).
- (C3): RAKIS efficiently handles multi-threaded workloads and consistently matches native execution performance on the Memcached benchmark. This is proven by experiment (E3) whose results are illustrated in Figure 4(c).

- (C4): RAKIS-SGX showcases a 2.8X performance advantage over Gramine-SGX in the fstime write benchmark as proven by experiment (E4) whose results are illustrated in Figure 5(a).
- (C5): RAKIS-SGX showcases a 2.6X performance advantage over Gramine-SGX in the Redis benchmark as proven by experiment (E5) whose results are illustrated in Figure 5(b).
- (C6): RAKIS-SGX demonstrates a 10% reduction in execution time compared to Gramine-SGX in the MCrypt benchmark as proven by experiment (E6) whose results are illustrated in Figure 5(c).

B.4.2 Experiments. For brevity, we refer to the `README.md` files in the RAKIS code repository for detailed explanations of the experiments. All necessary commands to run the experiments are automated via Makefile scripts. The specific experiments can be found in the following directories:

- (E1): `CI-Examples/iperf3/`.
- (E2): `CI-Examples/curl/`.
- (E3): `CI-Examples/memcached/`.
- (E4): `CI-Examples/unix-benchmark/`.
- (E5): `CI-Examples/redis/`.
- (E6): `CI-Examples/mcrypt/`.