Ringmaster: How to juggle high-throughput host OS system calls from TrustZone TEEs

Richard Habeeb
Yale University

Man-Ki Yoon
North Carolina State University

Hao Chen
CertiK

Zhong Shao
Yale University

Abstract

Many safety-critical systems require the timely processing of sensor inputs to avoid potential safety hazards. Additionally, to support useful application features, such systems increasingly have a large, rich operating system (OS) at the cost of potential security bugs. Thus, if a malicious party gains supervisor privileges, they could cause real-world damage by denying service to time-sensitive programs. Many past approaches to this problem completely isolate time-sensitive programs with a hypervisor; however, this prevents the programs from accessing useful OS services. We introduce Ringmaster, a novel framework that enables enclaves or TEEs (Trusted Execution Environments) to asynchronously access rich, but potentially untrusted, OS services via Linux's *io_uring*. When the untrusted OS denies service, enclaves continue to operate on Ringmaster's minimal ARM TrustZone kernel with access to small, critical device drivers. This approach balances the need for secure, time-sensitive processing with the convenience of rich OS services. Additionally, Ringmaster supports large unmodified programs as enclaves, offering lower overhead compared to existing systems. We demonstrate how Ringmaster helps us build a working, highly secure system with minimal engineering. In our experiments with an unmanned aerial vehicle, Ringmaster achieved nearly 1GiB/sec of data into enclaves on a Raspberry Pi4B, 0-3% throughput overhead compared to non-enclave tasks.

CCS Concepts

• **Security and privacy → Trusted computing**; • **Computer systems organization → Real-time operating systems**; **Embedded and cyber-physical systems**; **Availability**.

ACM Reference Format:
Richard Habeeb, Man-Ki Yoon, Hao Chen, and Zhong Shao. 2026. Ringmaster: How to juggle high-throughput host OS system calls from TrustZone TEEs. In *The 24th Annual International Conference on Mobile Systems, Applications and Services (MobiSys '26), June 21–25, 2026, Cambridge, United Kingdom.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3745756.3809240

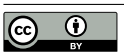
1 Introduction

Recent advancements in computer vision and artificial intelligence (AI) have driven significant progress in autonomous vehicles, drone applications, surgical robotics, and other safety-critical cyber-physical systems (CPS). However, ensuring the safety, reliability, and security of these systems remains a critical challenge [20, 21, 27, 41, 63, 67, 87, 100, 118, 137]. This stems from the need for rich operating systems and complex software stacks to support machine learning (ML) models and features like live video streaming and voice recognition. The Linux kernel, despite its maturity, still contains security vulnerabilities [35], made worse by third-party libraries, drivers, and packages, which expand the attack surface. Additionally, modern CPS are often internet-connected for remote operations and live data streaming [23, 88], increasing exposure to threats. These systems frequently use system-on-a-chip (SoC) designs [22, 60, 105], colocating *time-sensitive* programs with less critical ones, which introduces new security risks [50]. Despite prior research efforts, attacks continue to emerge [18, 19, 24, 36, 42, 71, 89, 102, 109, 117, 133, 134].

Trusted Execution Environments (TEEs) and *enclaves* (in this paper we use these terms interchangeably) have been explored to isolate software from a privileged host operating system [104], but they are not widely used for time-sensitive applications in practice. An ideal CPS enclave might serve as a flight controller or collision-avoidance system, for instance, ensuring *availability* to respond to sensor inputs promptly while communicating with remote operators or logging data to a disk. Running the controller in an enclave would drastically reduce the trusted-computing base (TCB), protecting critical software against privilege-escalation attacks. Focusing on the ARM TrustZone [7], despite many years of quality research on TEEs [97], with numerous strong defense designs for real-time systems [39, 58, 64, 76, 78, 81, 90, 91, 93–96, 106–108, 132]; it is difficult to find real-world examples of them being used to ensure availability in practice. From our observations, TEEs for deployed CPS appear to largely be used for secure boot or cryptography [38].

At a high level, we believe there are at least two major reasons for the general lack of enclave use in practice for time-sensitive applications. First, the most practical designs—those which support untrusted OS system calls, POSIX-compliance, or even unmodified applications—are not designed for availability. Second, the designs that focus on availability can be difficult to use; they lack comprehensive OS services and have a highly limited programming environment. We expand below.

Enclaves with rich OS support face timing vulnerabilities. Existing solutions that support rich OS services [1, 10, 16, 17, 29, 34, 45, 55, 72, 92, 112, 115, 121, 127, 129] completely rely on the untrusted OS for scheduling and memory management, making them unsuitable for time-sensitive applications. For instance, an adversary could delay page fault handling to subtly manipulate the response time of a program, or it could simply kill the process. Furthermore, the traditional, "blocking" POSIX-like programming model for system calls gives the adversary full control over enclave timing. An adversary could delay the return of an `open()`, `write()`, or `read()` system call to affect the response time of some critical behavior.



MobiSys '26, Cambridge, UK

ACM ISBN 979-8-4007-2027-7/2026/06
https://doi.org/10.1145/3745756.3809240

Enclaves with availability protection lack comprehensive OS support. Designs with availability protections [3, 4, 30, 39, 40, 58, 64, 76, 78, 81, 90, 91, 93–96, 106–108, 123, 128, 132] all have very limited OS services. This limits the scope of enclave functionality. By design, ARM TrustZone enclaves [7, 97] only have minimal Secure-world OS services because the trusted computing base (TCB) would have to be inflated to implement a file system or network stack, for example. Additionally, limited system call support forces developers to implement custom data transfer protocols [30, 83, 90, 103, 110, 120]. Thus, small changes to tasks require substantial effort, making this approach unappealing in practice.

We propose ***Ringmaster****,* a new design for making host OS system calls from the TrustZone without sacrificing availability, real-time, or performance needs. Ringmaster leverages the `io_uring` framework [31, 32] for asynchronous system calls, decoupling enclave timing from the system call protocol itself. This approach provides much-needed, practical access to host OS services from TrustZone enclaves (when the OS is well-behaved), and it prevents an enclave from *unnecessarily* blocking if the OS maliciously delays the result. Ringmaster's design unlocks the ability to easily communicate over encrypted network channels from an enclave, to write encrypted data to a disk, or to perform essential inter-process communication (IPC) through pipes and standard I/O—all of which previously required a great degree of manual effort for TrustZone TEEs. This approach further unlocks the ability to run unmodified software in TEEs, allowing the TrustZone to be used as a transparent security layer, whereas existing TrustZone solutions require significant expertise and manually engineered custom "trustlets."

Building availability-oriented TEEs on `io_uring` introduces several challenges not addressed by `io_uring` or existing TrustZone work. First, the `io_uring` memory model assumes a single address space between application and kernel, so Ringmaster must bridge this gap across trust boundaries. Second, the standard liburing library trusts the kernel implicitly, so Ringmaster must handle unsafe, adversarial shared-memory queue operations while maintaining algorithmically bounded execution time. Finally, the `io_uring` API exposed to programmers is dangerous from a DoS perspective; Ringmaster needs to provide an interface layer that makes it clear when blocking can occur.

Since an adversarial OS may choose to never "return" (even for an asynchronous call), time-sensitive I/O is routed through a Ringmaster-owned device via a separate interface. In practice, we observe that such device drivers are often small (*e.g.*, UART, I2C, SPI, or CAN), and we found robustly tested open-source versions available. Ringmaster ensures that enclaves will not starve by preempting and scheduling Linux with a real-time scheduler.

Without Ringmaster, even basic needs, like network I/O, require overly-complex engineering to get data into the TEE apps. Consider the autonomous quadcopter (Fig. 1), that we built to evaluate Ringmaster. Using our new approach, Ringmaster can run a ground-station-facing network server directly in the TEE, which simply uses Linux network sockets. Such a program can be written with minimal development effort. Even if high-level objectives fail to arrive, the enclave can still direct the system's autopilot over serial to ensure safe behavior. With encryption, the system detects tampering of network traffic, ensuring robust operation even under

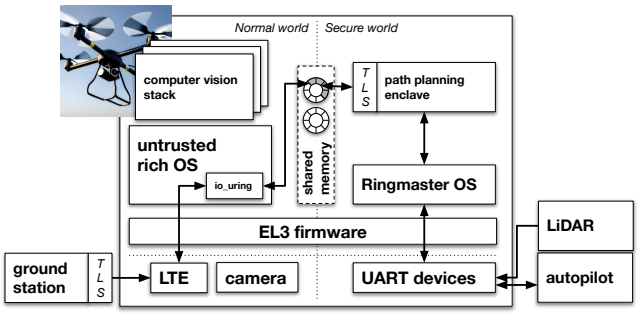


Figure 1: Example of Ringmaster on a drone with a flight controller enclave communicating over encrypted asynchronous `io_uring` operations with a remote operator; if the untrusted OS denies network service, the enclave will continue to stabilize and direct the drone.

adversarial conditions. The enclave can send detailed flight logs and raw sensor data back to the ground-control station with high-throughput, due to the parallelism of `io_uring` and Ringmaster's zero-copy argument passing scheme.

The number of use cases for Ringmaster is potentially quite large, especially for time-sensitive Internet of Things (IoT) or CPS devices. For example, a smart traffic-management system could optimize signal timing with live traffic data coming from the network, but it should still remain operational and timely if incoming data is denied. An AI-enhanced medical device should have clear isolation from timing-critical medical sensing and actuation while also being able to both connect to the cloud and to record sensor data into the file system. Furthermore, our above drone could include a computer-vision Linux program which detects fires or stranded hikers. Even if the application is adversarial, it can only send high-level objectives to the flight controller, greatly limiting the impact of the attack. Because many of these systems have no security against this kind of attack, Ringmaster significantly reduces the TCB for critical tasks.

Legacy POSIX (Portable Operating System Interface) applications can benefit from Ringmaster as well. We built a custom Ringmaster LibC that allows many unmodified programs to run in the TrustZone—which has only been achieved by one other work Using timeouts and signals, such programs can be minimally updated to protect them from being blocked waiting infinitely [45].

Contributions:

- We design a new approach for initiating Linux system calls from TrustZone TEEs with a zero-copy mechanism for transferring large arguments (§4). This includes handling of power and thermal management, mitigating polling impacts (§4.3).
- We present an asynchronous programming model that enables time-sensitive enclaves to securely request potentially untrusted services from the host OS, including an optimized shared memory manager (§5.1).
- We develop a design that supports minimally modified POSIX applications with non-starvation guarantees (§6). Experimental results show comparable or better latency for system calls than past approaches for unmodified enclaves (§9.3).
- We deliver an implementation on the Raspberry Pi4B based drone platform (§8.1), and evaluations using unmodified

GNU Core Utilities (§9.3). We showcase high-throughput TEE I/O, with our experimental configuration fully saturating the Pi's Ethernet port and achieving nearly 1 GiB/s sequential buffered file system read and write speeds.

2 Untrusted System Calls for Time-Sensitive Enclaves?

The intersection of three properties poses significant challenges for many CPS: (1) **time sensitivity**, requiring predictable response times to events (ranging from hard real-time deadlines to softer requirements without worst-case execution time analysis); (2) **distrust of privileged software**, which might be compromised; and (3) **reliance on rich OS services**, such as networking, file systems, pipes, drivers, and third-party software repositories.

While it may seem paradoxical to seek services from an untrusted source, it is common in modern CPS, IoT devices, and robots. For example, drones often use the MAVLink protocol [68] to transmit waypoints from ground stations or companion computers [6, 99] to real-time autopilot systems, like ArduPilot [5], which stabilize the drone. Because radio links are inherently unreliable, drones prioritize immediate sensor data over MAVLink packet timing. Similarly, autonomous vehicles use remote teleoperation in unexpected situations (*e.g.*, construction sites) for high-level guidance [54] while continuing real-time obstacle avoidance. For instance, several recent security analyses reveal that the Tesla Autopilot system is directly connected to an Ethernet switch with internet connectivity [18, 19, 89, 133, 134]. In both cases, time-sensitive systems depend on OS services (*e.g.* from Automotive Grade Linux [122] or Real-Time Ubuntu [25] for instance), such as networking, while maintaining robust real-time performance. Other examples include writing logs to the file system or inter-process communication (IPC) between less critical and more critical tasks.

To meet this need, CPS SoC platforms must enable secure access to rich OS services without fully trusting the OS. Ringmaster addresses this challenge by significantly reducing the trust placed on the OS while maintaining the functionality required for time-sensitive tasks. Our approach enables safe operation even when the OS is compromised. We discuss approaches to handling malicious data in §5.3 and comparisons with other related work in §10.

3 Models & Assumptions

Our models for enclaves, platforms, and adversaries build upon Subramanyan et al.'s definitions [119], extending them to address time-sensitivity and availability.

3.1 Hardware Model

This work targets the ARM TrustZone, but may apply to similar hardware platforms. Particularly, Ringmaster is for platforms that provide privilege levels above kernel mode, a memory-management unit (MMU), and the following requirements:

- **A dedicated timer.** An unmaskable timer interrupt for preemption and scheduling of the OS and enclaves.
- **Memory sharing and isolation.** Programmable control of OS memory access, for example, a TrustZone Address-Space Controller (TZASC) [9], RISC-V's physical-memory protection (PMP), or nested page tables (NPT).
- **Device isolation.** Configurable OS access to memory-mapped I/O (MMIO) and secure interrupt delivery *e.g.* using ARM's Generic Interrupt Controller (GIC) [8].
- **Power, voltage, frequency isolation.** Independent management of system power, clock speed, and voltage.

Intel SGX, as it stands, does not match this model due to its reliance on kernel mode for device and interrupt handling and its lack of programmable enclave page table management. Though our implementation is based on the ARMv8-A TrustZone, a virtualization-based isolation approach, *e.g.* SMACCM [30, 128, 129], or a RISC-V approach, as used by Keystone and ERTOS [70, 123], also fits our model. Future work needs to explore how Ringmaster's implementation details change across these enclave platforms.

3.2 Adversary Model

We define the adversary, *Eve* ($\mathcal{E}$), through two security games addressing real-time guarantees and traditional enclave integrity and confidentiality. In both games, $\mathcal{E}$ has access to kernel-mode privileges and can read or write to any address that is not protected by the mechanisms defined above. $\mathcal{E}$ may schedule any process and configure any interrupt, but she can also be interrupted by the enclave platform. Because the adversary can refuse to service system calls, full availability is not possible; thus, our goal is timeliness: time-bounded response and failure detection.

3.2.1 Game 1: Timeliness. Suppose a victim enclave, *Alice* ($\mathcal{A}$), must write a valid message to a serial device, in a tight time bound t_B. However, the message is stored by $\mathcal{E}$. Upon request, $\mathcal{E}$ chooses to send either a valid or an invalid message. $\mathcal{A}$ has a function $f(m)$ that decides if a message is valid; if it is invalid, she can send a valid *back-up* message. Without any interference or starvation, $\mathcal{A}$ will check and write a valid message in t_C time (even with the maximum number of cache misses, page faults, branch mispredictions, or other microarchitectural delays). In this game, we treat delays as equivalent to a complete DoS. The challenger is an enclave platform, and to win, it must provide a strategy to transfer the message to $\mathcal{A}$ such that: if $\mathcal{E}$ returns a valid message by $t_B - t_C$, $\mathcal{A}$ will write it. Timely but incorrect responses are addressed under Game 2.

3.2.2 Game 2: Integrity & Confidentiality. Suppose our user process from Game 1, $\mathcal{A}$, wants to receive a secret from an external party, *Bob* ($\mathcal{B}$). $\mathcal{E}$ can inspect messages, deny message delivery, modify or delete files, or perform or deny any other kernel action. $\mathcal{E}$ wins the game if she can read or modify communication between $\mathcal{A}$ and $\mathcal{B}$ without their knowledge. $\mathcal{E}$ also wins if she can read or infer secrets from $\mathcal{A}$'s memory.

3.3 Assumptions

Application responses to rich I/O starvation or corruption are out of scope. How enclaves choose to respond to host DoS or corruption varies in each particular context. The power of Ringmaster lies in its ability to enable enclaves to respond to this scenario. Additionally, even without guaranteed availability for OS services, having access is still advantageous (see §2). Malicious I/O can be mitigated via encryption, authentication, or other methods, discussed in §5.3.

Physical and microarchitectural attacks are out of scope. We assume that the adversary does not have physical access to the SoC,

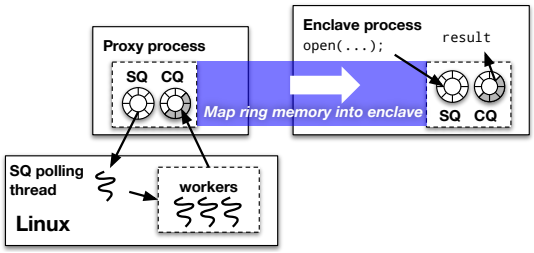


Figure 2: Illustration of how `io_uring` SQ and CQ memory could be mapped into an enclave, giving it access to I/O

and cannot sniff the memory or device buses, perform cold-boot attacks [47], power-analysis attacks [66], *etc.*. Microarchitectural side-channel attacks like ARMageddon, *etc.* [73, 143], are handled using emerging techniques [44, 144]. Additionally, we assume hardware contentions—adversarial cache accesses, page-table walks, branch predictions, memory bandwidth consumption, *etc.*—are being addressed with complementary work [83, 110, 141, 142, 145].

Implementation correctness. We assume that the business logic of $\mathcal{A}$ is written correctly. We trust (and test) that our implementation of Ringmaster is written correctly. Finally, we trust our build environment and compiler.

4 Asynchronous System Call Design

Ringmaster aims to provide I/O system calls into Linux from a TrustZone enclave while preventing Linux from blocking an application during any step of the protocol. Each design choice for Ringmaster is motivated by the need to preserve enclaves' availability—in terms of liveness, non-starvation, and having sufficient resources like memory—as well as confidentiality and integrity of its internal state. We start with `io_uring` as a baseline because it is asynchronous, decoupling system call timing from enclave execution.

Background on `io_uring`. Modern Linux kernels have an interesting system called `io_uring` [31, 32] which allows user-processes to make asynchronous I/O-related system calls. Processes make system calls by adding an entry (SQE) to the *submission queue* (SQ), and receive the system call's return by polling the *completion queue* (CQ) for an entry (CQE). A key option of `io_uring` is a kernel SQ-polling thread which checks for incoming SQEs from the process and sends the SQE work requests to a pool of worker kernel threads—such a polling thread allows for a design where system calls can be made without any traps in or out of the kernel.

4.1 Making System Calls from an Enclave

For Ringmaster, `io_uring` provides an exciting opportunity to change the hierarchical relationship between process and Linux kernel. For a process that is enclaved in the ARM TrustZone, if it could get access to SQ and CQ ring memory and shared I/O memory, then it could request I/O "system calls" from Linux without needing to be directly managed by Linux. If the SQ and CQ are mapped into a TrustZone process, as shown in Fig. 2, then it could perform arbitrary I/O system calls *as if Linux were a remote file system and I/O service.* This decouples non-essential I/O services from core runtime services.

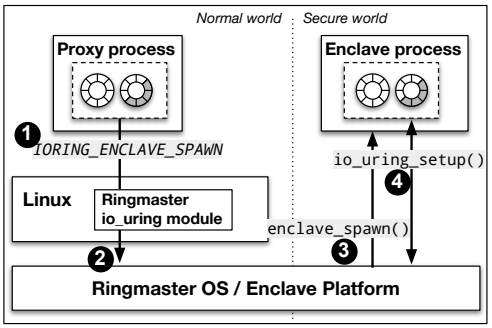


Figure 3: Diagram of Ringmaster's process for registering and mapping `io_uring` memory for an enclave

For Ringmaster, enclaves are processes running in the ARM TrustZone Secure world or in an ARM Confidential Compute Architecture (CCA) Realm, *etc.*; they are managed by a trusted OS (or TEE OS), which handles scheduling, virtual memory, and other essential OS tasks. OP-TEE is a popular open-source option, but we base our design on PARTEE [46], a real-time trusted OS. We will call this updated TEE OS "Ringmaster OS" in this paper. As we will discuss further in §8.2, this division of management ensures enclaves' liveness and non-starvation despite an untrusted Normal world Linux OS. Ringmaster gives enclaves access to I/O system calls by mapping Normal-world `io_uring` SQ and CQ memory into an enclave; thus, enclave operations on the queues are visible by the SQ-polling kernel thread. All operations on the queue are non-blocking and lock-free, which lets enclave execution continue regardless of the state of the queues. While Linux could refuse to service system calls, it cannot block the execution of a well-designed enclave.

The safe mapping of `io_uring`'s queues occurs through the following steps, shown in Fig. 3:

(1) Each enclave has a proxy running as a Linux process; the proxy creates `io_uring` queues, and then adds a `IORING_OP_ENCLAVE_SPAWN` entry to the SQ.
(2) The Ringmaster Linux kernel module implements this new SQE type.[1] Once the SQE is received, Linux registers the queues' physical memory address with Ringmaster OS. Additionally, Linux optionally notifies Ringmaster OS to spawn a new enclave at this time, if it has not been started already.
(3) If the enclave is not running, Ringmaster OS spawns the enclave to be associated with the proxy by actually loading the process ELF binary into memory along with the arguments and environment variables. At this point Ringmaster should optionally authenticate and attest the binary.
(4) The enclave checks for any registered `io_uring` memory (via a synchronous system call serviced by Ringmaster OS), and then Ringmaster OS maps the SQ and CQ rings into the enclave's address space.

Once the list of physical pages is registered, the binding is immutable to Linux. This is because Linux cannot modify the enclave's page tables, which are exclusively stored in Secure-world memory, and managed by Ringmaster OS. To make the registration interface completely secure, Ringmaster OS tracks all physical pages

[1] Neither the Linux kernel module nor the proxy are part of the TCB.

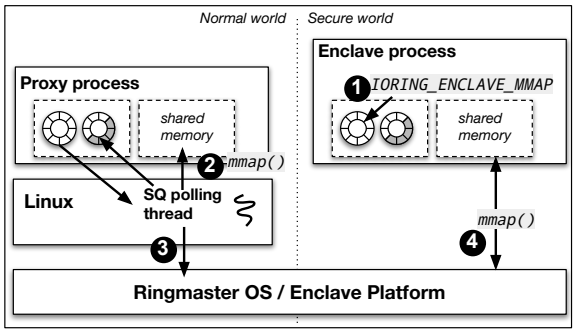


Figure 4: Diagram of Ringmaster's process for registering and mapping generic shared memory for an enclave

and rejects registrations if (1) physical pages are not unique across all TEEs, or (2) a page is not Normal-world physical memory. If Linux decides to reuse the page, then the ring memory would be corrupted, but we already assume Linux can corrupt the rings—see §5.2 for how we detect and mitigate tampering.

4.2 Shared Memory for Arguments

Most system calls do not take simple value arguments, *e.g.* in C/C++ `open()` requires a pointer to a file path string. However, because Ringmaster manages an enclave's address space, not Linux, then a pointer in an SQE to the enclave's memory will not translate when read by Linux. Moreover, dynamically mapping shared memory is itself liable to DoS (if the host refuses to provide shared pages). Thus, we design an asynchronous two-part mechanism for establishing translatable regions of shared memory between proxy and enclave.

The core idea is to use `io_uring` itself to request shared memory. The first part of the mechanism is establishing shared memory between the enclave and proxy (shown in Fig. 4):

(1) The enclave first adds a new `IORING_OP_ENCLAVE_MMAP` entry to the SQ, specifying the request size with an identifier.
(2) The Ringmaster module in Linux handles this new operation, which first allocates a set of pages and then performs an `mmap` of the requested size in the proxy.
(3) Linux then registers with Ringmaster OS with the set of pages, and the identifier. At this point, it is finished, and it puts a return in the CQ; notably, the return value is the virtual address of the memory mapped into the *proxy's address space*.
(4) Once the CQE is received, the enclave performs a `mmap` system call to Ringmaster OS which then maps the registered shared memory into the enclave. This ensures that Linux cannot over-consume resources with spoofed mappings.

The second mechanism for passing pointers in SQEs is a transparent pointer translation process. Ringmaster provides a set of user-space library functions to initialize each SQE type with the correct system call arguments. When an SQE is initialized, or "prepped," with the arguments, Ringmaster will silently translate any pointer arguments that lie within the set of previously mapped shared memory blocks. To do this, Ringmaster's user-space library maintains a lookup table of shared memory blocks, with the enclave's address of the memory, the proxy's address, and the block size. For more complex data structures, *e.g.* `writev` passes a pointer to an array of pointers, a deep translation can be done of each pointer in the list.

With these mechanisms, an enclave already has all it needs to perform `io_uring` operations through Ringmaster. To give the example of an `open` system call:

(1) First, acquire shared memory via an `IORING_OP_ENCLAVE_MMAP` operation (or reuse old shared memory).
(2) Then, copy the file path into the shared memory.
(3) Next, enqueue an SQE with the path argument pointing to the shared memory (internally Ringmaster translates the pointer), and submit the SQE.
(4) The Linux SQ-polling thread sees the SQE in the proxy's queue, and will service the `open` system call (adding a file descriptor to the proxy process) and the thread puts the descriptor in the CQ as the return value.
(5) Now, the enclave has opened the file and can use the descriptor number for further operations.

4.3 Power Management & Wake-Up Signals

Ringmaster needs to deal with two technical challenges involving power management for its design to work. Continuous polling of `io_uring` queues consumes too much power (a problem for battery-powered robotics); this is why, by default, Linux puts SQ-polling kernel threads to sleep after a period of inactivity.

This leads to the first challenge: when the SQ-polling thread sleeps, the enclave's ability to perform `io_uring` operations is severed. The only way to awaken the thread is for the program to execute the `io_uring_enter()` system call, but the program cannot make this call from the TrustZone. Our solution is this:

(1) the enclave sees that the SQ-polling kernel thread is asleep (indicated by a flag in the SQ) and calls `io_uring_enter()` to wake it, *which is actually implemented by Ringmaster OS.*
(2) Ringmaster OS wakes the Linux SQ-polling thread by making a software-generated interrupt (or inter-processor interrupt, IPI), and by implementing the respective handler in a Linux kernel module, then returning to the enclave.
(3) The IPI handler then runs Linux's `io_uring_enter()` on behalf of the enclave, waking the polling thread.

The motivation behind this IPI-based design is to allow completely asynchronous system call "forwarding" between worlds. Synchronous forwarding, which replays the system call exception in the Normal-world (by essentially jumping to the trap handler), as described by TrustShadow and Overshadow [29, 45], does not work for this context. In this approach, Ringmaster would instead switch to Linux, jumping to its system call handler at step 2; however, the system call would appear to be coming from whatever Linux process happened to be running, not the proxy. Thus, IPIs allow decoupled execution.

The second challenge is that enclaves themselves also need to continuously poll for data. On the enclave side, we do not implement wake-up notifications; however, by design, periodic tasks in a real-time scheduler go hand-in-hand with polling-style I/O. As we describe later in §A.2, Ringmaster uses a real-time scheduler and enclaves can be configured with periods and budgets. Thus, the pattern for proper power management would be, once the enclave wakes each period: poll while there are unread `io_uring`

completions, perform work, create submissions, and then yield the remaining budget (*e.g.* see Fig. 10). This common real-time pattern also allows unused budget to be used to either sleep or run Linux.

5 Ringmaster Enclave API

The `io_uring` user-space library, `liburing` [12], implicitly trusts the OS, so we redesigned a new library that considers an adversarial OS and potential programmer security pitfalls. Practically, this means Ringmaster must handle adversarial corruption of queues while maintaining deterministic runtime, *i.e.* we assume the contents of shared memory are untrusted and can change at any point. Additionally, the programmer must ideally have limited access to volatile shared memory, and when they do, it should be clear. In future work, static analysis could ensure at compile time that shared memory operations are memory safe and will not lead to infinite loops. Finally, we do not use any unbounded loops or locks in functions that involve interfacing with ring memory, and we always bound the head and tail indices stored in shared memory. The next sections describe in more detail our designs.

5.1 Dynamic Shared Memory Arenas

Because most `io_uring` operations require pointers, Ringmaster enclaves require reuse and management of shared memory for security, performance, and ease of programming. While a generic memory management software pattern (*e.g.* `malloc` / `free`) would work to help allocate and manage shared memory, we observe that shared memory allocation and freeing will likely follow the pattern of: allocate memory for one or more objects, make one or more `io_uring` system calls, then free the memory. Additionally, many system calls have defined sequences, *e.g.* `open`, `read`, `close`, or `socket`, `bind`, `listen`, `accept`; in these cases shared memory allocations will not be randomly interleaved, so a heavy-weight `malloc` for shared memory is not always needed. Furthermore, requesting, reading, and writing shared memory all present surface areas for adversarial timing attacks. To support a zero- or low-copy interface, programmers need to be made aware of which operations and pointers are volatile and untrusted. If no shared memory is available, they need to have a way to chain asynchronous requests.

With these patterns in mind, we chose an arena-based design [49] for managing shared memory in a performant and secure way. Ringmaster arenas are blocks of shared memory which can be used to quickly allocate and free multiple objects with a similar life cycle, throughout multiple I/O operations. Unlike malloc, programmers primarily manage explicit handles to shared memory (rather than pointers), making clear what memory is shared. No shared-memory allocation metadata (lengths, offsets, free-list pointers) should be stored in shared memory itself, so having explicit handles allows for both secure and fast metadata look-ups (see §A.4 for details). We illustrate in Fig. 5 how multiple arenas can be allocated from shared memory and used to store objects associated with submissions.

In Ringmaster's arena-based design, an enclave requests a shared-memory arena of a certain size. Ringmaster's user-space library immediately fulfills the request if shared memory is available. If no memory block is large enough, the library asynchronously fills its pool of shared memory with a `IORING_OP_ENCLAVE_MMAP` request. When the CQE arrives from Linux, it fulfills the original request;

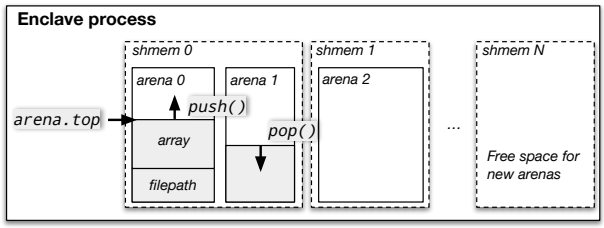


Figure 5: Diagram illustrating how Ringmaster shared memory arenas contain objects with similar life cycles

we provide a promise-based interface to make asynchronous programming possible here, see §5.4.

The power of an arena is through fast allocation and freeing of objects within the arena's memory. The constraint is that objects must be freed from an arena in *first-in-last-out*, stack order. A push is a constant-time operation that allocates memory by incrementing the *top* of the arena's stack by the number of bytes in the allocation. A pop decrements the top of the stack by *N* bytes. Typically, objects do not even need to be popped in many cases, as they are freed when the arena is freed. In this way, small objects, *e.g.* file paths, socket address structures, *etc.*, can be quickly managed. See §A.5 for further optimization details.

5.2 Protections via Ring Abstraction

Since the adversarial host OS can corrupt shared memory (and since host OS service availability is not guaranteed), Ringmaster must handle real-time unsafe queue operations gracefully while mitigating shared memory protocol attacks. We do this by instrumenting user-space API abstractions around the queue to ensure bounded-time failures. The goal is for the API to either return an error or *some* data without blocking; however, the user process needs to actually validate, sanitize, encrypt, and authenticate the I/O data itself using domain-specific protocols (see §5.3).

5.2.1 No direct access to rings: Detecting ring corruption. We do not provide programmer direct access to SQ and CQ ring memory. For example, the `ringmaster_try_get_sqe()` function, which is similar to `io_uring_get_sqe()`, does not return a pointer to the reserved SQE, rather an integer index to the slot in the SQ. This allows the library to always bounds check unsafe pointer de-references and forces Linux to adhere strictly to the shared memory protocol. For instance, we always confirm shared head and tail indices are sane and expected, and return actionable error codes.

5.2.2 One in, one out: Mitigating replay attacks and adversarial queue pressure. In an `io_uring` SQE, the `user_data` field allows the programmer to associate an identifier or data with an operation. It is a simple 64-bit field passed into the SQE and returned in the CQE. We observed that this field could be the source of many exploits if the OS manipulates it (*e.g.* completion replay attacks or spurious completion attacks). We eliminate these attacks by enforcing a 1-1 submission-to-completion pairing; this is done by abstracting the true `user_data` field and providing the programmer with a safe version. We store the programmer's `user_data` in a private table, and use a value internal to the Ringmaster library. This allows Ringmaster to mitigate double-completion attacks by forcing

a monotonically increasing internal field value. With 1-1 completion matching, we mitigate adversarial queue pressure. Even if a programmer polls for completions in a while loop until all are read, the loop is guaranteed to terminate because no new completions will be returned without a corresponding prior submission.

5.2.3 Force bounded loops: Handling protocol DoS. We do not use any unbounded loops or locks in functions that involve interfacing with ring memory, and we always bound the head and tail indices stored in shared memory. Prepping and submitting SQEs involves only straight-line code with only bounded loops and no recursion; the key step is an atomic increment of the tail value and an atomic read of the head value. If there is no room in the SQ, the function returns an error code. Likewise, getting a CQE also involves no loops or recursion, so run times are theoretically deterministic outside of microarchitectural side-effects or page faults.

5.3 Handling Malicious Rich I/O

As demonstrated by SCONE [10], BlackBox [129], TrustShadow [45], Occlum [112], Graphene-SGX [127], and others, encryption and authentication of I/O data can mitigate many Iago attacks [28] relating to the file system and network calls. The remaining source of malicious system call data is from host-owned devices or host user process—these should not be blindly trusted for safety-critical operations. We expect that Ringmaster enclaves will always need to perform some safety checks on such system calls, but this is already becoming common practice to avoid spoofing attacks and robustness issues [33, 56, 61, 77, 113]. Mitigation depends on the context; however, a benefit of Ringmaster is that enclaves are always given the opportunity to vet incoming data. For LiDAR sensors, it may be possible to detect tampering with a watermarking scheme [26, 74, 80]. The Simplex approach has also been explored in detail to validate untrusted ML outputs [51, 75, 79, 86, 111, 130, 139, 140].

5.4 Promises Framework

After using Ringmaster extensively to build different applications, it became clear that it is very difficult for programmers to manage potentially long chains of asynchronous events. Further, C lacks language features that make asynchronous programming easier and safer. Thus, we needed to provide a solution to help programmers manage asynchronous events easily and safely.

Thus, we turned to a well-known programming abstraction (often used in JavaScript for managing RESTful APIs), called *promises* [13, 43, 98]. At a high level, the programmer can simply call `ringmaster_read` or `ringmaster_write` which return promise objects. The promise objects simplify a chain of asynchronous events and callback functor objects (functions with arguments). Inside `ringmaster_write`, it requests a free shared memory, resulting in another promise. It then calls `ringmaster_sqe`, which returns a promise for an SQE; this could happen if the submission queue ring is full because Linux has not processed the SQEs. Then it chains onto that promise with one that waits for pending writes to be completed, and so on. At the end of the function, the `ringmaster_write` function returns the final promise in the chain.

Internally, Ringmaster's promises store a function pointer, next pointer, and a fixed-size array of arguments. Effectively, a promise call chain acts like an asynchronous call stack. The programmer only needs to poll for the promise to be complete (or busy wait), thus vastly simplifying the management of DoS attacks and availability.[2] Promises are allocated dynamically, which means the OS could delay system call returns so that promises build up and exhaust the heap. To prevent memory DoS, we bound the promise call stack with a static configuration variable. As we will see later, the ability to buffer data while waiting on a promise opens up many system call optimization features (§6.2).

6 Ringmaster LibC and LibOS: Legacy Code

We designed a Ringmaster LibC (based on Musl LibC) which can be compiled with unmodified source code to run as an enclave. We modified many LibC functions to perform ring operations via Ringmaster instead of making synchronous I/O system calls. Internally, after enqueuing an SQE (for example, when `read()` is called), the LibC waits for the completion of that system call. For system calls implemented by Ringmaster OS, we can make them synchronously like normal. Table 1 shows how we divided up the calls.[3]

From the application's perspective it can remain unaware of this split, with two exceptions. One, I/O output should be validated and sanitized; many applications likely trust the output of the OS, especially for pseudo files like `/dev/*`. We plan for a future version of Ringmaster to implement a hardware abstraction layer for Gramine [127], a library OS. This would help prevent some validation issues, since I/O can be transparently encrypted. Two, real-time or time-sensitive applications needing to address additional failure modes, discussed in the following section, §6.1.

6.1 Time-Sensitivity with a Synchronous API

Ringmaster allows programmers to make small changes to legacy POSIX applications to prevent OS blocking attacks. Instead of waiting for the completion event, the programmer can define a call-specific timeout for any system call. After the timeout, the LibC sets `errno` to `EAGAIN` or `ETIMEDOUT` to indicate a timeout failure. Additionally, the programmer can register a `SIGALRM` signal, which will interrupt spin loops waiting for completions. When a system call times out, the underlying SQE remains in flight—if the OS eventually completes it, the CQE will be consumed and discarded by the library on the next poll. This means a timed-out write may still take effect, so applications requiring strict ordering should treat a timeout as an ambiguous failure and verify state before retrying.

Alternatively, modifying a program to use non-blocking file descriptors would force it to handle timing-related failure modes.[4] Programmers can set the `O_NONBLOCK` flag for all file descriptors in the critical path; soft real-time Linux programs, like Ardupilot [5], typically already do this. Ringmaster's LibC handles these calls fully asynchronously by prefetching reads and buffering writes internally (only for regular files, FIFOs, and pipes).

[2] In the future, we plan to implement Ringmaster as a user-space Rust package, since Rust handles asynchronous events at the programming language level this is a strong choice for new development.

[3] For unsupported functionality, *e.g.* `fork()`, we plan to implement in the future, and we have a tool to warn when applications contain unsupported calls.

[4] For example, semantically, a non-blocking read on an empty pipe immediately returns.

Service / System Call	OS	Rationale
page-table management, `mmap`, `munmap`, `mprotect`, `mremap`, `sbrk`, `brk`, *etc.*	Ringmaster	Difficult to protect against Iago attacks [28], private data can be leaked through page access patterns [114]
scheduling, `yield()`, `gettimeofday()`	Ringmaster	Necessary for availability
randomness	Ringmaster	Linux-controlled randomness allows cryptographic replay attacks.
signal handling, `sigalarm()`, *etc.*	Ringmaster	Needed for availability, requires fine-grained access to enclave code
critical device access, I2C, UART, SPI, CAN drivers, *etc.*	Ringmaster	Needed for safety- or timing-critical I/O availability
pthreads, semaphores, mutexes, *etc.*	Ringmaster	Required for availability (future work for prototype implementation)
`spawn()`	Ringmaster	Necessary for enclave process-level parallelism
`getpid()`, `getppid()`	Linux / `io_uring`	Useful for getting the proxy's process ID. They are no longer used as entropy sources.
file system operations, `open()`, `read()`, `write()`, `close()`, `unlink()`, `mkdir()`, `statx()`, *etc.*	Linux / `io_uring`	Complex file system drivers are already available
`socket()`, `bind()`, `connect()`, `accept()`, `recv()`, `send()`, *etc.*	Linux / `io_uring`	Complex network drivers and TCP/IP stack drivers are already available in Linux, and most systems assume unreliable networking.
`posix_spawn()`	Linux / `io_uring`	Can be supported with the addition of proposed `IORING_OP_CLONE` and `IORING_OP_EXEC` [126] (Future work for current implementation)
file-backed `mmap()`	Unsupported / Future Work	Requires page-fault handling in Linux, which would break availability
`fork()`	Unsupported / Future work	Requires forking both enclave process and proxy process state, will complicate Ringmaster OS design [15] and inflate TCB

Table 1: Table showing how operating system services are divided for Ringmaster enclaves, using a synchronous trap into Ringmaster OS or asynchronous communication with Linux via `io_uring`

6.2 Asynchronous Optimizations

Similar to previous work, we optimize Ringmaster LibC by buffering in shared memory and making asynchronous writes and reads [124]. Implementing this optimization requires care not to violate POSIX standards or application behavior expectations when working with regular files, pipes, FIFOs, sockets, devices, etc (Ringmaster disables optimizations for pseudo files like `/dev/*`, *e.g.*).

We use our promise library's `ringmaster_write`, to buffer all incoming writes in shared memory. For regular files, it is safest to have only one ongoing write system call at a time to avoid issues with failures; however, we find that we can optimally buffer up to half the size of the system's last-level cache. Furthermore, we write and read in file-system optimal block sizes (determined through `stat`). We observe that promises and buffering increase the latency of a single write; however, when "pipelining" writes, the enclave actually runs faster than baseline, see Fig. 9.

In the future, we plan to optimize other calls as well, when the user makes a `listen()` call, we can automatically also enqueue a *multi-shot* accept `io_uring` operation. Internally, we queue incoming accepts as they arrive. The result is that we can have some parallelization and also reduce the number of `accept` operations a program would need to send to the Linux kernel.

7 Details on Ringmaster OS & Linux

Here we explain the details of Ringmaster OS to clarify how Ringmaster meets its security claims (see A.2, A.3 for background).

7.1 Secure System Calls & Devices

Similar to early enclave work on Proxos [121], the system call interface is split (some calls go asynchronously into Linux and some go normally into Ringmaster OS). However, for Ringmaster, the split division ensures not only confidentiality, and integrity, but also availability. The full table of how OS services are divided for Ringmaster is given in Table 1.

7.2 Changes To Linux

The set of default operations provided by `io_uring` is incomplete for many programs that only use `io_uring` for I/O; for example, `bind()` and `listen()` are not yet available in many kernel versions. For this reason, we added ten `io_uring` operations in the Linux kernel. We found that converting a system call to an `io_uring` operation can be relatively easy (we added the `sync()` operation in less than 10 minutes of engineering time). In future work, we plan to make the set of operations dynamically extensible via kernel module; this way Ringmaster is not tied to specific kernel versions.

8 Security Analysis

In this section, we first perform an analysis of Ringmaster based on Game 1 and 2 defined in §3.2. Then, we evaluate a drone application, discussed in §2, with Ringmaster security.

8.1 Case study: A Highly-Secure Drone

We implemented the drone shown in Fig. 1. We constructed a quadcopter platform with a CubePilot Orange autopilot device and a Raspberry Pi4B companion computer running a Linux 6.5 Buildroot OS.[5] The companion receives high-level objectives from a ground-control station via a radio link. On the companion computer, a path is continually planned between the objectives and sent to the autopilot via MAVLink over serial [68]. Additionally, a LiDAR device is used by the companion computer to collect data and avoid obstacles. The path planner adjusts the plan based on these readings. An improved design could also use a camera and computer vision stack on Linux to identify additional high-level objectives and stream video, but this was not necessary for our prototype.

The MAVLink serial channel is safety-critical, as the autopilot can be easily manipulated. We tested and confirmed that without any other measures, any companion computer process can arbitrarily crash and control the drone by sending forced disarm or manual control commands (see Fig. 6).

[5]The Pi4B does not have bus security needed for the TrustZone (*e.g.* no TZASC), so nested page tables must be used for memory isolation.

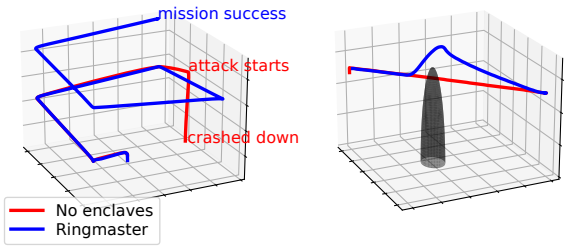


Figure 6: Left: Flight paths of the same mission, showing how Ringmaster protects the safety-critical MAVLink device from being attacked by the compromised host OS, prevent arbitrary crash; Right: Flight paths from the obstacle avoidance experiment, showing how an adversarial OS cannot affect the enclave's path-planner, so it avoids a near-miss scenario.

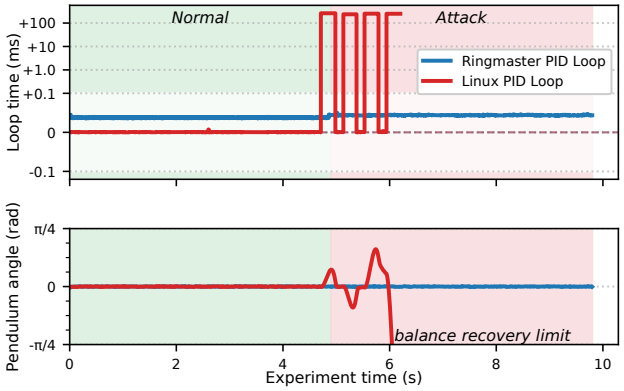


Figure 7: The inverted pendulum remains balanced under attack when PID control runs in Ringmaster enclave.

While the autopilot handles real-time stabilization, the companion computer is also highly time-sensitive because it must detect and avoid obstacles. Thus, to validate the security of Ringmaster, we implement the path-planner program so it can compile to run on Ringmaster or as a regular Linux process. The planner uses a TLS-encrypted connection with the ground station to ensure that objectives are private and not corrupted by the OS. Since the isolation of memory, device MMIO, and IRQ security are well-understood hardware security features, we primarily focus our experimental analysis on the exposed Ringmaster interface.

8.1.1 Obstacle-avoidance experiment: We performed an experiment where the path-planner enclave must read LiDAR data to avoid an object collision (in this case, just a "near-miss"). The object is not visible to the sensor at first, but once it appears in the flight path, the planner should reroute; thus, the LiDAR sensor is safety-critical and will be owned by Ringmaster. We tested two versions of this experiment: (1) no enclaves, *i.e.* no Ringmaster protection, and (2) with Ringmaster protections; results are shown on the right in Fig. 6. For the former, we confirmed that if the adversary killed the planner process after the mission started, the drone would fail the experiment. For the latter, we were unable to make the drone "hit" the obstacle even with killing the proxy, draining system resources, or crashing Linux entirely.

8.2 Detailed Analysis

To assess Ringmaster, we analyze our implementation with an additional experiment and the games from our adversary model.

8.2.1 Inverted Pendulum: Control-Loop Timing. We set up an experiment on the Pi4B to evaluate how well a control loop runs in an enclave when under attack from Linux. We designed an inverted pendulum experiment: to remain upright, it requires active balancing using a Proportional-Integral-Derivative (PID) control loop [11]. We run the same PID loop code in two trials: first as a regular Linux process, then as an enclave. At a high level, the loop reads sensors, calculates motor outputs, and logs information to standard output. We configured a 100Hz update rate; if it delays too long between loops, the pendulum will not recover its balance. During each loop (shown in detail in the appendix Fig. 11), the code reads the rotation angle of the arm, and calculates the target motor acceleration.[6] Then the motors are updated, and a log is output with a `write()` system call, which for Ringmaster is implemented as an async call. During the experiment, after 5 seconds, the root "adversary" throttles standard output. Under normal operation, the `write()` call blocks, delaying the control loop for over 100ms; however, when run as an enclave, the write never blocks the control loop's operation, so the bot stays balanced. We continued heavily loading and crashing Linux, but we were unable to make the pendulum fall. This experiment suggests that Ringmaster TEEs can provide security and I/O functionality for critical control code.

8.2.2 Game 1: Timeliness. In general, how does Ringmaster stand up to the adversary model from Game 1? Both the enclave and Linux will have a period and budget in Ringmaster's scheduler, so Linux will be preempted periodically after reaching its budget. If it tries to power off the machine or change CPU frequency or voltage, the firmware can be configured to deny access to these hardware features, as shown in RT-TEE [132]. Multicore synchronization locks in Ringmaster OS are MCS locks [85] with bounded wait times [62], and the Ringmaster OS kernel avoids recursive calls and uses only clearly bounded loops.

All of the Ringmaster API functions contain only bounded loops, so their runtime has a theoretical upper bound. If Linux delays or denies system call service, the enclave can still access the secure device. If Linux corrupts the queue, the enclave will potentially receive junk values as call returns—so authentication or bounds checking should be employed by the program. If Linux continuously fills the completion queue with returns, the Ringmaster library will internally ignore them as each completion must match a submission; hence, the enclave cannot be delayed by a continual stream of completions. The only exception is if multiple completions are expected (*i.e.* multi-shot accept), in which case, the application needs to consider that peek may always return non-null. If Linux kills the proxy or even completely crashes, it will not affect the execution of this event loop as the physical memory registered to the enclave will remain mapped into its page tables.

[6] For the enclave trial, the motor and sensor drivers are provided by Ringmaster OS instead of Linux.

Can Linux try to abuse the new `IORING_OP_ENCLAVE_MMAP` or `IORING_OP_ENCLAVE_SPAWN` calls to DoS an enclave? Since Linux can attempt SMC calls into Ringmaster OS, it could try to over-consume memory or Ringmaster's data structures, or it could try to consume CPU time by spawning many authentic enclaves repeatedly. Since we based our Ringmaster OS implementation on PARTEE [46], we can ensure that Linux is bounded in its resource consumption. This is because PARTEE partitions its TEE OS data structures to prevent DoS attacks. Additionally, once an enclave is started, PARTEE enforces its guaranteed CPU budget.

8.2.3 Game 2: Confidentiality & Integrity. To try to modify or read an enclave's private state (registers or data), Linux would need to read or infer the non-shared memory pages belonging to the enclave. Linux cannot directly alter or read non-shared enclave physical memory since Ringmaster OS configures the memory bus to disallow reads or writes to the physical address range from the Normal world. Typically, this is a TrustZone Address Space Controller (TZASC) [9] device (or the Granule Protection Table for ARMv9), or other SoC-specific bus-security mechanism (nested paging must be used in the absence of bus security). This also prevents a DMA device from reading/writing enclave memory, as long as the DMA device is subject to the bus security or SMMU rules. Ringmaster OS manages enclave page faults, page tables, and memory allocation, so traditional Iago attacks [28] on `malloc` do not apply. The enclave's registers are stored in TrustZone-only memory on context switch.

In this game, another way a malicious OS can try to alter or read an enclave's private state or communication is indirectly through manipulation of the `io_uring` interface. We already discussed protections for SQ and CQ shared memory in Game 1, but an adversary could still corrupt system call return data. As seen in Table 1, the adversary has control over I/O system calls, so *e.g.*, it could manipulate a file read to send incorrectly formatted data to confuse an application. There are some solutions to these attacks, but, similar to past enclave system call architectures [45, 112, 127], we assume application logic is implemented correctly and defensively, *e.g.* error conditions are always checked and I/O input is parsed securely. The main defense against manipulation and snooping is encryption of files and data, this can be done transparently if Ringmaster is the backend of a Gramine LibOS [127]. For applications that access pseudo files (*e.g.* `/dev/` and `/proc/`), either it needs to be written defensively since the OS can read arbitrary data into these buffers, or unmodified applications must use a LibOS to emulate pseudo files. The adversary could potentially infer private state by observing system call patterns; however, this can be addressed through related work via an "oblivious" file system [2].

The adversary can manipulate the two SMC calls that register `mmap` shared memory or `io_uring` params. For example, it could try to allocate too little shared memory or shared memory that overlaps with another enclave's private data. This is prevented by design: because Ringmaster OS confirms that there is no overlapping in any regions of physical memory used for enclaves, the Ringmaster OS kernel, or device MMIO—this is done by maintaining a physical page allocation table. Ringmaster enforces that shared pages are all unique and that the total size matches the equivalent of the size expected by the enclave (indicated by the `mmap()` arguments).

8.2.4 Reduced TCB.. Currently for ARMv8, Ringmaster OS is about 28k source lines of code (SLoC) in C, and the user-space Ringmaster library is only 1.2k SLoC. Note that our OS also implements `EL3` functionality as well, so no extra firmware is needed. We provide an optional minimal LibC, which is 4.7k SLoC; the full Ringmaster LibC based on Musl (§6) is 93k SLoC. Linux is already nearly 40 million SLoC; thus, for systems with no other protections, Ringmaster already provides a massive reduction in TCB size.

9 Performance Evaluation

To explore Ringmaster's efficacy, we aim to answer the following performance questions: **Q1:** What is the I/O latency overhead of Ringmaster? **Q2:** What is the I/O throughput overhead of Ringmaster? **Q3:** What is the non-I/O overhead of pure computation in an enclave? **Q4:** What is the throughput of Ringmaster when compared with an equivalent `io_uring` application? **Q5:** What is the overhead for unmodified programs using Ringmaster's LibC?

A source of noise in performance experiments for Ringmaster comes from the choice of real-time scheduling parameters and how Linux chooses to schedule the SQ-polling thread. Thus, for these benchmarks, we normalize execution time by dedicating one core for enclave execution and giving the remaining cores 100% utilization for Linux; this way, Linux can freely schedule its worker threads and SQ-polling threads without interference noise in latency measurements. Normally, system designers will have to tune the real-time parameters, core affinities, *etc.* to their specific context.

9.1 Latency Microbenchmarks

We tested Ringmaster's latency via LMBench's system call and I/O benchmarks [84] and compare with related work. Since some works use different hardware, we used normalized LMBench latencies by calculating relative overhead ratios (each system vs its baseline). Our measurements capture the overhead costs of (1) `user_data` protection, (2) arena allocation, (3) shared-memory address translation, (4) and shared-memory copies. We compiled LMBench sources with Ringmaster LibC so that the benchmark could remain as unmodified as possible. We used as many tests as Ringmaster supported, omitting any requiring `fork()` and `exec()`.

Table 2 shows our results. We observe comparable or better overhead for all benchmarks when compared the reported results of all related enclave system-call solutions that report LMBench results: TrustShadow (T.S.) [45], BlackBox (B.B.) [129], Virtual Ghost (V.G.) [34], InkTag (I.T.) [55], and Proxos [121]. Proxos's overhead ratios (from 2006) reflect pre-VT-x virtualization costs; BlackBox, which also uses virtualization, was evaluated on the same Pi4B platform as Ringmaster, and it shows much lower overhead, partly due to hardware-accelerated virtualization. The latency overheads of Ringmaster are minimal because we avoid the synchronous system call overhead and any extra work required in that process by previous efforts (note that for BlackBox some of the overhead is due to container protections as well). However, the extra work required for the above steps, multi-core cache coherence, and `io_uring` overhead latency all add a few hundred nanoseconds to each system call.

Q1: Ringmaster increases latency compared to baseline Linux system calls, but it is comparable to, or better than, similar works.

	Latency (μs)						Overhead					
	Linux + GNU Libc			Ringmaster				T.S.[45]	B.B.[129]	V.G.[34]	I.T.[55]	Proxos[121]
Test	min.	ave.	max.	min.	ave.	max.	ave.	rep.	rep.	rep.	rep.	rep.
null	1.26	1.39	36.15	1.41	2.06	34.07	1.48x	2.01x	2.50x	3.90x	55.80x	12.51x
open	9.93	14.31	934.94	12.67	16.78	721.52	1.17x	1.40x	1.50x	4.93x	7.95x	32.61x
read	1.48	1.63	84.02	2.37	3.02	585.04	1.85x	-	2.10x	-	-	13.06x
write	1.43	1.78	98.54	2.13	2.72	568.57	1.53x	-	2.10x	-	-	12.82x
stat	4.19	4.74	44.02	6.63	12.93	586.37	2.73x	-	2.90x	-	-	17.22x
fstat	3.02	3.52	44.87	5.72	12.41	589.61	3.53x	-	-	-	-	13.71x
mk 0k	24.70	32.00	776.85	30.31	48.43	680.07	1.51x	-	-	4.63x	-	-
rm 0k	15.69	20.19	630.15	22.74	34.67	613.44	1.72x	-	-	4.61x	-	-
mk 1k	45.48	55.24	512.11	59.52	93.04	656.37	1.68x	-	-	5.21x	-	-
rm 1k	29.15	33.24	504.63	16.33	56.74	4115.24	1.71x	-	-	4.52x	-	-
mk 4k	46.04	58.39	584.11	60.78	82.06	434.80	1.41x	-	-	5.19x	-	-
rm 4k	29.13	33.54	580.13	16.41	56.02	405.94	1.67x	-	-	4.52x	-	-
mk 10k	58.57	74.37	658.63	74.02	97.67	700.02	1.31x	-	-	4.71x	-	-
rm 10k	36.09	43.44	646.15	26.89	66.81	403.96	1.54x	-	-	4.71x	-	-

Table 2: Table of LMBench Microbenchmarks, instrumented to run on Ringmaster, overhead is compared with reported results from related enclave research

Test	Linux/io_uring	Ringmaster	Overhead
read file	946.79 MiB/s	936.86 MiB/s	1.011x
write file	826.41 MiB/s	803.58 MiB/s	1.028x
TCP server	111.37 MiB/s	111.38 MiB/s	1.000x

Table 3: Highly parallelized throughput tests for the network and file system, comparing a regular Linux process to a Ringmaster enclave

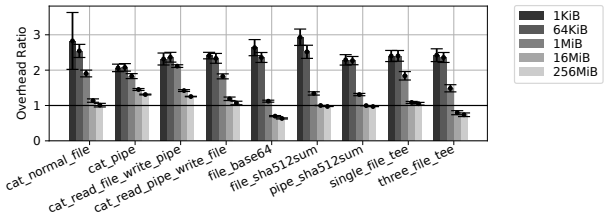


Figure 8: Benchmarks measuring overheads of unmodified GNU Coreutils programs linked against Ringmaster's LibC

9.2 Throughput Comparison with `liburing`

We designed benchmark applications that can be compiled into either a Ringmaster enclave, or a Linux process (using `liburing`). These benchmarks measure throughput for parallelizable workloads and test how the throughput changes when the application is ported to be an enclave. We hypothesized that Ringmaster could achieve nearly zero overhead compared to a regular process because there are no significant differences in the data paths, once enough shared memory has been established. Table 3 shows that, in general, we can receive large volumes of data into a process very quickly—fully saturating the Pi's gigabit Ethernet port, whether the code is running in an enclave or not.

Q2 & Q4: Ringmaster has low observed overhead (0-3%) compared to non-enclave `io_uring` programs.

9.3 Unmodified applications: GNU Coreutils, UnixBench

We tested Ringmaster on complex real-world unmodified software by compiling and linking a large portion (currently 22 programs) of the GNU Coreutils programs with Ringmaster LibC. We chose a few programs that were amenable to more rigorous performance analysis, `cat`, `tee`, `sha512sum`, and `base64`, and experimented with them. We measured their overhead with five different file sizes, normalized against the non-enclave version. While GNU Coreutils are not complex applications, they exercise core I/O paths (file, pipe, stdin/stdout) that are representative of many CPS workloads. More complex applications are possible with threading support, which is planned for future development work.

We observe the higher overheads for smaller files (shorter program runs). This is due to extra startup costs associated with launching enclaves and establishing shared memory (enclaves take about 12ms to start, compared to about 5ms on average for non-enclaves). For all tested programs, we observe that we can match or exceed baseline performance with longer running operations. In fact, the enclave `base64`, `sha512`, and `tee` applications had statistically significant performance improvements (nearly a 50% increase for `base64`), because they can utilize parallelism from our buffered standard I/O implementation.

Q5: Ringmaster has some overhead costs for starting up, but approaches and can exceed baseline performance over time for the Coreutils applications we tested.

We also measured throughput and raw computation overhead using UnixBench compiled with Ringmaster LibC. We normalized Ringmaster's scores against a standard execution of each test in Fig. 9 (lower numbers are better). First, we can see that pure computation has no significant overhead with the `hanoi` test. The `fsbuffer` and `fstime` tests benefit significantly from Ringmaster's internal buffering that allows the system to batch multiple writes and reads asynchronously. The `fsdisk` tests benefit less from buffering because the buffer usually fills well before the disk I/O completes. This is due to the test's design, writing to a file that's larger than Linux's internal file system buffers. The `pipe` test has the highest overhead we measured (2.8x). This test is unable to benefit from any buffering optimizations, as each write must fully complete before the next read. Thus, the overhead is due to the extra shared memory copying, and the Ringmaster promises and `io_uring` architectures.

Q2, Q3, & Q5: Throughput may be reduced for workloads that cannot benefit from I/O parallelism. We measured very low overhead for pure computation.

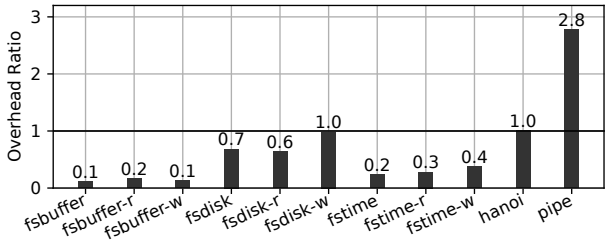


Figure 9: UnixBench Microbenchmarks, comparing Ringmaster's LibC against native Linux (lower is better)

10 Related Work

Asynchronous System Calls In SGX. Solutions such as SCONE [10], Eleos [92], and Occlum [112, 124] leverage asynchronous system calls and shared memory to build SGX-based enclaves, but have three major challenges for time-sensitive applications.

SGX shared memory is incompatible with other architectures: For SGX "shared memory" is any process data memory that is not enclave memory, so allocating shared memory remains a simple `mmap()` or `brk()` system call. When in enclave mode, a program can still directly read and write non-enclave memory, so these solutions were not faced with an additional address-translation challenge (§4.2) that occur outside of SGX-style enclaves. Thus, Ringmaster's innovation is a flexible and dynamic shared memory design *across address spaces and kernels.*

SGX is unable to provide availability: SGX is primarily for Intel (excepting HyperEnclave [57]) which is less common for CPS, and we are not confident that SGX will ever be able to ensure real-time availability because it must rely on the untrusted OS for scheduling and page-table management.

No asynchronous programming model or timeouts: Because these solutions focus on unmodified POSIX applications, they do not expose their asynchronous interface to the user. Additionally, they do not require any innovations in how shared memory is managed or exposed to users. Because of the event-driven nature of many time-sensitive programs, we anticipate that many Ringmaster programs will benefit from its programming model. Even for unmodified modified programs, Ringmaster LibC provides timeouts and signals to ensure availability.

Proxos: Proxos [121] conceptually inspired Ringmaster by splitting the system-call interface between trusted and untrusted VMs. However, Proxos lacks mechanisms to prevent infinite blocking on system calls and did not support asynchronous system calls. Additionally, its fixed-sized shared memory region and virtualization overheads limit suitability for realistic workloads (see Table 2).

In theory, Proxos could be combined with FlexSC [116], SCONE [10], Eleos [92], or Occlum [112, 124]. To the best of our knowledge, such a design has not been discussed or implemented before, so Ringmaster would be the first to explore this. Ringmaster also provides insights needed regarding dynamic shared memory, availability via timeouts and signals, and power management.

Unmodified Applications: When compared with other works supporting unmodified enclave designs [1, 10, 16, 17, 29, 45, 55, 92, 112, 115, 124, 127, 129], Ringmaster's major unique contributions are mechanisms to support availability (though with some modifications). While next-gen Occlum [124] utilizes `io_uring` to reduce SGX switch overheads, it does not address real-time requirements or inter-address-space memory sharing. TrustShadow [45] forwards system calls from TrustZone to Linux but cannot ensure availability due to its dependency on Overshadow-style page table management [29], even if it was extended with a timer interrupt to wake tasks with timed-out system calls.

Partitioning Hypervisors: Besides using a physically separate processor, partitioning hypervisors [82], like Jailhouse [101], PikeOS [59], VxWorks [135], Xen [14, 136], seL4 in SMACCM [30, 53, 65], Bao [83], and the TrustZone-assisted hypervisors [64, 78, 81, 96, 106] represent the dominant isolation technique for CPS. However, they have some challenging trade-offs for real-world systems, preventing their widespread traction. First, they have fixed and static partitions; this rigidity impedes fast-paced development iteration practices for modern physical AI developers. Second, they require specialized RTOS expertise to develop applications, and typical RTOS programming environments do not offer rich I/O services. Third, to communicate between the RTOS and general-purpose VM, they need a VirtIO-like system [90, 110]. This incurs significant engineering overhead to establish data flows, possibly requiring the entire system image to be rebuilt for a single change to a RTOS task. Fourth, virtualization simply for isolation may lead to excessive overheads for some embedded systems [52, 125]. Jailhouse [101] trades the cost of VM exits for the usage of an entire core, often 25-50% of the processing power on SoCs; on the other hand, Ringmaster has more flexibility without losing real-time properties.

Secure I/O for Enclaves: LDR demonstrates how to safely reuse Linux drivers in the TrustZone by dividing their functionality between worlds [138]; if paired with Ringmaster, this is a potential solution to the malicious I/O data problem. Works like StrongBox [37] and Graviton [131] extend enclave isolation to GPUs, and also pair well with Ringmaster. StrongBox, for example, could securely receive encrypted GPU workloads via Ringmaster operations, enabling private AI/ML tasks. MyTEE [48] and RT-TEE [132] protect device access have challenges scaling for broader applicability.

11 Conclusion

Modern safety-critical CPS have conflicting needs: rich I/O to meet consumer demands, and strong timing assurances for safety and reliability. Ringmaster strikes a delicate balance between these two by giving critical software access to rich I/O while not forcing them to rely on it for safety and security. We presented new techniques that allowed enclaves to have access to rich OS features while protecting them from many timing attacks. Our results showed that strong isolation plus an asynchronous approach to rich I/O allowed enclaves to have new timing guarantees which were previously not possible—allowing for much greater security for robotics.

Acknowledgments: We thank our reviewers for their insightful feedback. This work is supported in part by NSF grants 2019285 and 2313433, and by DARPA and NIWC Pacific under Contract No. N66001-21-C-4018. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] Ahmad, A., Kim, J., Seo, J., Shin, I., Fonseca, P., and Lee, B. CHANCEL: efficient multi-client isolation under adversarial programs. In *Network and Distributed Systems Security Symposium* (Virtual, Feb. 2021), NDSS '21, The Internet Society.

[2] Ahmad, A., Kim, K., Sarfaraz, M. I., and Lee, B. OBLIVIATE: A data oblivious filesystem for Intel SGX. In *Network and Distributed Systems Security Symposium* (2018), NDSS '18, The Internet Society.

[3] Alder, F., Van Bulck, J., Piessens, F., and Mühlberg, J. T. Aion: Enabling open systems through strong availability guarantees for enclaves. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea, Nov. 2021), CCS '21, ACM New York, NY, USA, pp. 1357–1372.

[4] Aliaj, E., Nunes, I. D. O., and Tsudik, G. GAROTA: Generalized active Root-Of-Trust architecture (for tiny embedded devices). In *31st USENIX Security Symposium* (Boston, MA, Aug. 2022), USENIX Security '22, USENIX Association, pp. 2243–2260.

[5] ArduPilot. ArduPilot. Available: https://ardupilot.org/.

[6] ArduPilot. NVidia TX2 as a companion computer. Available: https://ardupilot.org/dev/docs/companion-computer-nvidia-tx2.html.

[7] ARM. Building a secure system using TrustZone technology. Tech. rep., ARM, 2009.

[8] ARM. *ARM Generic Interrupt Controller Architecture Version 2.0 — Architecture Specification*, 2013.

[9] ARM. *ARM CoreLink TZC-400 TrustZone Address Space Controller Technical Reference Manual*, 2014.

[10] Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M. L., Goltzsche, D., Eyers, D., Kapitza, R., Pietzuch, P., and Fetzer, C. SCONE: Secure Linux containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation* (Savannah, GA, Nov. 2016), OSDI '16, USENIX Association, pp. 689–703.

[11] Åström, K. J., and Hägglund, T. *PID Controllers: Theory, Design, and Tuning*, 2nd ed. Instrument Society of America, Research Triangle Park, NC, 1995.

[12] Axboe, J. liburing. Available: https://github.com/axboe/liburing.

[13] Baker, H. C., and Hewitt, C. The incremental garbage collection of processes. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages* (Rochester, N.Y., Aug. 1977), AI*PL '77, ACM New York, NY, USA, p. 55–59.

[14] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review 37*, 5 (2003), 164–177.

[15] Baumann, A., Appavoo, J., Krieger, O., and Roscoe, T. A `fork()` in the road. In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy, May 2019), HotOS XVII, pp. 14–22.

[16] Baumann, A., Peinado, M., and Hunt, G. Shielding applications from an untrusted cloud with Haven. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO, Oct. 2014), OSDI '14, USENIX Association, pp. 267–283.

[17] Baumann, A., Peinado, M., and Hunt, G. Shielding applications from an untrusted cloud with Haven. *ACM Transactions on Computer Systems (TOCS) 33*, 3 (2015), 1–26.

[18] Berard, D., and Dehors, V. I feel a draft. opening the doors and windows 0-click RCE on the Tesla Model3. In *Hexacon* (Vancouver, BC, Oct. 2022).

[19] Berard, D., and Dehors, V. 0-click RCE on the Tesla infotainment through cellular network. In *OffensiveCon* (May 2024). Available: https://www.synacktiv.com/sites/default/files/2024-05/tesla_0_click_rce_cellular_network_offensivecon2024.pdf.

[20] Bhamare, D., Zolanvari, M., Erbad, A., Jain, R., Khan, K., and Meskin, N. Cybersecurity for industrial control systems: A survey. *Computers & Security 89*, 101667 (Feb. 2020).

[21] Bonaci, T., Yan, J., Herron, J., Kohno, T., and Chizeck, H. J. Experimental analysis of denial-of-service attacks on teleoperated robotic systems. In *Proceedings of the ACM/IEEE Sixth International Conference on Cyber-Physical Systems* (Seattle, Washington, 2015), ICCPS '15, ACM New York, NY, USA, p. 11–20.

[22] Burns, A., and Davis, R. Mixed criticality systems—a review. Tech. Rep. 13, Department of Computer Science, University of York, Feb. 2022.

[23] Byres, E. The air gap: SCADA's enduring security myth. *Communications of the ACM 56*, 8 (2013), 29–31.

[24] Cai, Z., Wang, A., and Zhang, W. 0-days & mitigations: Roadways to exploit and secure connected BMW cars. In *Black Hat USA 2019* (Las Vegas, NV, 2019).

[25] Canonical. A CTO's guide to real-time Linux. Tech. rep., Canonical, Mar. 2023.

[26] Changalvala, R., and Malik, H. Lidar data integrity verification for autonomous vehicle using 3D data hiding. In *2019 IEEE Symposium Series on Computational Intelligence* (2019), SSCI '19, IEEE, pp. 1219–1225.

[27] Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S., Koscher, K., Czeskis, A., Roesner, F., and Kohno, T. Comprehensive experimental analyses of automotive attack surfaces. In *20th USENIX Security Symposium* (San Francisco, CA, Aug. 2011), USENIX Security '11, USENIX Association.

[28] Checkoway, S., and Shacham, H. 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* (Houston, Texas, USA, Mar. 2013), ASPLOS '13, ACM New York, NY, USA, pp. 253—-264.

[29] Chen, X., Garfinkel, T., Lewis, E. C., Subrahmanyam, P., Waldspurger, C. A., Boneh, D., Dwoskin, J., and Ports, D. R. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. *ACM SIGOPS Operating Systems Review 42*, 2 (Mar. 2008), 2–13.

[30] Cofer, D., Backes, J., Gacek, A., DaCosta, D., Whalen, M., Kuz, I., Klein, G., Heiser, G., Pike, L., Foltzer, A., Podhradsky, M., Stuart, D., Grahan, J., and Wilson, B. Secure mathematically-assured composition of control models. Tech. rep., Air Force Research Laboratory (RITA), Sept. 2017.

[31] Corbet, J. Ringing in a new asynchronous I/O API. Available: https://lwn.net/Articles/776703/, Jan. 2019.

[32] Corbet, J. The rapid growth of io_uring. https://lwn.net/Articles/810414/, Jan. 2020.

[33] Costin, A., and Francillon, A. Ghost in the air (traffic): On insecurity of ADS-B protocol and practical attacks on ADS-B devices. In *Black Hat USA* (Las Vegas, NV, July 2012).

[34] Criswell, J., Dautenhahn, N., and Adve, V. Virtual ghost: Protecting applications from hostile operating systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA, Feb. 2014), ASPLOS '14, ACM New York, NY, USA.

[35] CVEdetails.com. Threat overview for Linux kernel. Available: https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html.

[36] Daan Keuper, T. A. The connected car: Ways to get unauthorized access and potential implications. Tech. rep., Computest, 2018.

[37] Deng, Y., Wang, C., Yu, S., Liu, S., Ning, Z., Leach, K., Li, J., Yan, S., He, Z., Cao, J., and Zhang, F. StrongBox: A GPU TEE on ARM endpoints. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA, Nov. 2022), CCS '22, ACM New York, NY, USA, pp. 769–783.

[38] DJI. Drone security white paper (version 3.0). Tech. rep., DJI, Apr. 2024.

[39] Dong, P., Burns, A., Jiang, Z., and Liao, X. TZDKS: A new TrustZone-based dual-criticality system with balanced performance. In *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications* (Hakodate, Japan, Aug. 2018), RTCSA '18, IEEE, pp. 59–64.

[40] Eldefrawy, K., Rattanavipanon, N., and Tsudik, G. HYDRA: Hybrid design for remote attestation (using a formally verified microkernel). In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks* (Boston, Massachusetts, July 2017), WiSec '17, ACM New York, NY, USA, pp. 99–110.

[41] Fagnant, D. J., and Kockelman, K. Preparing a nation for autonomous vehicles: Opportunities, barriers and policy recommendations. *Transportation Research Part A: Policy and Practice 77* (2015), 167–181.

[42] Falliere, N., Murchu, L. O., and Chien, E. W32.Stuxnet dossier. Tech. rep., Symantec Corp., Feb. 2011.

[43] Friedman, D., and Wise, D. Cons should not evaluate its arguments. Tech. rep., Indiana University, 1976.

[44] Giner, L., Steinegger, S., Purnal, A., Eichlseder, M., Unterluggauer, T., Mangard, S., and Gruss, D. Scatter and split securely: Defeating cache contention and occupancy attacks. In *2023 IEEE Symposium on Security and Privacy* (2023), IEEE S&P '23, IEEE, pp. 2273–2287.

[45] Guan, L., Liu, P., Xing, X., Ge, X., Zhang, S., Yu, M., and Jaeger, T. TrustShadow: Secure execution of unmodified applications with ARM TrustZone. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services* (Niagara Falls, New York, USA, June 2017), MobiSys '17, ACM New York, NY, USA, pp. 488–501.

[46] Habeeb, R., Chen, H., Yoon, M.-K., and Shao, Z. It's a non-stop PARTEE! Practical multi-enclave availability through partitioning and asynchrony. In *2025 Annual Computer Security Applications Conference* (Honolulu, HI, 2025), ACSAC '25.

[47] Halderman, J. A., Schoen, S. D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J. A., Feldman, A. J., Appelbaum, J., and Felten, E. W. Lest we remember: Cold-boot attacks on encryption keys. *Communications of the ACM 52*, 5 (2009), 91–98.

[48] Han, S., and Jang, J. MyTEE: Own the trusted execution environment on embedded devices. In *Network and Distributed Systems Security Symposium* (San Diego, CA, Feb. 2023), NDSS '23, The Internet Society.

[49] Hanson, D. R. Fast allocation and deallocation of memory based on object lifetimes. *Software: Practice and Experience 20*, 1 (1990), 5–12.

[50] Hasan, M., Kashinath, A., Chen, C.-Y., and Mohan, S. Sok: Security in real-time systems. *ACM Computing Surveys 56*, 9 (Apr. 2024).

[51] Hasan, M., and Mohan, S. Protecting actuators in safety-critical IoT systems

from control spoofing attacks. In *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things* (London, United Kingdom, 2019), IoT S&P'19, ACM New York, NY, USA, p. 8–14.

[52] Heiser, G. The role of virtualization in embedded systems. In *Proceedings of the 1st workshop on Isolation and Integration in Embedded Systems* (Glasgow, UK, Apr. 2008), IIES '08, pp. 11–16.

[53] Heiser, G., Parker, L., Chubb, P., Velickovic, I., and Leslie, B. Can we put the "S" into IoT? In *IEEE 8th World Forum on Internet of Things* (2022), WF-IoT '22, IEEE, pp. 1–6.

[54] Herger, M. How teleguidance helps zoox vehicles to navigate difficult traffic scenarios. Available: https://thelastdriverlicenseholder.com/2020/11/12/how-teleguidance-helps-zoox-vehicles-to-navigate-difficult-traffic-scenarios/, Nov. 2020.

[55] Hofmann, O. S., Kim, S., Dunn, A. M., Lee, M. Z., and Witchel, E. InkTag: Secure applications on an untrusted operating system. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA, Mar. 2013), ASPLOS '13, ACM New York, NY, USA, pp. 265–278.

[56] Jia, W., Lu, Z., Zhang, H., Liu, Z., Wang, J., and Qu, G. Fooling the eyes of autonomous vehicles: Robust physical adversarial examples against traffic sign recognition systems. In *Network and Distributed Systems Security Symposium* (San Diego, CA, Apr. 2022), NDSS '22, The Internet Society.

[57] Jia, Y., Liu, S., Wang, W., Chen, Y., Zhai, Z., Yan, S., and He, Z. HyperEnclave: An open and cross-platform trusted execution environment. In *2022 USENIX Annual Technical Conference* (Carlsbad, CA, July 2022), USENIX ATC '22, USENIX Association, pp. 437–454.

[58] Jiang, Z., Dong, P., Wei, R., Zhao, Q., Wang, Y., Zhu, D., Zhuang, Y., and Audsley, N. PSpSys: A time-predictable mixed-criticality system architecture based on ARM TrustZone. *Journal of Systems Architecture 123* (2022).

[59] Kaiser, R., and Wagner, S. Evolution of the pikeos microkernel. In *First International Workshop on MicroKernels for Embedded Systems* (Jan. 2007), MIKES '07.

[60] Kani, A. NVIDIA DRIVE thor strikes AI performance balance, uniting AV and cockpit on a single computer. Available: https://blogs.nvidia.com/blog/2022/09/20/drive-thor/, Sept. 2022.

[61] Kerns, A. J., Shepard, D. P., Bhatti, J. A., and Humphreys, T. E. Unmanned aircraft capture and control via GPS spoofing. *Journal of Field Robotics 31*, 4 (2014), 617–636.

[62] Kim, J., Sjöberg, V., Gu, R., and Shao, Z. Safety and liveness of MCS lock—layer by layer. In *15th Asian Symposium on Programming Languages and Systems* (Suzhou, China, Nov. 2017), APLAS '17, Springer, pp. 273–297.

[63] Kim, K., Kim, J. S., Jeong, S., Park, J.-H., and Kim, H. K. Cybersecurity for autonomous vehicles: Review of attacks and defense. *Computers & Security 103* (2021).

[64] Kim, S. W., Lee, C., Jeon, M., Kwon, H. Y., Lee, H. W., and Yoo, C. Secure device access for automotive software. In *2013 International Conference on Connected Vehicles and Expo* (2013), ICCVE '13, IEEE, pp. 177–181.

[65] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Oct. 2009), SOSP '09, ACM New York, NY, USA, pp. 207–220.

[66] Kocher, P. C., Jaffe, J., and Jun, B. Differential power analysis. In *19th Annual International Cryptology Conference* (Santa Barbara, CA, USA, Aug. 1999), M. J. Wiener, Ed., CRYPTO '99, Springer-Verlag.

[67] Koscher, K., Czeskis, A., Roesner, F., Patel, S., Kohno, T., Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., and Savage, S. Experimental security analysis of a modern automobile. In *2010 IEEE Symposium on Security and Privacy* (Oakland, CA, USA, May 2010), IEEE S&P '10, IEEE, pp. 447–462.

[68] Koubâa, A., Allouch, A., Alajlan, M., Javed, Y., Belghith, A., and Khalgui, M. Micro Air Vehicle Link (MAVLink) in a nutshell: A survey. *IEEE Access 7* (2019), 87658–87680.

[69] Lea, D., and Gloger, W. A memory allocator. Available: https://www.cs.tufts.edu/~nr/cs257/archive/doug-lea/malloc.html, 1996.

[70] Lee, D., Kohlbrenner, D., Shinde, S., Asanović, K., and Song, D. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Apr. 2020), EuroSys '20, ACM New York, NY, USA.

[71] Lee, R. M., Assante, M. J., and Conway, T. Analysis of the cyber attack on the ukrainian power grid. Tech. rep., SANS ICS, E-ISAC, Washington, DC, 2016.

[72] Li, Y., McCune, J., Newsome, J., Perrig, A., Baker, B., and Drewry, W. MiniBox: A two-way sandbox for x86 native code. In *2014 USENIX Annual Technical Conference* (Philadelphia, PA, June 2014), USENIX ATC '14, USENIX Association, pp. 409–420.

[73] Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., and Mangard, S. ARMageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium* (Austin, TX, Aug. 2016), USENIX Security '16, USENIX Association, pp. 549–564.

[74] Liu, J., and Park, J.-M. "seeing is not always believing": Detecting perception error attacks against autonomous vehicles. *IEEE Transactions on Dependable and Secure Computing 18*, 5 (2021), 2209–2223.

[75] Liu, R., and Srivastava, M. PROTC: PROTeCting drone's peripherals through ARM TrustZone. In *Proceedings of the 3rd Workshop on Micro Aerial Vehicle Networks, Systems, and Applications* (Niagara Falls, New York, USA, June 2017), DroNet '17, ACM New York, NY, USA, pp. 1–6.

[76] Liu, Y., An, K., and Tilevich, E. RT-Trust: Automated refactoring for trusted execution under real-time constraints. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Boston, MA, USA, Nov. 2018), GPCE '18, ACM New York, NY, USA, pp. 175–187.

[77] Liu, Z., Miao, Z., Zhan, X., Wang, J., Gong, B., and Yu, S. X. Large-scale long-tailed recognition in an open world. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (June 2019), CVPR '19, Computer Vision Foundation, pp. 2537–2546.

[78] Lucas, P., Chappuis, K., Paolino, M., Dagieu, N., and Raho, D. VOSYSmonitor, a low latency monitor layer for mixed-criticality systems on ARMv8-A. In *29th Euromicro Conference on Real-Time Systems* (Dagstuhl, Germany, 2017), M. Bertogna, Ed., vol. 76 of *ECRTS '17*, Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 6:1–6:18.

[79] Luo, B., Ramakrishna, S., Pettet, A., Kuhn, C., Karsai, G., and Mukhopadhyay, A. Dynamic Simplex: Balancing safety and performance in autonomous cyber physical systems. In *Proceedings of the ACM/IEEE 14th International Conference on Cyber-Physical Systems (with CPS-IoT Week 2023)* (San Antonio, TX, USA, 2023), ICCPS '23, p. 177–186.

[80] Marazzi, M., Longari, S., Michele, C., and Zanero, S. Securing LiDAR communication through watermark-based tampering detection. In *Symposium on Vehicles Security and Privacy* (2024), VehicleSec '24.

[81] Martins, J., Alves, J., Cabral, J., Tavares, A., and Pinto, S. μRTZVisor: A secure and safe real-time hypervisor. *Electronics 6*, 4 (2017).

[82] Martins, J., and Pinto, S. Shedding light on static partitioning hypervisors for ARM-based mixed-criticality systems. In *29th Real-Time and Embedded Technology and Applications Symposium* (2023), RTAS '23, IEEE, pp. 40–53.

[83] Martins, J., Tavares, A., Solieri, M., Bertogna, M., and Pinto, S. Bao: A lightweight static partitioning hypervisor for modern multi-core embedded systems. In *Workshop on Next Generation Real-Time Embedded Systems* (2020), NG-RES '20, Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[84] McVoy, L., and Staelin, C. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference* (San Diego, CA, USA, 1996), USENIX ATC '96, pp. 279–294.

[85] Mellor-Crummey, J. M., and Scott, M. L. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS) 9*, 1 (1991), 21–65.

[86] Musau, P., Hamilton, N., Lopez, D. M., Robinette, P., and Johnson, T. T. On using real-time reachability for the safety assurance of machine learning controllers. In *2022 IEEE International Conference on Assured Autonomy* (2022), ICAA '22, pp. 1–10.

[87] Nassi, B., Bitton, R., Masuoka, R., Shabtai, A., and Elovici, Y. SoK: Security and privacy in the age of commercial drones. In *2021 IEEE Symposium on Security and Privacy* (2021), IEEE S&P '21, pp. 1434–1451.

[88] Nehmy, A. The air gap is dead. it's time for industrial organisations to embrace the cloud. Available: https://www.paloaltonetworks.com/cybersecurity-perspectives/the-air-gap-is-dead.

[89] Nie, S., Liu, L., and Du, Y. Free-fall: Hacking tesla from wireless to CAN bus. In *Black Hat USA 2017* (Las Vegas, NV, 2017).

[90] Oliveira, A., Martins, J., Cabral, J., Tavares, A., and Pinto, S. TZ-VirtIO: Enabling standardized inter-partition communication in a TrustZone-assisted hypervisor. In *2018 IEEE 27th International Symposium on Industrial Electronics* (Cairns, QLD, Australia, June 2018), ISIE '18, IEEE, pp. 708–713.

[91] Oliveira, D., Gomes, T., and Pinto, S. uTango: An open-source TEE for IoT devices. *IEEE Access 10* (2022), 23913–23930.

[92] Orenbach, M., Lifshits, P., Minkin, M., and Silberstein, M. Eleos: ExitLess OS services for SGX enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia, Apr. 2017), EuroSys '17, pp. 238–253.

[93] Pinto, S., Araujo, H., Oliveira, D., Martins, J., and Tavares, A. Virtualization on TrustZone-enabled microcontrollers? voilà! In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium* (Montreal, QC, Canada, Apr. 2019), RTAS '19, IEEE, pp. 293–304.

[94] Pinto, S., Oliveira, A., Pereira, J., Cabral, J., Monteiro, J., and Tavares, A. Lightweight multicore virtualization architecture exploiting ARM TrustZone. In *43rd Annual Conference of the IEEE Industrial Electronics Society* (Beijing, China, Oct. 2017), IECON '17, IEEE, pp. 3562–3567.

[95] Pinto, S., Pereira, J., Gomes, T., Ekpanyapong, M., and Tavares, A. Towards a TrustZone-assisted hypervisor for real-time embedded systems. *IEEE Computer Architecture Letters 16*, 2 (Oct. 2016), 158–161.

[96] Pinto, S., Pereira, J., Gomes, T., Tavares, A., and Cabral, J. LTZVisor: TrustZone is the key. In *29th Euromicro Conference on Real-Time Systems* (Dagstuhl, Germany, 2017), M. Bertogna, Ed., vol. 76 of *ECRTS '17*, Leibniz International

Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 4:1–4:22.

[97] Pinto, S., and Santos, N. Demystifying ARM TrustZone: A comprehensive survey. *ACM Computing Surveys 51*, 6 (2019), 1–36.

[98] Promises/A+ Organization. An open standard for sound, interoperable JavaScript promises—by implementers, for implementers. Available: https://promisesaplus.com/, May 2014.

[99] PX4. PX4 guide (main) — companion computers. Available: https://docs.px4.io/main/en/companion_computer/.

[100] Quarta, D., Pogliani, M., Polino, M., Maggi, F., Zanchettin, A. M., and Zanero, S. An experimental security analysis of an industrial robot controller. In *2017 IEEE Symposium on Security and Privacy* (San Jose, CA, May 2017), IEEE S&P '17, IEEE, pp. 268–286.

[101] Ramsauer, R., Kiszka, J., Lohmann, D., and Mauerer, W. Look mum, no vm exits!(almost). arXiv preprint arXiv:1705.06932, 2017.

[102] Rodday, N. Hacking a professional drone. In *Black Hat Asia 2016* (Singapore, Mar. 2016).

[103] Russell, R. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review 42*, 5 (July 2008), 95–103.

[104] Sabt, M., Achemlal, M., and Bouabdallah, A. Trusted execution environment: What it is, and what it is not. In *Proceedings of the 2015 IEEE Trustcom/BigDataSE/ISPA* (Helsinki, Finland, 2015), Trustcom '15, IEEE, pp. 57–64.

[105] Saidi, S., Ernst, R., Uhrig, S., Theiling, H., and de Dinechin, B. D. The shift to multicores in real-time and safety-critical systems. In *2015 International Conference on Hardware/Software Codesign and System Synthesis* (Amsterdam, Netherlands, Oct. 2015), CODES+ISSS '15, IEEE, pp. 220–229.

[106] Sangorrin, D., Honda, S., and Takada, H. Dual operating system architecture for real-time embedded systems. In *6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications* (July 2010), OSPERT '10.

[107] Sangorrin, D., Honda, S., and Takada, H. Integrated scheduling for a reliable dual-OS monitor. *IPSJ Transactions on Advanced Computing Systems 5*, 2 (2012), 99–110.

[108] Sangorrin, D., Honda, S., and Takada, H. Reliable and efficient dual-OS communications for real-time embedded virtualization. *Computer Software 29*, 4 (2012), 182–198.

[109] Schiller, N., Chlosta, M., Schloegel, M., Bars, N., Eisenhofer, T., Scharnowski, T., Domke, F., Schönherr, L., and Holz, T. Drone security and the mysterious case of DJI's DroneID. In *Network and Distributed Systems Security Symposium* (San Diego, CA, Feb. 2023), NDSS '23, The Internet Society.

[110] Schwäricke, G., Tabish, R., Pellizzoni, R., Mancuso, R., Bastoni, A., Zuepke, A., and Caccamo, M. A real-time virtio-based framework for predictable inter-vm communication. In *2021 IEEE Real-Time Systems Symposium* (2021), RTSS '21, IEEE, pp. 27–40.

[111] Sha, L. Using simplicity to control complexity. *IEEE Software 18*, 4 (2001), 20–28.

[112] Shen, Y., Tian, H., Chen, Y., Chen, K., Wang, R., Xu, Y., Xia, Y., and Yan, S. 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, Mar. 2020), ASPLOS '20, pp. 955–970.

[113] Shin, H., Kim, D., Kwon, Y., and Kim, Y. Illusion and dazzle: Adversarial optical channel exploits against lidars for automotive applications. In *19th International Conference on Cryptographic Hardware and Embedded Systems* (Taipei, Taiwan, Sept. 2017), CHES '17, Springer, pp. 445–467.

[114] Shinde, S., Chua, Z. L., Narayanan, V., and Saxena, P. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security* (Xi'an, China, 2016), ASIA CCS '16, ACM New York, NY, USA, p. 317–328.

[115] Shinde, S., Le Tien, D., Tople, S., and Saxena, P. Panoply: Low-TCB Linux applications with SGX enclaves. In *Network and Distributed Systems Security Symposium* (San Diego, CA, USA, Feb. 2017), NDSS '17, The Internet Society.

[116] Soares, L., and Stumm, M. FlexSC: Flexible system call scheduling with Exception-Less system calls. In *9th USENIX Symposium on Operating Systems Design and Implementation* (Vancouver, BC, Oct. 2010), USENIX Association.

[117] Starks, T., and DiMolfetta, D. Downed U.S. drone points to cyber vulnerabilities. Available: https://www.washingtonpost.com/politics/2023/03/16/downed-us-drone-points-cyber-vulnerabilities/, Mar. 2023.

[118] Stellios, I., Kotzanikolaou, P., Psarakis, M., Alcaraz, C., and Lopez, J. A survey of IoT-enabled cyberattacks: Assessing attack paths to critical infrastructures and services. *IEEE Communications Surveys & Tutorials 20*, 4 (2018), 3453–3495.

[119] Subramanyan, P., Sinha, R., Lebedev, I., Devadas, S., and Seshia, S. A. A formal foundation for secure remote execution of enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), pp. 2435–2450.

[120] SYSGO. PikeOS — VirtIO. Available: https://www.sysgo.com/virtio.

[121] Ta-Min, R., Litty, L., and Lie, D. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, WA, USA, Nov. 2006), OSDI '06, USENIX Association, pp. 279–292.

[122] The Linux Foundation. Automotive Grade Linux. Available: https://www.automotivelinux.org/.

[123] Thomas, A., Kaminsky, S., Lee, D., Song, D., and Asanovic, K. ERTOS: Enclaves in real-time operating systems. In *Fifth Workshop on Computer Architecture Research with RISC-V* (Virtual, June 2021), CARRV '21.

[124] Tian, H. Re-architect Occlum for the next-gen Intel SGX. In *Open Confidential Computing Conference* (2021), OC3 '21.

[125] Toumassian, S., Werner, R., and Sikora, A. Performance measurements for hypervisors on embedded ARM processors. In *2016 International Conference on Advances in Computing, Communications and Informatics* (2016), ICACCI '16, IEEE, pp. 851–855.

[126] Triplett, J. Spawning processes faster and easier with io_uring. In *Linux Plumber's Conference 2022 (LPC Refereed Track)* (2022).

[127] Tsai, C.-C., Porter, D. E., and Vij, M. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *USENIX Annual Technical Conference* (Santa Clara, CA, July 2017), USENIX ATC '17, USENIX Association, pp. 645–658.

[128] UNSW Trustworthy Systems Group. SMACCM: TS in the DARPA HACMS Program. Available: https://trustworthy.systems/projects/OLD/SMACCM/.

[129] Van't Hof, A., and Nieh, J. BlackBox: a container security monitor for protecting containers on untrusted operating systems. In *16th USENIX Symposium on Operating Systems Design and Implementation* (Carlsbad, CA, July 2022), OSDI '22, USENIX Association, pp. 683–700.

[130] Vivekanandan, P., Garcia, G., Yun, H., and Keshmiri, S. A simplex architecture for intelligent and safe unmanned aerial vehicles. In *22nd International Conference on Embedded and Real-Time Computing Systems and Applications* (Daegu, Korea (South), Aug. 2016), RTCSA '16, IEEE, pp. 69–75.

[131] Volos, S., Vaswani, K., and Bruno, R. Graviton: Trusted execution environments on GPUs. In *13th USENIX Symposium on Operating Systems Design and Implementation* (Carlsbad, CA, Oct. 2018), OSDI '18, USENIX Association, pp. 681–696.

[132] Wang, J., Li, A., Li, H., Lu, C., and Zhang, N. RT-TEE: Real-time system availability for cyber-physical systems using ARM TrustZone. In *2022 IEEE Symposium on Security and Privacy* (San Francisco, CA, May 2022), IEEE S&P '22, IEEE, pp. 352–369.

[133] Weinmann, R.-P., and Schmotzle, B. TBONE–a zero-click exploit for Tesla MCUs. Tech. rep., ComSecuris, 2020.

[134] Werling, C., Kühnapfel, N., Jacob, H. N., and Drokin, O. Jailbreaking an electric vehicle in 2023. In *Black Hat USA* (Las Vegas, NV, June 2023).

[135] WindRiver. Vxworks safety planforms. Available: https://www.windriver.com/products/vxworks/safety-platforms.

[136] Xi, S., Xu, M., Lu, C., Phan, L. T. X., Gill, C., Sokolsky, O., and Lee, I. Real-time multi-core virtual machine scheduling in xen. In *Proceedings of the 14th International Conference on Embedded Software* (New Delhi, India, 2014), EMSOFT '14, ACM New York, NY, USA.

[137] Yaacoub, J.-P., Noura, H., Salman, O., and Chehab, A. Security analysis of drones systems: Attacks, limitations, and recommendations. *Internet of Things 11* (2020).

[138] Yan, H., Ling, Z., Li, H., Luo, L., Shao, X., Dong, K., Jiang, P., Yang, M., Luo, J., and Fu, X. LDR: Secure and efficient Linux driver runtime for embedded TEE systems. In *Network and Distributed Systems Security Symposium* (San Diego, CA, Feb. 2024), NDSS '24, The Internet Society.

[139] Yoon, M.-K., Liu, B., Hovakimyan, N., and Sha, L. VirtualDrone: virtual sensing, actuation, and communication for attack-resilient unmanned aerial systems. In *Proceedings of the 8th International Conference on Cyber-Physical Systems* (Pittsburgh, Pennsylvania, Apr. 2017), ICCPS '17, ACM New York, NY, USA, pp. 143–154.

[140] Yoon, M.-K., Mohan, S., Choi, J., Kim, J.-E., and Sha, L. SecureCore: A multicore-based intrusion detection architecture for real-time embedded systems. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium* (Philadelphia, PA, USA, June 2013), RTAS '13, pp. 21–32.

[141] Yun, H., Mancuso, R., Wu, Z.-P., and Pellizzoni, R. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *19th Real-Time and Embedded Technology and Applications Symposium* (2014), RTAS '14, IEEE, pp. 155–166.

[142] Yun, H., Yao, G., Pellizzoni, R., Caccamo, M., and Sha, L. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium* (Philadelphia, PA, USA, June 2013), RTAS '13, pp. 55–64.

[143] Zhang, N., Sun, K., Shands, D., Lou, W., and Hou, Y. T. TruSense: Information leakage from TrustZone. In *IEEE Conference on Computer Communications* (Honolulu, HI, Apr. 2018), INFOCOM '18, IEEE, pp. 1097–1105.

[144] Zhao, S., Zhang, Q., Qin, Y., Feng, W., and Feng, D. SecTEE: A software-based approach to secure enclave architecture using TEE. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, UK, Nov. 2019), CCS '19, ACM New York, NY, USA, pp. 1723–1740.

[145] Zuepke, A., Bastoni, A., Chen, W., Caccamo, M., and Mancuso, R. MemPol:

Policing core memory bandwidth from outside of the cores. In *29th Real-Time and Embedded Technology and Applications Symposium* (2023), RTAS '23, IEEE, pp. 235–248.

A Design Details

A.1 Example Loops

```c
while(1) {
  /* Handle all secure device events */
  while((res = chardev_read(serial_dev, buf, n)) > 0)
    handle_serial_data(res, buf, n);
  /* Handle up to max_sz io_uring events */
  for(int sz = 0; sz < max_sz; sz++) {
    struct io_uring_cqe * cqe = ringmaster_peek_cqe(rl);
    if(!cqe) break;
    res = ringmaster_cqe_get_result(cqe);
    switch(ringmaster_cqe_get_data64(cqe)) {
      case ARENA: handle_arena(cqe); break;
      case SOCKET: do_bind(res); break;
      case BIND: do_listen(); break;
      /* etc ... */
      default: handle_error(cqe); break;
    }
    ringmaster_consume_cqe(rl, cqe);
  }
  yield(); /* handled all events this period */
}
```

Figure 10: Example Ringmaster enclave event loop

In our adversary model's assumptions (§3.2, we suggested that enclaves must be implemented to avoid potential timing vulnerabilities. However, this is not a major burden for enclave design; we illustrate this in Fig. 10 and discuss in §5.4. This code shows a simple polling event loop, where an enclave is handling incoming secure device and `io_uring` events. We can observe that by "peeking" on incoming completions, we can avoid hanging waiting for the OS to respond.

A.2 Background: Preemption and Scheduling of Linux

Past work on real-time enclaves and TrustZone-assisted hypervisors has demonstrated how to preempt and schedule Linux on ARM architectures [46, 78, 96, 106, 132]; we use these core ideas for Ringmaster, and will briefly explain how we applied these designs. Typically ARM cores come with a separate Secure world timer device, which can only be configured from `S-EL1` (secure kernel mode) or `EL3` (firmware or monitor mode). We configure the timer to periodically interrupt and the control registers to trap into `EL3` for secure timer interrupts. From `EL3` mode, Ringmaster OS can take over, saving and restoring state, switching to enclaves, *etc.*.

Ringmaster OS implements a real-time scheduler which can support fixed-priority (FP), earliest-deadline-first (EDF), among other algorithms. The scheduler is budget-enforcing: each enclave process has a time budget which is replenished at the enclave's defined period; once the budget is exhausted the enclave will not be scheduled until replenishment. Each Linux core is represented as a thread in the Ringmaster OS scheduler, the main difference is instead of restoring only user-space registers, Ringmaster OS must also switch to `EL3` to swap `EL1` registers for this type of thread.

```c
while(1) {
    /* wait/yield() for 10ms period */
    wait_period();

    clock_gettime(CLOCK_MONOTONIC, &now);
    dt = (double)(now.tv_sec - last.tv_sec)
            + (double)(now.tv_nsec - last.tv_nsec) * 1e-9;
    last = now;

    /* calculate target acceleration */
    double u = pid_step(&pid, bot.theta, dt);

    /* update the motors or simulation */
    bot_step(&bot, u, dt);

    /* write() formatted string to stdout, MAY BLOCK for Linux */
    log_data(
            "i=%07d theta=%+09.5f theta_dot=%+09.5f "
            "x=%+08.4f u=%+09.4f dt=%8.4fms",
            i,
            bot.theta,
            bot.theta_dot,
            bot.x,
            u,
            dt * 1e3);

    if (fabs(bot.theta) > FALL_THETA || fabs(bot.x) > FALL_X)
    {
        log_data("pid_demo: BOT FALLEN i=%d theta=%.4f x=%.4f\n",
                i,
                bot.theta,
                bot.x);
        break;
    }
}
```

Figure 11: Inverted Pendulum PID Loop

A.3 Starting Enclaves with Resource Donation

An adversary may try to launch many enclaves to exhaust memory or time resources—the two primary finite resources in our system design. These resources need to be distributed securely when enclaves are allowed to be dynamically started. Ringmaster OS leverages PARTEE's budget-enforcing scheduler, which ensures that CPU resources are bounded. Furthermore, with a `ulimit`-style physical memory quota, Ringmaster OS can ensure that memory is also bounded per enclave process. When spawning a new enclave, Ringmaster OS implements a resource donation system. A parent thread must donate some of its time budget and memory quota to the child process.

A.3.1 Starting enclaves during bootstrapping. During the boot process, Ringmaster OS assumes exclusive control of the platform before Linux starts. The EL3 firmware initializes Ringmaster OS, which configures memory isolation, interrupt routing, and the secure timer before releasing the Normal world. We assume the boot chain is authenticated (e.g., via secure boot), so that the firmware and Ringmaster OS binary have not been tampered with. Enclaves can be configured to launch at boot; these later attach to proxies once Linux boots.

A.4 Managing Free Arenas

When it comes to managing freed arenas, we can borrow many techniques from traditional memory allocators; however, the key difference is that we cannot trust the contents of the shared memory blocks. Traditional boundary-tag allocators typically have a block header stored adjacent to the allocated or freed block. For Ringmaster, we want to avoid storing block metadata in shared memory because an adversarial Linux could overwrite the header, causing a memory corruption exploit. This means that block metadata must be stored in private enclave memory, and we cannot use constant-time pointer arithmetic in the arena `free` function to identify the block's metadata. The shared memory block metadata problem further reinforces our choice of an arena-based system, where arena metadata is stored in a `struct` which is owned by the programmer. Once an arena is freed, the metadata struct can go into a free-list or binning data structure, as used in dlmalloc [69].

A.5 Early Request Optimization

Given the potential overhead of additional `IORING_OP_ENCLAVE_MMAP` calls, we want to minimize the number of requests made. Ringmaster implements a few techniques; however, the best optimizations remain an open problem. First, we can prefetch an approximation of the necessary shared memory even before the enclave's `main()` begins execution. At enclave launch, Ringmaster's user-space library checks for an environment variable containing an initial request size for shared memory. By making a large request immediately, the enclave overlaps the request time with other initialization tasks. Once a block arrives, it can be divided into useful arena sizes; we do this with a low-level buddy allocator. Upon exhaustion, we request another block based on the previous request heuristics.

A.6 Thread- & Process-Level Parallelism

A.6.1 `pthread`. Ringmaster can support multi-threaded enclaves as long as the Ringmaster OS supports thread-level parallelism. For future work, Ringmaster OS can provide an interface for donating budget to a new thread. Additionally, Ringmaster OS should provide mutexes to prevent infinite blocking.

A.6.2 `fork()`. It is likely possible to implement `fork()` for Ringmaster, the design would be to first invoke a `fork()` in the proxy process, for example through the addition of a `IORING_OP_FORK` or through a enclave-proxy IPC command pipe. Then it would be needed to implement `fork()` as a Ringmaster OS system call.

Unfortunately, there are some complications with simply forking in this way. First, the shared memory would need to be renegotiated somehow, as the regions would be shared between child and parent. Second, the forked proxy would need a way to connect with the new enclave, *i.e.* inform Ringmaster OS about the proxy's new physical pages associated with the `io_uring` SQ and CQ rings; however, the proxy will need a handle for that new enclave process, which may be difficult to identify. Finally, `fork()` will add complexity to Ringmaster OS, and as discussed in recent research, `fork()` should likely be deprecated because of the problems of state duplication in the kernel [15].

A.6.3 `posix_spawn()` *and* `io_uring_spawn()`. Josh Triplett recently presented proposed operations for `io_uring`: `IORING_OP_CLONE` and `IORING_OP_EXEC` [126]. The power of these two operations in `io_uring` is that when linked into chains of operations with the `IOSQE_IO_LINK` and `IOSQE_IO_HARDLINK` flags, a chain of *clone*, followed by some linked *open*, *close*, and other file operations, followed by an *exec* operation allows an entirely programmable spawning interface to happen with zero context switches. If released publicly, these features would allow Ringmaster to support spawning new Linux processes—which could be Ringmaster proxies that start new enclaves. Furthermore, this would also allow the implementation of LibC's `posix_spawn` using `io_uring_spawn` as the backend.