Abstract

Building a new foundation for critical time-sensitive systems: security and performance through enclaves

Richard Thomas Habeeb

2025

Many types of modern critical systems are response-time sensitive, *e.g.* autonomous taxis, industrial robotics, or even cloud providers. Due to their application needs, the software stacks needed for these systems are typically large, complex, mostly third-party, internet-connected, and black-box; however, these properties make them difficult to secure and verify—leading to potential harm or economic loss. While past work has shown how to build "clean-slate" formally verified time-sensitive cyber-physical systems (CPS), these approaches struggle to scale to the needs of today's applications. This dissertation presents work towards building a new foundation for critical time-sensitive systems through techniques that embrace existing legacy and third-party software requirements. Using secure enclaves and trusted execution environments (TEEs), we invert the clean-slate approach, protecting only critical applications.

We broke the construction of this platform into two parts, each solving issues related to availability, performance, and usability. Enclaves are used to protect code and memory against an untrusted OS, but they generally do not have good availability protections. In response, we first present *PARTEE*, the first design and implementation of a "partitioning" TEE OS for the diverse, distributed, and time-sensitive robotics software ecosystem (*e.g.* ROS). PARTEE ensures time-sensitive enclaves cannot be denied service by partitioning system resources, providing reliable communication channels and a time-sensitive system call interface. Secondly, 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 service is denied by the untrusted OS, 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. Together, we use the combined PARTEE and Ringmaster designs to build many applications including an autonomous UAV system which is able to continue flying its mission even when compromised by an adversary.

Building a new foundation for critical time-sensitive systems: security and performance through enclaves

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Richard Thomas Habeeb

Dissertation Director: Zhong Shao

December 2025

To my family, for your unending support, love, and encouragement to shoot for the moon.

Acknowledgments

This research would not be here without the support and guidance of so many people who I owe a great debt to. I am deeply thankful for my PhD advisor, Zhong Shao. He has taught me so much about how to approach hard problems and how to defend my ideas. His kind feedback has both grown and humbled me, and his enthusiasm for this work has been forever imparted to me. I am so grateful for Man-Ki Yoon, as a mentor and colleague. He helped me through many difficult rejections and provided so much insight that I needed to push forward. Additionally, I am thankful to Hao Chen, through building this OS together, I learned so much from your approach to engineering and safety-critical software. I also want to thank my collaborators and peers in my research group, Jérémie Koenig, Yuyang Sang, Wolf Honore, Arthur Vale, and Yixuan Chen, for teaching me so much and helping me work out my research ideas. Furthermore, I need to thank Abhishek Bhattacharjee, for teaching me how to continually strive to improve our teaching while also thinking deeply about research. Thank you to Ben Cifu, Alex Briasco-Stewart, Dion Ong, Bradley Yam, Alexander Jones, and Cesar Segura for the amazing contributions to our CertiKOS prototype.

I am also so incredibly thankful for all of the incredible mentors, peers, and colleagues who I have learned from over the years. I want to thank again Simon Ou, my Master's advisor, who set me upon this wild journey into research and academia. I would like to also specifically thank Andrew King, Parisa Khorsland, and Jake Askeland for teaching me so much about building autonomous vehicles at Zoox. Additionally, I must acknowledge Kyle McGahee, David Maas, and Garrett Peterson for shaping how I think about robotics, embedded systems, and software development in general.

I want to thank the many friends for all of the incredible, life-changing memories I made through this time. Especially, thank you to Talley for the best ice-breakers, the email threads and When-to-Meets for Gryphon's pool nights, the COVID walks down Farmington trail that kept me sane, the joyful and chaotic parties, for that one adventure where we ended up sneaking a whole bottle of wine into Toads, and for always ending the night at Mamouns or Rudys; I could not have asked for a better, more generous friend. Thank you to Ty for being so unapologetically, genuinely yourself, for being sharp-tongued and warm-hearted; you may not know it, but you create and

hold space for people to live their truth, even in a world that sometimes can't hold you. Thank you to Judy for making me organize, helping me through hard times, and for a rich, complex, and unique friendship that has changed my life. Thank you to Caleb for sharing my obsession with the Open-Source Religion and my love for old technology. Thank you to Jim for never getting mad that I don't watch sports but always inviting me and letting me come anyway. Thank you to Nick Lindsay, for always being so kind and for going along with my craziness. Thank you to Madison, Abigail, Nick, Zina, Vlad, Anabel, May, and many others for the countless nights dancing at Partners—especially to Madison and Abigail, who were probably getting signatures at Partners, for showing me what serious and joyful organizing looks like. Thank you to Chris Harshaw for being an overflowing and generous person who I've often looked up to. Thank you to Jill Emerson, Jessy Yates, Karthik Sriram, Samuel Judson, Jake Brauer, Kimmy Cushman, Paul Seltzer, Lena Eckert-Erdheim, Josh Stanley, Alex Josowitz, Bugra Mirsat, Lukey Elsberg, Emily Lorin, Ryan Carlson, the whole OC, CC, and BC teams, and the many other people that given me feedback, mentorship, friendship, love, and who shaped who I am today.

Additionally, thank you to my mother; you have always given me the profound freedom, respect, and love I needed to grow my curiosity for the world. Thank you to my father; you always inspired and encouraged me to build and create things, and to never be afraid to get my hands dirty. Thank you to my step-dad; you showed me how to be organized and how to succeed in this world, and you always helped me feel secure. Thank you to my step-mom; you have pushed me out of my shell, and helped me be the best person I could be. Thank you to my grandparents, Grace and Jim, you have opened up my world in ways that I could have never imagined. And, thank you to my grandparents, Ute and Ron, for teaching me about art, music, and philosophy, and for inspiring me to go to graduate school. Thank you to my brothers, and the rest of my family, who I am today is because of you. Finally, thank you deeply to my wife, Tati. You have provided me with unlimited support and encouragement through dark days and good times. I am in awe of your curiosity for the world and your generosity to everyone around you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problems with the blank-slate approach to software security and verification

For many complex time-sensitive systems—*e.g.* autonomous taxis reacting to pedestrian street crossings or cloud providers meeting service-level-agreement (SLA) response times—security is the Achilles heel that could lead to major economic [14] and safety problems [107, 68]. Consider the self-driving car example. On one hand, consistent improvements over time in machine learning, AI, and GPUs have brought about autonomous drivers that are on average much safer than their human counterparts [219]. Yet, a large fleet of internet connected vehicles or a data center of servers, running identically versioned software stacks of hundreds of millions of lines of code, is waiting for a virus to spread like wild fire, and we have already seen examples of real-world attacks [160, 26, 25, 38]. In the past, to ensure safety, the airline, automotive, and manufacturing industries developed *simple, custom, and isolated* real-time cyber-physical systems (CPS); however, modern iterations tend to be *complex, mostly third-party, and internet connected.*

One of the key technical questions driving this dissertation is how to better secure and make reliable technology that has those latter properties (complex, mostly third-party, internet connected, and black-box), especially software with real-world impacts and responsibility. I use the term *time-sensitive* as an umbrella term which covers both real-time systems as well as applications where denial-of-service (DoS) attacks or latency issues cause serious failures (but may not be traditionally considered real time). For old systems, exhaustive testing could verify the functional correctness of the system, perhaps for every single line of code (*e.g.* DO-178C), but for

modern systems this type of testing becomes intractable. Furthermore, ideally, we could reason about any type of critical software system in a rigorous way from the ground up; *i.e.* we want to *formally verify* the correctness with rigorous methods.

Past research done through the DARPA HACMS [142], CRASH [58], and the NSF DeepSpec [61] programs and many other projects have shown it is possible to build clean-slate time-sensitive CPS that are formally verified—*e.g.* CertiKOS [76, 79, 78] and seL4 [111, 110, 212]; however, I argue that these approaches also struggle to scale to the needs of modern complex time-sensitive systems.

1.1.1 Problem 1: Complexity in hardware architectures

Due to breakdowns in Dennard scaling and the slowing down of Moore's Law, we have seen increasing consolidation of system functionality onto "system-on-a-chip" (SoC) platfoms. Robotics and CPS use SoCs regularly (*e.g.* NVIDIA DRIVE [161] or Thor [102]) to handle growing application performance needs. In these designs, numerous accelerators and coprocessors, *etc.* form a complex heterogeneous environment in which various tasks are scheduled on tensor-processing units, GPUs, and other accelerators. In this heterogeneous, mixed-trust environment, the management software running at the hardware-software interface is critically important for the safety and security of the overall SoC.

SoCs need large, rich operating systems to manage hardware complexity. The world has benefited massively from Linux becoming a de-facto repository for hardware drivers and common hardware interface, and for a growing number of SoCs, the only practical choice is to use Linux or another Unix-like operating system (OS) to make integration easier with the hardware. The software drivers needed to manage USB, Ethernet, WiFi, TCP/IP, GPUs, and myriad bus devices, for instance, make up tens of millions of lines of code alone, and they are continually added to the kernel with every update [41]. Additionally, hardware components in a SoC may be undocumented or use closed source firmware if they are implemented by third-party vendors, for example. In these cases, drivers are typically provided by those same vendors because they have domain expertise. The end result of all of this hardware complexity means that any clean-slate

approach to building a critical system would have numerous challenges supporting necessary hardware features while verifying all of the drivers.

Formally verified hypervisors lack necessary compatibility and performance requirements for CPS. Clean-slate formally verified OS/hypervisors like CertiKOS [78, 79] and seL4 [111, 110] can provide some of the benefits of a large rich OS by virtualizing it; however, they have a few drawbacks. Firstly, they do not typically provide good device support usually provided by mainstream hypervisors like KVM [109] and Xen [19]. For example, device emulation support, or support for multiple guests at all is limited or non-existent—likely due to implementation and verification complexity. Second, virtualization purely as a mechanism for security leads to difficult performance tradeoffs, especially for any power-constrained contexts [90]. In particular, for systems with multiple security domains, virtualization can over-consume both time and energy in address translation tasks. Ultimately, for these reasons we are cautious about hypervisor-based clean-slate design as a scalable long-term solution.

1.1.2 Problem 2: Third-party and legacy software make business sense.

For developers working on robotics and CPS applications, working completely from scratch is not usually an option or desired, so any approaches to verifying critical software must take into account this ecosystem. It is common knowledge among start-up companies, for example, to "ship early and often," to allow for iteration based on customer needs [145]. In particular, developers and companies can use the Robot Operating System, or ROS [173], to quickly bootstrap a working prototype which later can be built out. ROS not only provides a middleware and user-space libraries, it also comes with a large open-source repository of components for virtually every physical control task you can imagine. ROS builds upon specific version of Ubuntu, with Python and C++ support. Putting everything together, this is an incredibly large stack of trusted software from Linux kernel, essential GNU and Ubuntu user-space programs, ROS middleware, numerous C/C++ libraries, the Python runtime and language-level open-source packages from the Python PyPi repository, and third-party open-source ROS programs from the ROS repository. While parts of this ecosystem may not always be strictly required, each make developers' lives easier, and

together they represent significant inertia and investment into not repeatedly solving the same problems. Furthermore, a "ratcheted" approach to incrementally verifying an entire complex system is difficult because new unverified software is always being developed and checked in across these repositories. We will likely need to embrace this cultural and economic reality if we want to provide verification or security to critical software.

1.1.3 Problem 3: Blank-slate approaches can impinge internet connectivity, file-system access, and other rich OS features.

Applications that are highly time-sensitive, like a real-time robotics control task or data-center server application, still usually need access to a variety of services and features normally provided by the OS. This is especially relevant in a new era of AI, as large-language models can easily exceed hundreds of billions of parameters, effectively requiring centralized cloud support for performance; thus, a constant internet connection becomes important. In addition, for next-gen robotics like autonomous taxis most applications still need to constantly log and record sensor data, and intermediate processing information related to decision making—for debugging, monitoring, and legal purposes. Moreover, for time-sensitive data center applications, network and file system access are simply non-negotiable; *e.g.*, a database application needs both of these. For these reasons, a file system and TCP/IP stack, among other related OS services are an indispensable feature, but they add technical burden to any blank-slate approach.

1.2 Towards future modular verification of complex time-sensitive systems through *enclaves*

In this dissertation I will discuss my two major investigations (in collaboration with my research group) towards a new approach to securing time-sensitive, critical systems—inverting the blank-slate approach and embracing the complexity of modern systems. At the core, we take the "enclave" approach to security, where critical software can be protected from the rest of a system, so that even if other parts of the system are compromised, there is a reliable and trusted enclave service. We have developed an unverified next-generation version of the CertiKOS operating system

[76, 79, 78], which is able to now serve as privileged firmware in a way that can create time-sensitive enclave applications which are protected from a primary rich host Linux OS. The goal is to use this version as a starting point for a future formal verification effort; for now focusing on the systems efforts first required.

PARTEE. Chapter 2 shows how modern TEE architectures are not designed for availability or data-centric communication. TEEs are typically designed to fully be at the whim of the untrusted OS's scheduling and fault-handling decisions. Thus, they are not suitable for any hard real-time CPS or robotics applications, and they do not provide good guarantees to softer real-time internet-of-things (IoT), time-sensitive cloud vendor, or latency-sensitive blockchain auctioning algorithms. I show that the impact of the availability security property is highly significant for TEE OSes, and then, I present PARTEE, a new TEE platform, based on a new-version of CertiKOS, which considers availability security from the ground-up. Furthermore, we design a TEE OS-mediated middleware framework which allows performant and secure messaging among a network of applications and enclaves.

Ringmaster. Many past security approaches completely isolate time-sensitive programs with a hypervisor; however, this prevents the programs from accessing useful OS services. Especially in the context of ARM systems, the options available for the TrustZone provide very little programmer support, and is highly error prone [35]. I introduce Ringmaster, a novel framework that enables enclaves or TEEs to asynchronously access rich, but potentially untrusted, OS services via Linux's *io_uring*. Ringmaster builds upon the PARTEE framework: when service is denied by the untrusted OS, 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 a unmanned aerial vehicle, Ringmaster achieved nearly 1GiB/sec of data into enclave on a Raspberry Pi4b, almost zero throughput overhead compared to non-enclave tasks.

A note on authorship. Chapters 2 and 3 are made up of content from two technical conference papers that are under peer review at the time of this writing. This work was done in collaboration with my research group and peers where I am the primary author.

1.2.1 Overview of technical contributions:

In order to build this new future platform, I had to handle many previously unsolved technical research and engineering problems. The outcome of this work is the following solutions:

1. **A new TEE OS, written from scratch for availability.** Previous work on TEE OSes for real-time systems, called RT-TEE [218], unfortunately did not achieve proper availability. While they added a real-time scheduler and provided an device sharing, this work does not account for the fact that modern TEE OSes are not designed for availability (See §2.3). Thus, any privileged host OS or root process can easily DoS any enclave, even with RT-TEE. For robotics and CPS, this is a critical safety vulnerability. From our analysis, we concluded to achieve availability, a new TEE OS must be built since this property deeply affects system design (§2.5).

2. **A new approach to time-sensitive zero-copy enclave IPC that also transparently integrates with popular middleware frameworks like ROS.** We developed a new user-space library and TEE OS mechanism to allow enclaves to asynchronously publish and subscribe to middleware topics shared with ROS (Robot Operating System [173]). This design includes an optimized zero-copy shared memory interface for trusting enclave parties, and a single-copy partially read-only interface for distrusting parties. With this protocol no two enclaves can block each other in communication, and distrusting enclaves cannot corrupt or modify each other's messages.

3. **A combined design, integrating TrustZone firmware and TEE OS for improved performance and security.** By default for ARMv7, ARMv8, and ARMv9 the TEE OS is independent from the underlying firmware and boot loader stack; they are two separate binaries typically. One feature of the firmware is that it provides a "monitor call" (akin to system or hyper call) interface to manage the hardware related to power, voltage scaling, and CPU

frequency. Since there's no way for the TEE OS to configure firmware security policy about this hardware, the untrusted host OS can turn off cores or slow the system down via the firmware—denying CPU time to enclaves. Additionally, this firmware needs to switch between worlds, which can add extra latency and cache pressure. To fix this, I redesigned an integrated TEE OS and firmware system without any loss of hardware security (§A.1.1) and measured improved context switch overheads by at least 10%.

4. **A new Linux kernel mechanism for high-performance systems call from TrustZone enclaves.** In many ways, the TrustZone is very difficult to use, especially for time-sensitive real-time applications. One of the major reasons for this is the limited interface provided between Normal and Secure worlds which must be programmed by hand using a software development kit (SDK). First, there is no way to easily access host OS data, enclaves must be invoked by the host OS, and any invocations must be custom designed. Second, the only interfaces that exist between worlds are explicitly synchronous, limiting any potentially for availability properties. I designed a mechanism (Ringmaster) to allow arbitrary I/O system calls from the TrustZone (§3.4), and a novel mechanism for marshalling arguments between worlds using a modified Linux `io_uring` interface (§3.4.2).

5. **A secure, and time-sensitive interface for safer I/O access from enclaves.** Using Ringmaster's async system call interface I built a series of security features needed for high-performance availability-sensitive I/O. Namely, I built a shared-memory allocation system, buffered reader and writer interfaces, a C implementation of a promise/functor pattern used to handle callback context across Ringmaster's interfaces, and safe abstractions over raw shared memory I/O rings. The result is that enclaves can cleanly identify and manage potential sources of DoS using async/await semantics.

6. **A custom C standard Library that allows for legacy applications to run in the Arm TrustZone with secure real-time properties.** With Ringmaster as the backend, I instrumented a C standard Library to make system calls using `io_uring` (§3.6). With this, we provide a new way to run unmodified legacy applications as TrustZone enclaves. Furthermore, we show that this approach can achieve better performance in many cases due to

optimizations around async parallelized I/O (§3.9.3).

7. **Working implementations leveraging PARTEE and Ringmaster to build a highly secure, safe, and reliable drone prototype**. We explore PARTEE and Ringmaster's security in sections 2.7 and 3.8.1. To achieve this, I implemented the tooling and instrumentation to configure, build, integrate, run, profile, and test a Linux kernel, a customized file system, a new firmware/TEE OS that supports multiple platforms, and various sets of enclaves.

Chapter 2

It's a non-stop PARTEE! Practical multi-enclave availability through partitioning and asynchrony

Due to the growing third-party software stack necessary to build modern data-rich robotics and cyber-physical systems (CPS), it has become important to protect safety-critical and timing-sensitive programs and their communication—even against an adversarial rich operating system (OS). Enclaves and Trusted Execution Environments (TEEs) are often used to protect code and memory against an untrusted OS, but they generally do not have good availability protections. To illustrate, we present three attacks, showing that even with secure timer access and memory protections, existing TEE platforms still face challenges in achieving availability.

In response, we present *PARTEE*, the first design and implementation of a "partitioning" TEE OS for the diverse, distributed, and time-sensitive robotics software ecosystem. PARTEE ensures time-sensitive enclaves cannot be denied service by partitioning system resources, providing reliable communication channels and a time-sensitive system call interface. We analyze the security and performance of PARTEE using an unmanned aerial vehicle implemented on the Raspberry Pi4B using the ARM TrustZone, and show that despite the behavior of an adversarial partition or a rich OS, the drone's most safety-critical enclaves remain available and can communicate to prevent harm or damage. [1]

[1]This chapter is adapted from an in-review conference paper submission by Richard Habeeb, Hao Chen, Man-Ki Yoon, and Zhong Shao.

2.1 Introduction

Next-gen robotics and cyber-physical systems (CPS) increasingly rely on machine-learning and large-language models integrated with frameworks like Robot Operating System (ROS) atop multicore system-on-chip (SoC) devices [173, 102, 181]. These systems feature highly distributed software architectures composed of many third-party processes communicating over a publish/subscribe framework [67, 146]. As robotics and AI-assisted physical systems take over safety-critical domains—autonomous vehicles, drones, smart avionics systems, surgical robots, factory robotics, smart medical implants and devices, *etc.*—they face growing security concerns [28, 68, 172, 199, 224, 107, 156, 27]. With increased complexity and Internet connectivity, many are at risk of privilege escalation once an adversary gains a foothold [160, 56, 38, 220, 186, 26, 25]. Furthermore, many CPS must be *available* to respond to sensor input to avoid danger or economic harm; thus, privileged denial-of-service (DoS) attacks should be considered.

Trusted Execution Environments (TEEs) offer a lightweight solution by isolating processes in *enclaves* from an untrusted host operating system (OS) [180], but these are generally not designed to provide the availability guarantees needed by CPS [2]. While some TEEs can provide a basis for *CPU availability* (access to CPU time, *e.g.* using ARM TrustZone [158] with hierarchical scheduling [218, 183, 184]), achieving stronger availability for critical applications is tricky on a modern TEE. Of the platforms that need a TEE OS (like OP-TEE [133]), adapting these solutions to provide availability is difficult due to numerous timing and DoS vulnerabilities stemming from their designs. We propose that they still face at least four fundamental availability challenges, particularly for CPS:

Problem 1: **Over-trusted host OS for essential enclave services.** Because enclave platforms in general are not typically designed for availability, most delegate scheduling to the untrusted host OS[47, 94, 53, 132, 22, 23, 12, 164, 80, 211, 193, 97, 190, 1, 215], and enclaves can be trivially starved. Many designs rely on the OS to service page faults, manage page tables, handle system calls, or manage encrypted swap space [47, 80, 51, 133], and many allow encrypted access to enclave pages. Thus, an untrusted host OS can easily stall enclave execution.

Problem 2: **No systematic management of TEE resource availability.** The TEE OS itself

manages finite resources which can be exhausted by the untrusted host OS or any enclave. This includes CPU time, physical memory, and device I/O; however, subtly, any type of TEE OS *kernel object* could be a vector for a DoS attack. For instance, OP-TEE [133] heavily uses a kernel heap for numerous types of objects which can be drained by the untrusted OS or a faulty enclave (demonstrated in §2.3). Seemingly, a TEE OS must be structured from the start to prevent over-consumption.

Problem 3: **No defenses for both inter-enclave communication integrity and availability.** Due to the distributed nature of modern robotics architectures, inter-process communication (IPC) can be critically important and time sensitive. For instance, messages between sensor-processing and actuation enclaves should have both integrity and availability to ensure safe and timely control. Yet, for many designs, enclaves' IPC can be denied or corrupted by an adversary (see §2.3). Furthermore, TEE IPC does not integrate well ROS-like patterns, leaving enclaves isolated.

Problem 4: **Non-negligible algorithmic overheads.** Some TEE OS services have scaling runtime overheads which can be abused to deny availability. For instance, enclaves can be spawned dynamically, but this requires authenticating the enclave's signature; the memory and time needed to perform this operation can be non-trivial, as they increase with the binary size (§2.3). Other operations, such as device I/O, large memory maps, or swapping could lead to delays waiting for locks or untrusted OS services.

To show that these four problems are not simply theoretical, we provide three concrete example attacks and discuss more in §2.3. TEEs should be able to handle the complex distributed software ecosystem of next-gen CPS; ideally, they should isolate unrelated subsystems, ensure reliable fast communication, and allow for secure run-time flexibility. Unfortunately, these attacks highlight difficulties with modifying existing TEE OSes to support such needs.

Our proposed solution: PARTEE, a resource partitioning TEE OS designed to address these availability problems. As shown in Fig. 2.1, PARTEE creates resource containers [18] called *partitions* which divide up TEE OS kernel objects, physical memory, CPU time, device I/O, and IPC access into isolated security domains. Enclaves then run in partitions with dedicated resources. Partitioning enables system developers to clearly define the security domain for each potential

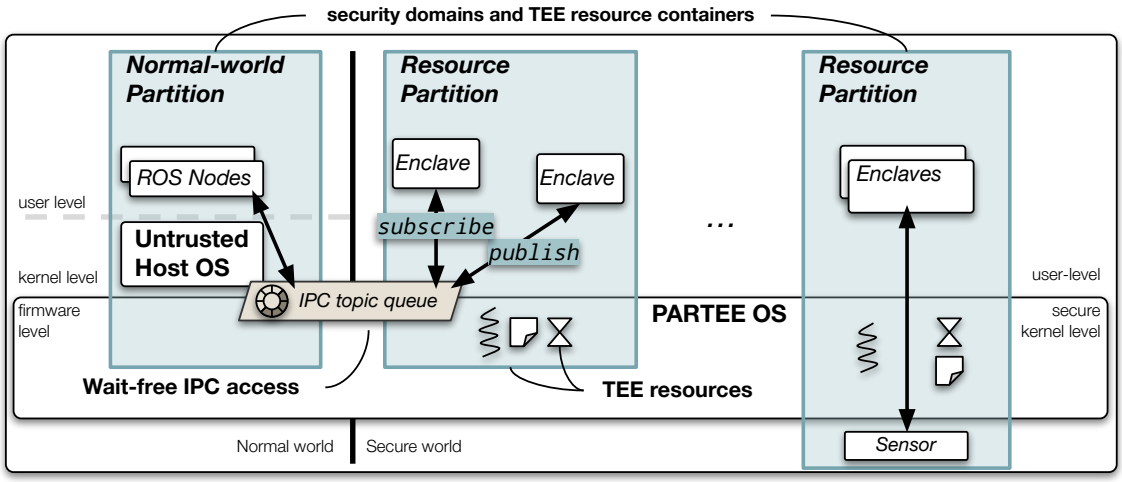

Figure 2.1: PARTEE prevents DoS attacks by dividing TEE resources into containers called *partitions* which also act as security domains and hierarchical scheduling partitions; IPC availability is protected and made practical for distributed ROS architectures via a wait-free broadcasting mechanism.

enclave, ensuring all have essential resources. To handle the challenge of completeness, we considered how DoS (and partitioning) will affect every software abstraction level in its design, from the start, in order to ensure availability. The result is a massively reduced trusted computing base (TCB) for critical control processes compared to direct execution on a host OS.

PARTEE provides ROS-compatible publish-subscribe IPC for enclaves that balances performance and security, ensuring that messages between honest enclaves are never dropped or corrupted with minimal overhead. Non-enclave (potentially untrusted) ROS processes can also publish data, but only to communication "topics" accessible by their partition. To support ROS-like design patterns, PARTEE works like modern TEE architectures, allowing dynamic spawning of enclaves to handle flexible workloads via process-level concurrency. Due to the necessity for a (less-tested or unverified) third-party software ecosystem, we design partitions to handle exploitable or potentially malicious enclaves. Enclaves from trusted sources and third parties can be authenticated on a per-partition basis to ensure separation.

One of the major research challenges for PARTEE is how to handle IPC DoS at a TEE OS level. PARTEE's IPC is by default asynchronous, so tasks are guaranteed periodic execution time to handle any incoming published messages (or lack thereof). We use the ARM TrustZone to preempt and schedule an untrusted OS similar to previous works [183, 168, 218]. While regular

devices are assigned to the host OS, time-sensitive sensor processing and actuator I/O (*e.g.* UART, SPI, or CAN) is provided directly to enclaves by PARTEE, guaranteeing availability for safety-critical data and control. As an example, consider a smart automotive touch dashboard and media center; such a device must process and display data from a variety of source and services. It must support numerous complex consumer services using unreliable, untrusted connections such as cellular, WiFi, USB, and Bluetooth. This device should also provide real-time visual and auditory alerts based on collision-avoidance sensor data. However, if such a system is hacked or malfunctions, all bets are off [160, 26, 154]. PARTEE paves the way for protection of this type of critical use-case on complex devices like this by allowing partitioning of various time-sensitive subsystems and via available IPC and I/O.

To examine the efficacy of PARTEE, we use it to build a highly secure search-and-rescue drone on the Raspberry Pi4B. Our experiments show that in the presence of a malicious host OS and even a malicious enclave, the remaining critical systems will function with essential functionality. For instance, the drone can continue to avoid obstacles and return to home safely after detecting a DoS attack or malicious message. Summarizing, our contributions are:

- *A novel partitioning TEE OS:* A design that mitigates critical DoS attacks on shared TEE OS resources, objects, and services to protect key aspects of enclave availability (§2.5).
- *Secure enclave publishing and subscribing:* An optimized enclave IPC system that can be seamlessly integrated with ROS-like middleware; it ensures IPC availability between honest partitions, message integrity, and fine-grained access controls via partitioning (§2.6).
- *Real-world implementation and attacks*: An analysis of availability challenges for modern TEE OSes using three example attacks (§2.3) and tests of PARTEE's security and performance with our attack tools and drone implementation on a Raspberry Pi4b (§2.7), with additional support for the NVIDIA Jetson TX2.

2.2 Motivating Context

Motivation 1: Why consider a privileged adversary? Modern robotics, IoT, and CPS generally need a rich OS to support features like live video streaming, voice recognition, or advanced

machine-learning and AI features; however, some of these services may be more safety-critical than others [34, 33]. For instance, recent security analyses showed that Tesla's full self-driving autopilot runs Linux and is connected to the Internet for data collection in addition to the vehicle's controller area network (CAN) bus of critical vehicle control systems [117, 221]. Many others have shown the potential for privilege escalation once a foothold is gained on a CPS [160, 56, 220, 26, 25, 186].

Our drone implementation (show in Fig. 2.10 in §2.7) also concretely demonstrates the devastating impacts of a privileged adversary. If the critical tasks do not run as enclaves, a compromised OS can arbitrarily control and crash the drone by sending manual control or forced motor disarm messages to the autopilot. More subtly, the OS could block or modify IPC between the tasks involved in obstacle avoidance. See §2.7 for further discussion.

Motivation 2: Why consider more than two security domains? As mentioned above, many CPS involve a complex integration of subsystems with varied impacts on safety, mission success, privacy, and quality of service (QoS). Applying the principle of least privilege [182], if one subsystem is compromised or has a fault, the other systems ideally should remain as unaffected as possible [178]; thus, a simple binary division of security domains will not always provide strong security. The ARINC 653 standard [222, 177, 135] and ongoing research on partitioning hypervisors are based on this premise [149]. One notable example is the Boss self-driving architecture [213], which has three subsystems of various tasks just for mission planning: "Lane Driving," "Intersection Handling," and "Goal Selection."

The historical assumption is that all software in the TrustZone is assumed to be correct and "trusted," due to enclave binaries authentication. However, for mobile devices, recent works found numerous real-world vulnerabilities in enclaves, especially due to the open app-store ecosystem [128, 201, 42, 35]. For CPS and robotics, given the diversity and complexity of modern software stacks with software from many sources and vendors, we adopt an *open-system* adversarial model in this work (see §2.4.2). This allows us to model potentially malicious third-party enclaves.

2.3 Availability Challenges for TEE OSes

How are TEEs constructed which makes it difficult to modify them to support the problems outlined in §2.1? To answer this question, we look at the highly popular OP-TEE [133] as an example TEE OS. Typical TEE OSes do not have a scheduler and thus rely on the host OS. For example, instead of using spin locks in the TEE kernel, OP-TEE defers to the host OS and waits for it to return. This is one of nine essential remote-procedure calls (RPCs) apart of the TEE protocol for Linux, which cause the TEE to block until Linux decides to return. Additionally, the host OS can be used to store swapped pages, which it can deny to block enclave execution. Interrupts that occur during TEE execution will cause the TEE to context switch to Linux as well, halting enclave execution for an indefinite period. These functions would be trivial to attack; however, there are many other lurking DoS exploits which highlight other major design problems.

Even though many have proposed ways to use the ARM TrustZone for real-time applications [183, 108, 168, 148, 64, 100], only one recent work, RT-TEE [218], has availability measures for enclave-like TEEs specifically [180]. RT-TEE showed that, conceptually, a TEE OS should provide CPU availability and I/O availability. Although it proposes adding I/O protections and hierarchical scheduling of the host OS and enclaves, this is not yet sufficient to ensure availability. To strengthen OP-TEE's defenses, we also test the RT-TEE artifact, which extends OP-TEE with a real-time scheduler.

Attack 1: TEE OS Heap Drain. We found at least 193 places where `malloc()` or `calloc()` were directly used in the OP-TEE kernel. OP-TEE uses a static Secure-world kernel heap for the majority of TEE OS kernel objects; allocations occur on nearly all public interfaces for both the host OS and enclaves. For example, when pages are mapped to an enclave, an object allocated is to track them. When one enclave invokes another enclave for IPC, OP-TEE attempts to `malloc()` two blocks of memory to share the message. Without adequate heap space, we found that no IPC sessions can be opened for any enclave, and no enclaves can be loaded. Cryptographic operations also require the heap for large prime math, and they can fail without enough space.

We designed a proof-of-concept tool to exploit one of these vectors; it allows an adversary to prevent communication between two enclaves—even without the need for an open-system or

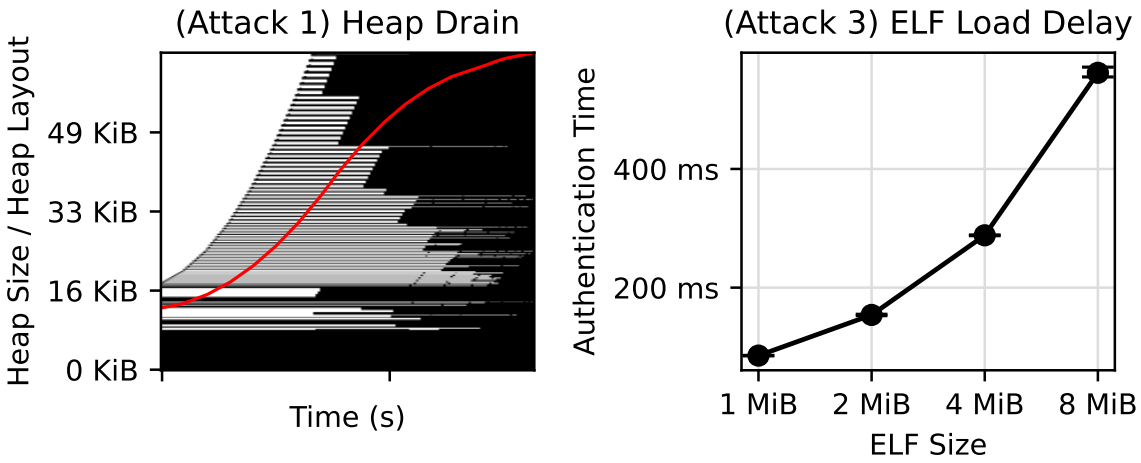

Figure 2.2: **Left:** Heap space over time during a heap-drain attack on OP-TEE OS; black space is allocated memory and the red-line indicates overall capacity. **Right:** Measurements of malicious ELF loading time for OP-TEE by file size, showing linear scaling between size and hash time.

malicious enclave code. For this attack, the adversary host OS repeatedly registers shared memory with OP-TEE. Each registration will result in new objects being allocated on OP-TEE's kernel heap. By strategically altering the sizes of registrations, Linux can create objects with various sizes. With certain patterns of registrations, the adversary can ensure that the heap is completely consumed or heavily fragmented, as shown in Fig. 2.2 with a mapping of heap memory.

If we tried to patch this attack by limiting the number of requests, the heap could fill up naturally, making the attack viable again. The attack could be adjusted to create larger objects or to create objects using different interfaces, as most involve heap memory. Modifying this system would lead to endless cat-and-mouse games trying to patch each interface.

Attack 2: Execution-Blocking IPC Attack. The way that threading and IPC was built for OP-TEE (for the Global Platform [GP] specification for TEEs [73, 74, 75]) did not consider the potential impacts of the availability requirements for CPS—especially for distributed ROS-like architectures. For ROS-like systems, one task does not have to fully trust another enclave to benefit from communication with it. IPC can transfer logging information, high-level objectives, encrypted network traffic, or video streams asynchronously, decoupling data availability from scheduling and allowing validation of message contents. With OP-TEE and any design that uses the GP spec, all IPC between enclaves is implemented as client-server remote-procedure calls (RPCs), where the client's thread context switches to the server's enclave process to invoke some function syn-

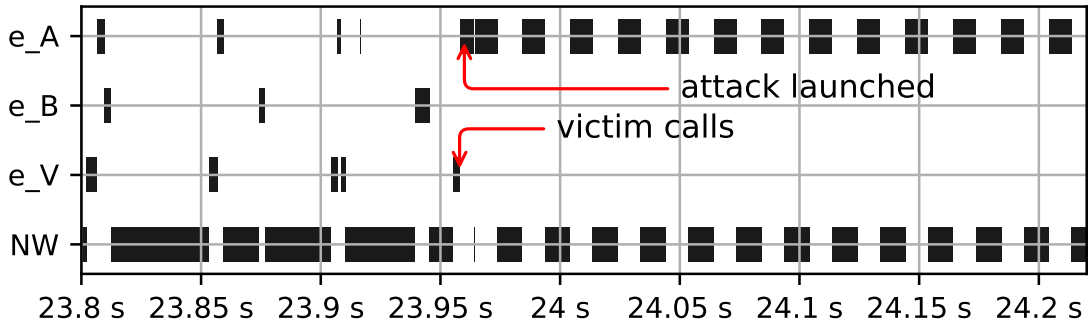

Figure 2.3: Task trace of an execution-blocking attack implementation on RT-TEE [218] where victim $e_{\mathcal{V}}$ calls adversary $e_{\mathcal{A}}$, which never returns, causing $e_{\mathcal{V}}$ to miss its deadline. Periodic budget refills simply resume $e_{\mathcal{A}}$.

chronously (and concurrently with other clients). In this design, every client (caller) must trust that the server will not DoS it, and if a server (callee) is compromised, it can block or delay clients.

No other channels of communication or shared memory could be established (except raw shared memory with the host OS). Hence, any TEE OS that implements this spec (*e.g.* Alibaba's Cloud Link TEE, Huawei's iTrustee, Qualcomm's QTEE, Samsung's TEEgris, Trustkernel's T6, or Trustonic's Kinibi), an enclave's availability is strictly dependent on all other enclaves in its call chain. As noted in **Motivation 2**, we argue this is not a safe assumption. In 2024 alone, fourteen new zero-day exploits in enclaves were found due to the GP specification's lack of invocation argument sanitization and type-checking [35]. Moreover, to simply delay a caller, an adversary does not need to achieve remote-code-execution; *e.g.* overwriting a loop counter or lock variable could cause a DoS.

To demonstrate the impacts of this problem, we implement a victim caller enclave, $e_{\mathcal{V}}$, and an adversary-controlled server, $e_{\mathcal{A}}$, on top of OP-TEE (using the RT-TEE artifact with an event-based scheduler [218]). Fig. 2.3 shows that once the adversary refuses to return, the client will never execute again. When the timer event wakes the victim thread each tick, it is still executing the adversary's code.

To fix this issue, one could organize all IPC so that the most trusted enclaves are servers and the least trusted are clients; however, this leads to troubling design patterns. For instance, enclaves will then implement a server and a client interface for each type of IPC to communicate with lower and higher criticality tasks; yet the system has no way of enforcing any access control

so any enclave can still call anyone else. Traditional real-time priority inheritance schemes will not help here either, if the server is malicious. Forced timeouts, if implemented, could help; though timeouts tend to be very inefficient as one has to grossly overestimate CPU budgets. On top of everything, this form of IPC is wildly inefficient for broadcasting: each subscriber needs a copy and two context switches per message.

Attack 3: Confused Deputy ELF Loader. Certain TEE OS operations can take arbitrarily long amounts of time, controlled by the adversarial host OS. If the TEE OS has any locks held or resources consumed during this process, then the adversary can deny services to enclaves. We developed a tool to demonstrate this problem.

Typically, cryptographically-signed enclave ELF files are stored in the host OS's file system. Using one, our tool can generate maliciously large ELF files, causing the system to delay during spawning. First, the tool generates a valid, large unsigned ELF file. Then, the tool signs it with an arbitrary key, creating a TEE header with a signature of the new ELF file's hash. At this point, if loaded, the TEE OS would immediately reject the ELF because the signature is not authentic. Finally, the tool copies only the signature and hash of the authentic ELF binary to the fake one. Now, when loading the malicious ELF, the system checks the signed hash, which is authentic, so it begins to hash the ELF contents. The malicious ELF causes OP-TEE to act as a confused deputy [87]: consuming large amounts of secure memory, heap, and page tables. In addition, the ELF loader will hold locks over many kernel objects for a long amount of time, limited only by OP-TEE's heap size. Fig. 2.2 shows our measurements for how long this hashing process could potentially take, scaling linearly with the ELF size. The delay is due to hashing overheads and copies of data into the new enclave's address space. Only once it is fully loaded and hashed does the final hash comparison fail, finally releasing the consumed resources.

2.4 PARTEE Goals and Security Models

Using our motivating attacks above, we define several goals beyond prior work, which are later analyzed in §2.8. Related prior work primarily focused on enclave non-starvation, I/O availability, and memory protections.

Goal 1. ***(Guaranteed physical memory reservations)*** *Enclaves should be able to allocate the memory needed to make progress—in spite of DoS attacks.*

Goal 2. ***(TEE resource and service availability)*** *Shared TEE OS kernel state and objects should be available for enclaves to make progress. TEE services must consider how algorithmic input scaling affects all enclaves.*

Goal 3. ***(Wait-free publish and subscribe IPC)*** *No enclave should block during the IPC protocol.*

Goal 4. ***(Guaranteed correct message delivery)*** *Two honest communicating enclaves should not have their messages corrupted or blocked before the receiver can receive it.*

Goal 5. ***(Flexible IPC access control)*** *System designers can specify IPC topic access controls and bandwidths limits while also supporting runtime changes.*

2.4.1 Hardware Model

We consider a multi-core SoC hardware platform with the following: (1) Programmable support for making regions of physical memory read-only or inaccessible from kernel mode; (2) Programmable interrupt access control, so that some may not be masked or handled by the host OS; (3) A secure timer for scheduling, which cannot be accessed by kernel mode; (4) Support for isolation of system power, core voltage, core frequency, clock management, reset management, and other potentially safety-critical hardware components from kernel mode.

This paper primarily looks at ARM TrustZone[7] platforms as an instances of this model. Briefly, the TrustZone provides a bisection of the CPU into two worlds: "Normal" and "Secure" which share main memory and other CPU registers. Privileged programmable firmware is used to interrupt execution of each world to swap out registers; additionally, it is used to manage system power. Memory access control is done on the bus level by a TrustZone address-space controller[10]. Interrupts can be configured to be owned by a certain world and can be disabled during the non-owning world's execution.

2.4.2 Adversary Model

Using our informal attacks from §2.3, we now define a formal adversary model, $\mathcal{A}$. For this model, we define a partition $\mathcal{P}$ as a set of enclaves in the same logical security group or domain. Each enclave, e, belongs to some partition on a system divided into m partitions: $\mathcal{P}_0, ..., \mathcal{P}_{m-1}$. Enclaves in the same partition share "partition-level" permissions (akin to Unix group permissions), so we assume they trust each other accordingly. One partition, $\mathcal{P}_0$, includes a large rich OS kernel (*e.g.* the TrustZone's Normal world). Our adversary has the following properties:

Escalated Privilege: $\mathcal{A}$ can run arbitrary code in $\mathcal{P}_0$'s user (`EL0`) and supervisor (`EL1`) modes. Therefore, it can read or write to any physical memory which is not protected by a memory protection mechanism.

Device Control: $\mathcal{A}$ can configure memory-mapped IO and interrupts of any devices which are not protected by TrustZone bus hardware. $\mathcal{A}$ can start arbitrary DMA transfers, but they must respect the hardware isolation.

Open System: $\mathcal{A}$ can run arbitrary code in an adversarial enclave $e_{\mathcal{A}} \in \mathcal{P}_a$, *i.e.* $\mathcal{P}_a$ is an adversarial partition.

$\mathcal{A}$'s goal is to compromise or deny service to any other enclave $e_{\mathcal{V}} \in \mathcal{P}_k$ where $k \neq a$. Concretely, $\mathcal{A}$ wants to: (1) alter or read $e_{\mathcal{V}}$'s internal state; (2) alter or read $e_{\mathcal{V}}$'s private communication with another enclave $e_{\mathcal{B}}$ where $\mathcal{B} \neq \mathcal{A}$; (3) cause $e_{\mathcal{V}}$ to stall or block during the IPC protocol with any enclave; (4) cause a DoS of TEE OS resources necessary for $e_{\mathcal{V}}$ to make progress.

We assume messages from $e_{\mathcal{A}}$ to $e_{\mathcal{V}}$ can be sent asynchronously; *i.e.* the logic of $e_{\mathcal{V}}$ can make progress in a meaningful way in the absence of a message. Such progress could be an emergency backup plan in the worst case, or normal functionality with a minor loss of quality in the best case. We also assume messages from $e_{\mathcal{A}}$ to $e_{\mathcal{V}}$ can be validated or sanitized using techniques described by related work (§2.6). If $e_{\mathcal{A}}$'s message is strictly essential to both the timing and business logic of $e_{\mathcal{A}}$ with no way of validating or sanitizing it, all bets are off. In our experience and analysis, these assumptions do not meaningfully weaken PARTEE's ability to provide security; practically, they mean that the critical software we want to protect cannot itself be adversarial.

See §2.6 for in-depth design discussions on this issue, and see §2.7 for how these affected our drone implementation. We now describe PARTEE's challenge to this adversary in the following sections.

2.5 PARTEE System Design

At the heart of the PARTEE design is a compact, special-purpose TEE OS, called *PARTEE OS*, which runs at a higher privilege level than kernel mode (EL1). In our implementation, we run this kernel in both ARMv8-A "Monitor" mode (EL3) in place of the firmware and as the Secure-world kernel (S-EL1) of the TrustZone. Running in both modes is primarily used to enforce access control on system power primitives, and we found that doing this reduces the TCB (removing the need for a separate Trusted Firmware binary [7]) and can improve world switch latency (see §A.1.1). Normal hypervisor mode (EL2) is not needed for PARTEE (as long as the hardware model in §2.4.1 applies), and secure hypervisor mode (S-EL2) is not used.

PARTEE partitions prevent one part of the system from denying service to another, whether through fault or adversary attack. They are resource containers [18], similar to Linux *cgroups*, *i.e.* a lightweight group of processes which share a subset of the whole system's resources:

Definition. *(Partition)*

1. *An exclusive set of system resources (e.g. CPU budget, physical memory, I/O, IPC channels).*
2. *A set of running enclave processes in the same security domain and hierarchical scheduling partition.*
3. *A separate root-of-trust for dynamic enclave spawning.*
4. *The host OS (Normal world) is treated as one partition.*

We define an enclave as a (Secure world) user-space process running in some partition on PARTEE, which provides it with essential OS services: page-table management, scheduling, and time-sensitive I/O. Fig. 2.4 shows an overview of the partitioning architecture.

The PARTEE Rules: We envision the system designers would divide a system up by functionality and criticality (*e.g.* DO-178C software levels). Currently, at development and provisioning time, developers must define the PARTEE *Rules* in a specification file. These rules are enforced by

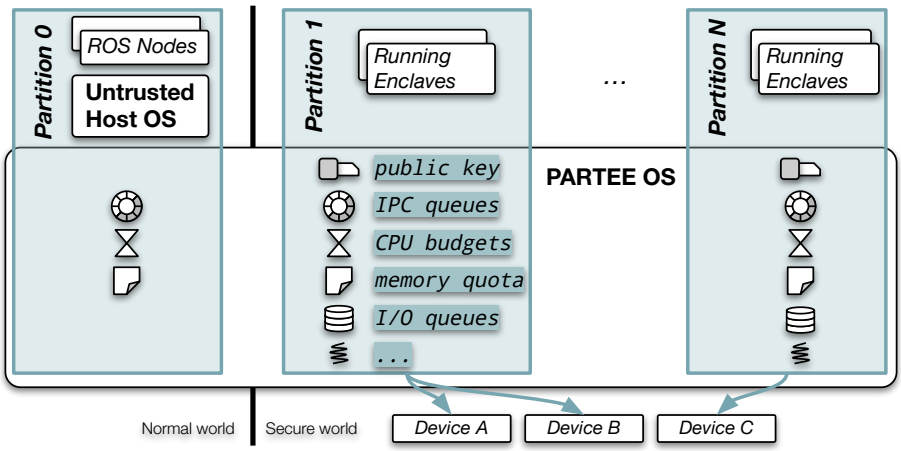

Figure 2.4: PARTEE's partitioning system design where TEE OS managed resources and objects are partitioned among running enclaves.

PARTEE at runtime to ensure isolation; they specify the initial sets of partitions and their static properties along with IPC access control policy. We will briefly discuss background for the main security techniques, then dive into how PARTEE manages resources.

Question 1: *How is time partitioned?*

PARTEE OS uses well-known TrustZone techniques [183, 168, 148, 218] to preempt Normal world execution via a secure timer interrupt. The timer drives a *budget-enforcing hierarchical real-time scheduler*, which first selects a partition based on the scheduling parameters defined in the rules, then selects an enclave from that partition to run (or Linux). See §A.1 for further details.

Question 2: *How is memory protected?*

PARTEE uses standard TrustZone techniques for memory access control. Regions of the physical address space are protected using memory bus security hardware (*e.g.* a TZASC [10]), see §A.1 for details.

2.5.1 New TEE OS Design for Availability

What types of resources must be available for an enclave? To answer this question systematically, we use the OS implementation process as a discovery methodology. TEE OSes like regular kernels manage many different kinds of state for processes and hardware, and not all states can be efficiently statically allocated due to dynamic workloads. We assume that partitions can be

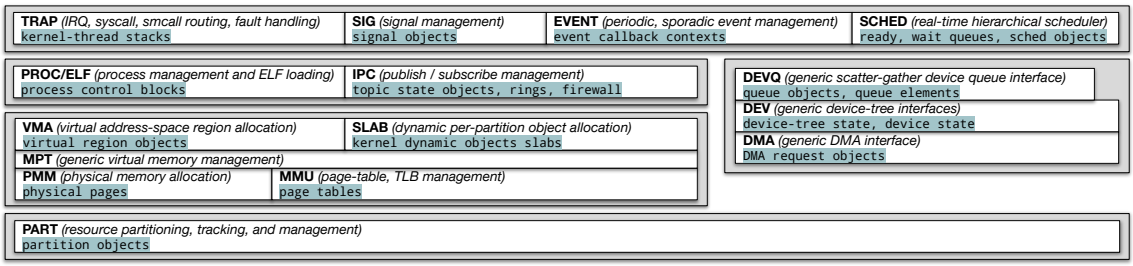

Figure 2.5: Breakdown of core PARTEE OS design layers showing what enclave availability-relevant state is managed at each abstraction to prevent DoS

dynamically started, but that there is a statically defined maximum number of partitions and enclaves. Our approach was to break the TEE OS into small abstraction layers to encapsulate each piece of state; below we discuss several key layers, shown in Fig. 2.5.

Physical memory allocation: Starting at the lowest abstraction, physical pages must be assigned to partitions to prevent memory DoS. This layer tracks page allocations, aliases, and frees charging against per-partition page *quotas*.

Page-table management: Enclave page tables are allocated using the physical page allocator in order to map virtual memory. PARTEE stores a reference to each root table; we assume a static maximum number of running processes and use a process-indexed table.

Partitioned slab allocation: In order to handle dynamic workloads and prevent DoS attacks, PARTEE's kernel objects are allocated in per-partition slabs. Because fragmentation could lead to availability issues, slabs of discontinuous physical pages are virtually mapped into PARTEE's kernel page table into per-partition virtual regions.

Virtual address space management: Using the partitioned slab allocator, PARTEE manages a data structure for each enclave's virtual address space (including heap, stack, and anonymous memory mappings). Each page fault, `mmap` system call, *etc.* that allocates a physical page for an enclave is subtracted from the partition's quota in the underlying calls to the page allocator.

Multicore, kernel threading, and locking: Each thread requires a kernel stack (allocated from the slab allocator). Since recursion or large stack frames could lead to overflows, we avoid using these. For simplicity, we opt for a non-preemptable TEE kernel design, where threads are running

to completion. Because some kernel data structures are inevitably shared between cores, we use per-layer MCS locks[153, 106] with carefully bounded critical sections to ensure lock-acquire ordering.

Asynchronous device I/O: PARTEE offers direct access to time-sensitive sensors and actuators via a system call interface; this avoids trusting the host OS for critical I/O. These devices typically use simple bus interfaces, *e.g.* UART, SPI, I2C, or CAN. To prevent long-running device I/O operations from blocking a core, we implemented a generic I/O queuing system. Outgoing I/O is queued up in each partition for each device, and the OS can optionally divide the bandwidth among the partitions.

Partitioned Enclave Authentication: PARTEE includes the standard asymmetric key authentication design used to authenticate enclave binaries; however, we improve on the protocol to protect against DoS attacks, as shown in Fig. 2.6. First, at boot time the root-of-trust is "branched" to create unique trust domain for each partition. This provides a separate authentication key for each partition, allowing new enclaves to come from various stakeholders with different privileges. Second, we alter the authentication protocol so that the prover (who wants to load an enclave) bears the loading and authentication costs of the enclave binary.

Normally, the enclave binary is hashed, and the hash is then signed or verified. We already prevent arbitrarily sized enclave loads by adding an additional signature of the ELF header, which is checked before the original hash is calculated. Then PARTEE uses a trusted user-space ELF loader and verifier which runs in the *prover's* partition using the prover's memory quota (note: the quota for the untrusted Host OS partition is a "Secure world memory" quota, unrelated to its ability to allocate regular memory). Only after authentication is the enclave charged to the destination partition.

2.6 PARTEE Communication

PARTEE uses a unique design based on asynchronous message passing over regions of shared memory. At a high level, PARTEE uses a DDS-like communication style, where enclaves publish and subscribe to named topics [165, 81] via a TEE-OS-mediated IPC protocol which prevents

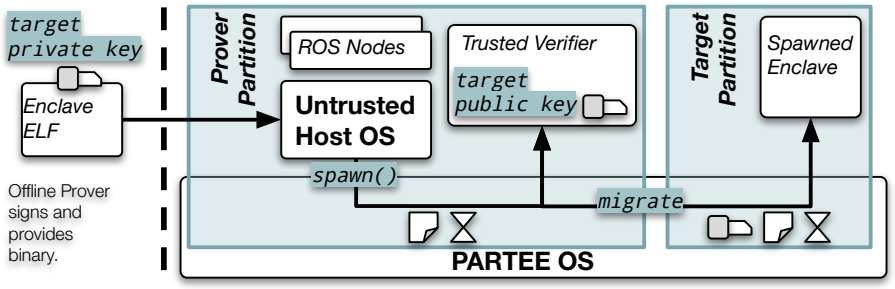

Figure 2.6: Resources needed to authenticate and spawn an enclave in a different partition must be provided by the "prover" partition (or parent) until the signature check is complete.

blocking attacks between any pair of partitions, protects against message corruption, and minimizes copying for broadcasts. Messages are passed via ring-buffer queues; thus, no rendezvous is required to send or receive data, and publishing will succeed with high probability without any need to busy loop. Our *Wait-Free Broadcasting Ring-Buffer* queue design is similar to well-known implementations, but tweaked slightly; the design is provided in §A.2.

Question 3: *How is lack of data handled?*

When IPC arrival timing is critical, the sender and receiver must trust each other to that extent. Yet, in many cases, PARTEE allows *graceful failure* in the absence of data streams. In other words, it still has guaranteed CPU time, IPC access, and device access which it can perform backup operations (landing or pulling to the roadside), or operate with reduced data. For example, consider an audio rendering task for an avionics device (like a Garmin GMA device) which mixes inputs from many sources: co-pilot microphones, flight-attendant microphones, music, radio, voice recognition, text-to-speech, and alerts from various subsystems. If one of the sources denies an audio signal, the system should play a timely audio notification to alert the pilot of a system error or just continue rendering the other sources without halting.

Question 4: *How are malicious messages handled?*

The power of PARTEE is that enclaves will have guaranteed execution time to sanitize incoming data and validate messages. We leave handling maliciously-crafted, faulty message content, and lack-of-messages to ongoing related work. Solutions to these are application-specific, and PARTEE is an OS-level design. For data coming from remove source, a common solution would be ap-

pending a message authentication code (or MAC) to each message and encrypting the contents—our implementation provides a library to help with this.

Identifying and mitigating bad data is a difficult problem, even for fully trusted software stacks [197]. Research is ongoing to investigate the problem of maliciously spoofed sensor data or ML model robustness attacks [191, 52, 104, 191, 98, 141]. The Simplex approach has been explored to validate complex controller outputs [189, 216, 228, 227, 139, 89, 155, 144]. Other approaches use physics modeling or hypothesis testing to identify anomalous spoofed data [197]. In some cases, data from trusted devices could be authenticated [214, 188]. For instance, if LiDAR data is published from an untrusted ROS node, it is possible to detect tampering via watermarking [43, 136, 147]; however, this does not deal with the potential for the spoofing. Ultimately, PARTEE is focused on the overall communication mechanism and ensuring that enclaves will have availability at the OS level—which is necessary to handle bad or absent incoming data with related work.

Question 5: *How is ring corruption handled?*

The protocol presented in §A.2 is designed to not trust the other readers and writers. It is resistant to corruption by keeping local copies of head and tail pointers, as well as bounding array indices. In the worst case, an adversary could corrupt messages from enclaves in the same partition; however, this is expected by design.

Topic Firewall: If the adversary gets access to a safety-critical topic, they could spoof malicious messages. Additionally, some contexts may have privacy-sensitive data, *e.g.* in HIPAA-compliant medical devices; here, the adversary should not be able to access privacy-critical topics.

We use the PARTEE Rules to allow developers to specify which partitions can access which topics with a "Topic Firewall" specification. The firewall specifies the rate at which each partition is allowed to publish to some topic and if the partition can read from some topic.

Question 6: *How are rates determined?*

Publish rates can be determined in many cases by the sensors, which produce data often at a fixed rate defined in a datasheet (or it can be overestimated experimentally). However, topics can be created at runtime with arbitrary sizes—as long as the partition's quota can support the queue

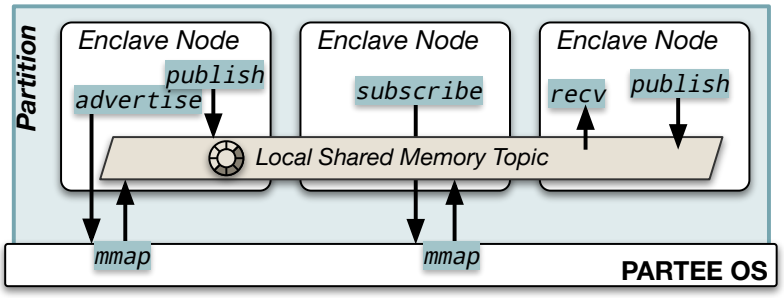

Figure 2.7: PARTEE's intra-partition publish-subscribe over pure shared memory incurs no TEE OS overhead for local message transfers.

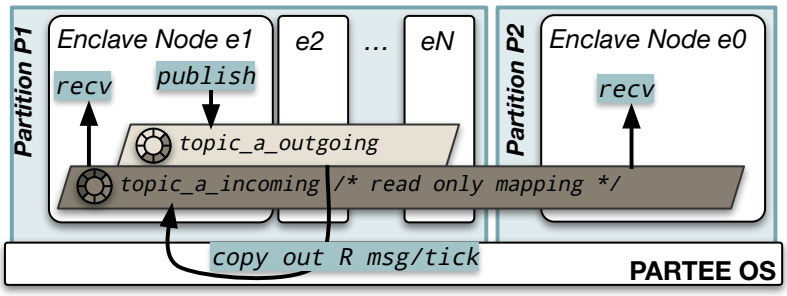

Figure 2.8: PARTEE's inter-partition publish-subscribe protocol enforces security efficiently by rate limiting with a single copy from partition-local outgoing to global incoming ring buffers

size. Thus, new topics can be scaled to runtime workloads as needed.

Intra-Partition Communication: Enclaves within a partition are safe to trust each other, so PARTEE uses direct shared memory for intra-partition, or *local*, message transfers, avoiding copying and system calls. The topic's shared memory can only be mapped into enclaves in that partition, ensuring confidentiality and integrity. The enclaves use PARTEE OS system calls to "advertise" to a topic (start publishing) and to "subscribe" (start subscribing). Once PARTEE validates the access, it maps the rings into the enclave's page tables. After this, IPC is entirely in polled shared memory, except for system calls to allow yielding, waiting, and signaling.

Inter-Partition Communication: Intra-partition communication assumes honest enclaves; however, *inter*-partition communication cannot assume this. A naive solution would be to have PARTEE OS copy messages into each subscriber; however, this would not scale well, requiring $O(\texttt{partitions} \cdot \texttt{ring\_size})$ copies.

Question 7: *How is broadcasting performance improved?*

PARTEE divides inter-partition communication into two parts: outgoing and incoming, as illustrated in Fig. 2.8. All enclaves in a partition share an outgoing ring per topic, and an additional read-only *global* incoming ring is mapped into all subscribing enclaves, regardless of partition. PARTEE OS copies messages from each partition's outgoing ring into the incoming ring. The copy allows PARTEE to rate limit publishes from each partition. Each message only needs to be copied once for inter-partition broadcasting; on the other hand, to broadcast on a system like OP-TEE requires copying the message to each receiver on invocation.

Ring Size Calculation: Using PARTEE OS to copy messages lets it enforce the rules of communication, protect message integrity, and prevent message flooding attacks. It uses the max rate `R` of messages per tick per CPU defined for each partition in the PARTEE rules to rate limit the number of copies it makes. Thus, the global incoming ring can be sized to be large enough so that all publishes will be visible to all partitions. *i.e.*, greater than the total messages that can be published over the `longest_period` of any partition:

$$\texttt{ncpu} \cdot \sum (\texttt{R}_i \cdot (\texttt{longest\_period}/\texttt{period}_i \cdot \texttt{budget}_i + \texttt{min}(\texttt{longest\_period} \bmod \texttt{period}_i, \texttt{budget}_i)))$$

Because we can calculate the incoming ring's size based on the enforced rate limits, no partition will be able to flood the ring before all other partitions have had a full budget of execution time. Messages are copied out on each timer tick, but an enclave can use either a `sync` or `yield` system call to trigger an early copy in latency sensitive contexts. PARTEE could also support sleeping on a topic, where the enclave will wake when only a message is copied into the Incoming Ring, but our implementation did not include this.

Data Distribution and Topic Registration: As shown in Fig. 2.9, we create a ROS proxy Linux process which interfaces with an (untrusted) Linux kernel module for PARTEE. The module creates file system entries to provide an interface to the TEE. Periodically, this proxy will execute `rostopic list` to get the current list of all ROS topics, then use the `ioctl()` calls to synchronize

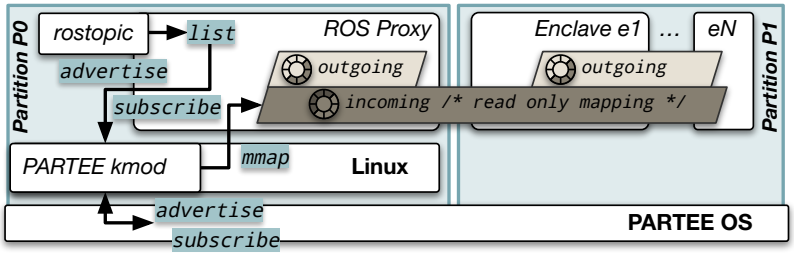

Figure 2.9: Diagram of how PARTEE integrates with ROS, automatically fetching topic lists from ROS and creating PARTEE topics, and *vice versa*, and publishing messages back and forth

topics. The proxy will then forward all messages between ROS PARTEE topics.

2.7 Case Study: Secure Partitioned Drone

We tested PARTEE on a drone software architecture (overviewed in Fig. 2.10) since it is a common COTS example of a time-sensitive system. We envision this design will encourage similar approaches for automotive, medical, or industrial settings, where a security vulnerability can have severe safety and economic consequences—especially when considering a large fleet. These systems should be reliable and safe; thus, we designed an obstacle-avoidance system on a search-and-rescue drone, derived from relevant drone architectures [63, 186, 37, 224, 156]. The drone uses an off-the-shelf quad-copter frame with a Pixhawk CubePilot Orange autopilot device and a Raspberry Pi4B companion computer.[2] The companion computer is the focus of our experiments where Linux is the host OS. We set up two scenarios: one with all tasks running on the host OS, and one where PARTEE runs as the EL3 firmware and S-EL1 TrustZone TEE OS with some tasks as enclaves (see §A.1.1).

The ground-control (enclave) task communicates over a TLS connection to receive high-level mission objectives and send sensor logs. We classify this enclave as *mission critical* because it handles sensitive mission data (suppose compromise has severe consequences [17]). This enclave spawns children to handle concurrent network connections.

The MAVLink-gateway (enclave) task interfaces with the autopilot's MAVLink serial connec-

[2]Since the Pi4B does not have memory bus security, we instrumented SeKVM [126, 127] (with no guest) to ensure isolation, see §A.1.

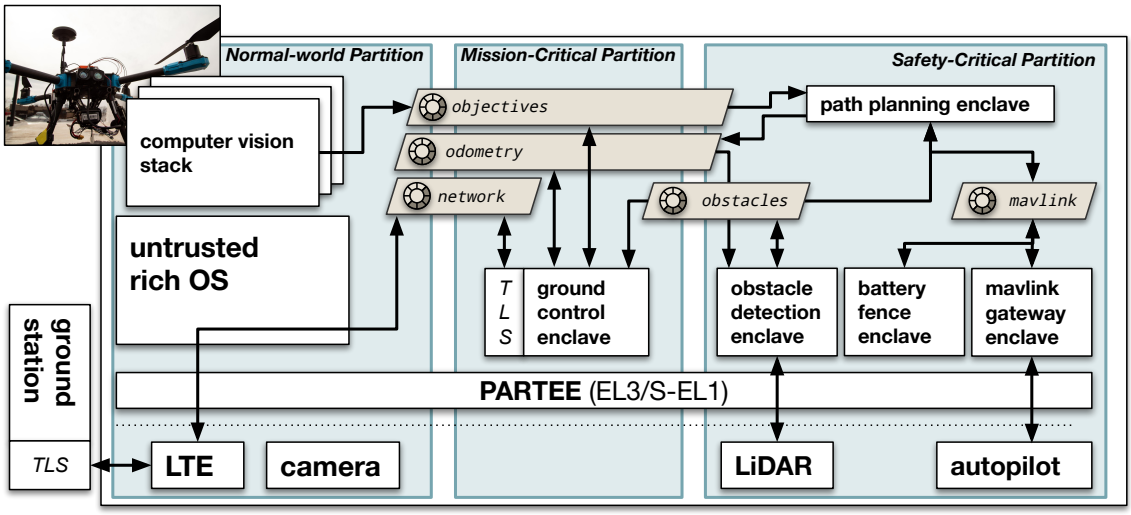

Figure 2.10: Overview of highly-secure PARTEE drone implementation where control tasks are divided into partitioned enclave processes. Enclaves publish and subscribe to isolated ROS-compatible topics to transfer data, and they interface with their partition's devices—thereby protecting availability, integrity, and confidentiality for enclaves and IPC.

tion [116] (via a PARTEE driver), and proxies the packets. This interface directly controls the drone and provides odometry data. We implemented packet validation and filtering, so invalid or disallowed packets are dropped. This task is *safety critical* since it can arbitrarily control the drone's movements.

The obstacle-detection (enclave) task reads a LiDAR sensor in order to detect obstacles (and publish detections). It also reads odometry data in order to smooth noise from the LiDAR from frame-to-frame and coalesce detections. We classify this as *safety critical* because if it fails to detect obstacles in time the drone will crash.

The path-planner (enclave) task gets objectives and obstacles and intelligently updates an ongoing search task. We implemented a rapidly-exploring random tree (RRT) algorithm for the planner, allowing it to traverse around the obstacles while searching new areas. The planned path is sent to the autopilot via the MAVLINK gateway (over a "mavlink" topic). This enclave is also safety critical.

The battery-fence (enclave) task reads the battery levels and initiates a return-to-landing command to the autopilot based on the charge and distance to the home base.

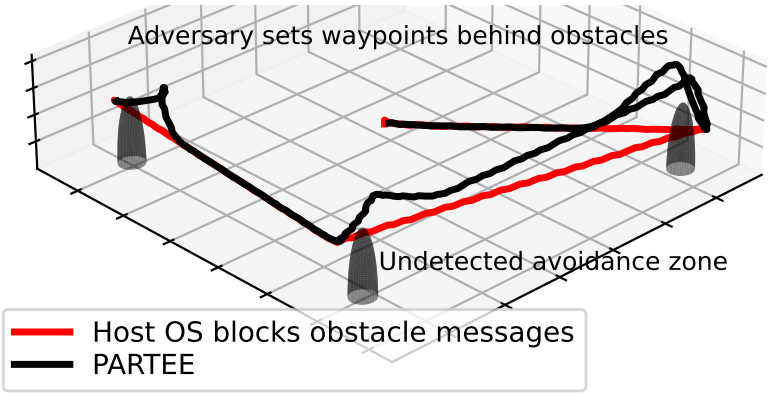

Figure 2.11: Comparison of drone flight paths: Insecure version where the host OS blocks obstacle-avoidance IPC, and a secure version using PARTEE, which guarantees IPC delivery. The insecure drone does not detect the obstacle and does not avoid flying too close.

The computer vision tasks represent an objective identifying ML model, which reads from a downward-facing camera to find rescue targets (for our implementation these tasks are shims) and publish them as objectives. Suppose this model requires complex third-party libraries and software, so it must be run directly on Linux.

Obstacle avoidance under adversarial conditions: We created three partitions: *Normal-World*, *Mission-Critical*, and *Safety-Critical* based on the classifications above. The autopilot and LiDAR devices are connected to the companion computer via UART serial. We configured the PARTEE rules so that only the Safety-critical partition could access them. Additionally, the PARTEE Topic Firewall (access control rules) are visually shown in Fig. 2.10. The adversary, $\mathcal{A}$, has two modes: In Mode 1, $\mathcal{A}$ has root access to Linux. In Mode 2, $\mathcal{A}$ has additionally compromised the ground-control enclave, $e_{\mathcal{A}}$, and can run any code there. The victim, $e_{\mathcal{V}}$, could be any Safety-critical enclave.

Fig. 3.7 shows two flight traces. In the experiment shown with a red trace, we programmed the host OS to block incoming messages from the obstacle-avoidance task. The host OS also sends malicious waypoints to try to cause the drone to fly dangerously (programmed to nearly hit the obstacle). The result is that the drone did not detect obstacles, flying in an unsafe way and not performing avoidance. The black trace shows an experiment running exactly the same source code, recompiled to run inside PARTEE enclaves. The *path-planner* and *obstacle-avoidance*

TCB (SLOC)	EL3	S-EL1	S-EL0	**Totals**
PARTEE	<1,000	35,311	6,997	43,308
OP-TEE	46,559	86,500	9,787	142,846

Table 2.1: Source line counts for the PARTEE and OP-TEE trusted computing bases.

enclaves continue to run, regardless of the actions of Linux. We ran a number of other tests on the drone to validate our implementation, which will be documented in our artifact.

Protecting against malicious data or DoS: Considerations around malicious data and DoS are foundational to this drone's design. With our Mode 1 Adversary (with just root access), it can publish arbitrary objectives or to corrupt the Normal world's outgoing objectives ring. Additionally, it can do the same for the network topic. By design, ring corruption will result in "junk" messages. We easily mitigate the impact of this attack by validating objective messages. The path-planner enclave must bounds check coordinates and ensure that they are within the geofence; thus, the adversary can at-best make false-positive objectives. Objectives between the ground-control enclave and path-planning enclave cannot be overwritten until the path-planner has executed, due to the protocol in §2.6. Published network traffic cannot be read or forged due to the use of TLS. With Mode 2, again $\mathcal{A}$ can make false-positive objectives. However, because the obstacle-avoidance system is isolated in another partition, the adversary still cannot make the drone crash, even with malicious objectives. If objectives are DoS'ed the drone will return home after visiting all known search area objectives.

2.8 Security Analysis

PARTEE aims to reduces the TCB of critical CPS applications. To assess this, we measured the TEE OS and firmware source size needed for the Raspberry Pi4B implementation, for both PARTEE and OP-TEE, shown in Table 2.1. Our combined firmware/TEE OS design and simplified TEE interface is able to further reduce the TCB, even over OP-TEE. To further assess PARTEE, below we perform a case-by-case analysis based on the goals defined in §2.4.

Goal 1. Guaranteed physical memory reservations: Physical memory can be allocated through several interfaces; however, due to the encapsulation of PARTEE's design, all physical page allo-

cations must occur through the physical-memory allocator which takes a partition ID argument. The allocator tracks each partition's quota, decrementing as pages are allocated and rejecting once the quota is zero. Thus, the enclaves cannot over consume memory.

Goal 2. TEE resource and service availability: Through building PARTEE's, we constructed a list of all TEE OS exposed interface points and TEE OS objects, summarized in Fig. 2.5. The non-trival system calls available to enclaves are for memory mapping (`mmap`, `munmap`, `mremap`, `brk`), IPC (`advertise`, `subscribe`, `sync`), process control (`yield`, `sleep`, `spawn`, `exit`), device I/O (`dev_control`, `dev_-open`, `dev_push`, `dev_pop`, `dev_waitany`, `dev_waitall`), and system management (`shutdown`). The memory mapping systems calls interface with the virtual-memory layer, which manages the address space and ultimately calls the physical memory layer. The number of mappings can increase map time, but it is bounded by the maximum number of pages a partition is allowed. `advertise` and `subscribe` lookup and manipulate topic data structures. Because topics can be looked up by string names, we use a prefix tree to prevent topic name flooding attacks. Topic objects are allocated by the partition that creates them. Spawn uses the algorithm specified in §2.5.1. A new enclave is only admitted into the scheduler if it will not exceed the partition's maximum budget. Device I/O creates queues using partitioned slab allocation, and the operations are constant time. Enclaves can only power off the system if shutdown access is granted in the PARTEE Rules. The normal world is given a limited subset of these system calls as monitor calls.

Goal 3. Wait-free publish and subscribe IPC: By inspection of the algorithms in §A.2, we can see that there is no unbounded loops in the publishing or subscribing protocol. Loop bounds for reading, which are based on untrusted values read from shared memory, are bounded to a maximum of the length of the ring buffer. Even if a thread is interrupted and context-switched out, other threads will be able to communicate as the size of the Data Ring must be greater than the maximum number of publishing threads. Because of this, simultaneous access to the data structure will yield unique indices into the Data Ring. Thus, a publisher will not lose access to the data structure due to other publishers.

Goal 4. Guaranteed Correct Message Delivery: Upon publishing, the kernel will copy this message from the Outgoing to the Incoming Ring (see §2.6) at the next timer tick or `sync`, which is

mapped as read-only to user-space. Because the kernel will enforce rate limits on how many messages it will copy per partition, per topic, per tick, the Incoming Ring will have a maximum number of messages that could be written over any time period t: $M(t)$. Additionally, the scheduler will execute each partition for exactly its full budget each period; thus, there exists a maximum time delta T between which all subscribing partitions will be scheduled for a full budget. PARTEE sizes the Incoming Ring is sized to be larger than $M(T)$. Therefore, no messages will be overwritten before all have a chance to read them.

Goal 5. Flexible IPC access control: The kernel validates access to topics according to the Rules on each `advertise` and `subscribe` call. If the topic exists, it will have a set of rules that apply; otherwise, it will use catch-all rules. If the enclave is in a partition without access to the topic, the system call will correctly fail.

2.9 Performance Evaluation

We aim to answer the following performance questions for PARTEE: (**Q1**) What is the baseline overhead cost of PARTEE on the host OS? (**Q2**) What is the performance overhead of a PARTEE enclave? (**Q3**) What is the cause of latency for publishing between enclaves? (**Q4**) How is latency affected by PARTEE's intra-partition optimization?

Q1: Host OS Overhead. To evaluate the effect of PARTEE on Normal world performance, we use both UnixBench [194] and MiBench [83] suites. This experiment measures impacts caused by preempting the Normal world to run PARTEE. We ran these benchmark on our baseline system, and on PARTEE with just the normal world partition.

Fig. 2.12 shows the results. Most benchmarks were not affected in a significant way, and measurement noise appears to be the cause of benchmarking discrepancies. This is expected as we measure the world switch time at about 5 μs, which is about 0.5% of the timer interval. We chose this setup so we could assess the impacts of interrupting Linux on the caches; however, using the PARTEE rules the CPU budget of the Normal world on each core can be capped for security purposes. We tested this, and the benchmark performance proportionally degrades, as one would expect.

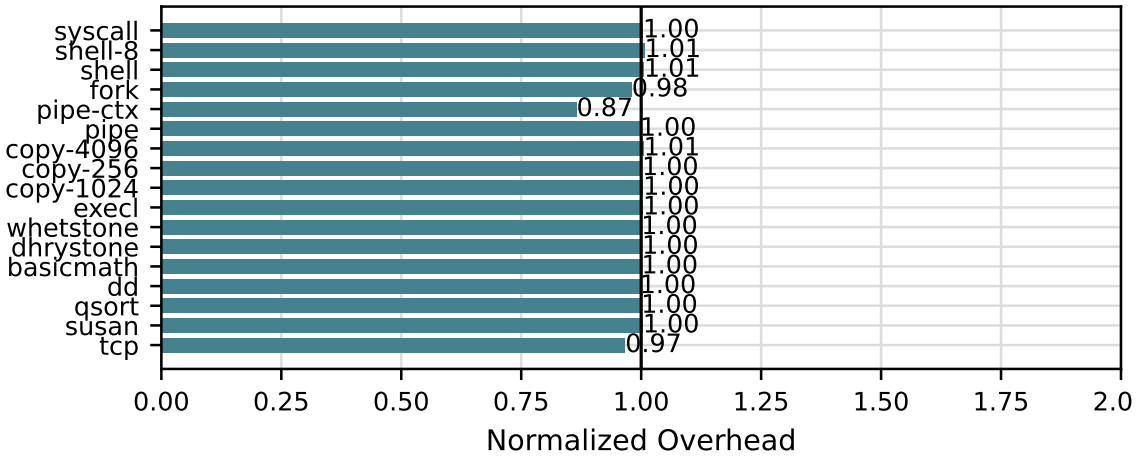

Figure 2.12: **Host OS Overhead:** Overheads of PARTEE on UnixBench and MiBench applications executed on Linux

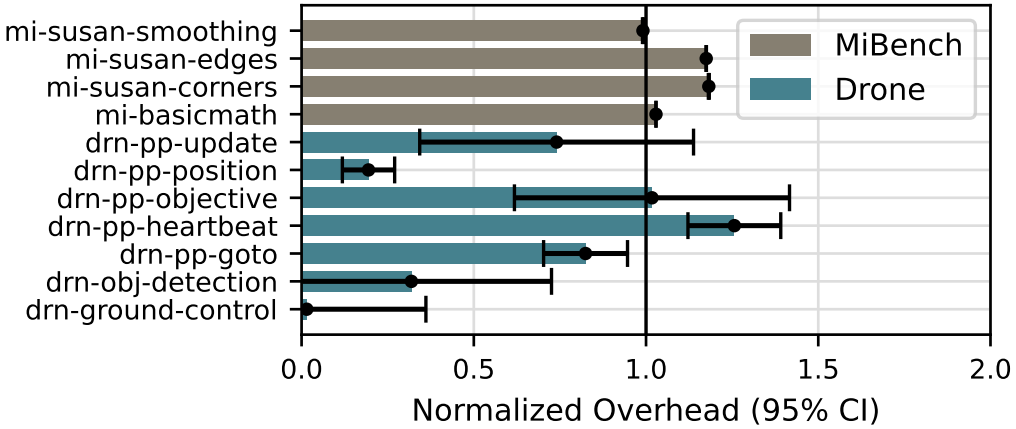

Figure 2.13: **Estimates of enclave overhead (smaller is better):** Benchmarks run in enclaves demonstrate overheads for memory-intensive operations, but communication-intensive drone functions typically had improved performance over the host OS due to PARTEE's IPC.

Q2: Enclave Overhead. We took two approaches to measuring enclave overhead. First we measured execution times of key enclaves of our drone implementation: specifically, we looked at the core functionality of the path planner, object detection, and ground control enclaves. Second, we took major automotive benchmarks from MiBench [83] and ported them to each be PARTEE enclaves. The results are shown in Figure 2.13.

On the drone, we measured several key macro functions, prefixed by drn in the graph. These include some 3D-spatial calculations, task-planning updates, and communication with other enclaves or processes. Even though these functions have higher variances, we observed that PAR-

TEE could improve on native performance, for several communication-heavy operations. This is due to PARTEE's use of shared memory with little to no copying, and the underlying transport used in native IPC makes a significant impact on performance.

For MiBench, we measured the runtime of the core routines used for image processing and basic computation: susan-corners, susan-edges, susan-smoothing, and basicmath. These stress our LibC implementation, virtual-memory management, and scheduling overheads. For applications that use heavy memory allocation, there was room for improvement, potentially in reducing the number of page faults due to our lazy memory-mapping strategy.

Q3 and Q4: Message Latency. Our end-to-end message transfer time is shown in Fig. 2.14. In this experiment, we measure time starting from when the publisher acquires a frame, copies a message into it, and publishes it to when the subscriber receives it; thus for realism, all publishes include at least one copy. When comparing Intra-Process Sync with Local we observe that the overhead due to the shared-memory protocol is minimal compared to the overheads of system calls (the sync call triggers the kernel to copy outgoing to incoming). Additionally, comparing the Intra- and Inter-Partition curves, we can see that as the message size increases, our local topic optimization scales better due to reduced copying.

2.10 Related Work

Static Partitioning and TZ-Assisted Hypervisors: Some high-assurance CPS employ temporal and spatial partitioning to isolate mixed-criticality workloads [178, 177, 33], protecting one or more real-time OS from a rich untrusted OS. Solutions such as static partitioning hypervisors [149, 222, 101, 174, 150, 19] or TrustZone-assisted designs [183, 184, 185, 108, 167, 168, 148, 166, 143, 162, 170, 64, 100] offer isolation but come with limitations for modern robotics: (1) *Limited scaling and granularity.* Each partition needs its own instance of a VM; thus, as the number of partitions increases, performance will degrade due to various VM-switch trade offs [90, 31]. For example, Jailhouse [174] trades the cost of VM exits for the usage of an entire core, 25% of the Raspberry Pi4B's CPU power, and μRTZVisor [148] requires a full cache flush on world switches. (2) *Over-privileged tasks.* Many RTOSes are not designed for security, and will run all

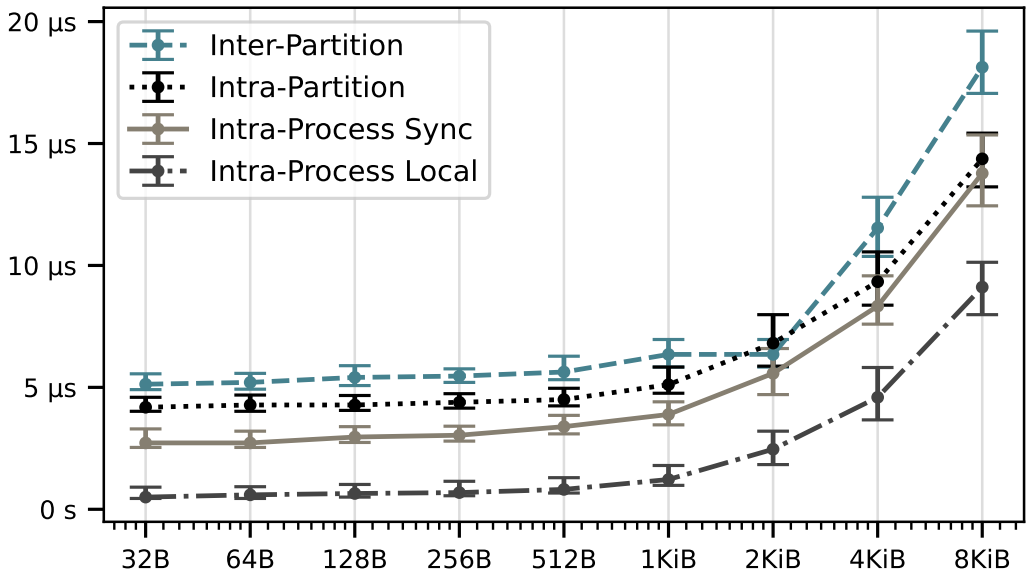

Figure 2.14: **Single Message Latency:** Comparison of PARTEE's end-to-end latency for inter-partition, intra-partition, and intra-process scenarios; PARTEE's local-topic optimization slows scaling due to copying overheads (n=128).

tasks in a single address space in kernel mode. For the ARM TrustZone, a faulty or hostile task could then own the entire system [128, 201, 42, 35]. (2) *Engineering overheads.* Tasks in different partitions must use inter-VM communication [187, 162, 203]. Hence, for ROS-based systems, ad-hoc protocols are necessary to transport published data across VMs and real-time OS interfaces. Ultimately, partitioning can provide isolation, but these hypervisors can also suffer from steep performance and engineering overheads.

SMACCM: The SMACCM architecture [212, 48] for the DARPA HACMS project, aims to build a highly-secure drone with from many layer of formally verified software. If we focus on SMACCM's seL4-based [111] software architecture for the companion computer, it provides strong isolation of enclave-like seL4 drone control tasks from Linux guest OS. In our view, PARTEE could theoretically be implemented on top of seL4, as a service which manages shared memory, and other resources for sets of enclaves. The upper PARTEE layers would need to ported to user-space in order to support dynamic spawning of enclaves, publish-subscribe IPC and topic creation, asynchronous device I/O, and other essential features. Without PARTEE, this architecture is likely vulnerable to IPC blocking attacks for seL4's IPC mechanisms.

Other DoS defenses: For hard real-time contexts, the TEE OS may need to account for memory bandwidth and cache delay attacks. In these cases, cache-partitioning and MemGuard-style approaches [230, 24, 229, 71] could be deployed in addition to PARTEE, using hardware performance counters to budget cache and memory bandwidth, preventing temporary delays due to limited microarchitectural queues. Future work could explore implementing PARTEE on the microarchitectural partitioning version of seL4 [111, 71].

Secure I/O: One area for future research for PARTEE is improving enclave access to dedicated devices. Recent work on the Linux Device Runtime (LDR) [225], demonstrated a new approach for supporting more complex devices in the TrustZone by reusing Linux drivers. PARTEE potentially pairs well with LDR, and could pave the way for enclaves with available access to USB cameras and other rich devices. MyTEE [85] and RT-TEE [218], and others [82, 105] also have potential solutions that complement PARTEE OS to enable more complex I/O.

Protections against malicious enclaves: Recent works vTZ[95], TEEV [128], 3rdParTEE [96], ReZone [42] have aimed at sand boxing insecure enclaves; however, the challenge still remains how to isolate untrusted enclaves, untrusted VMs, and an untrusted host OS with total availability. To support more complex software architectures, *i.e.* multiple VMs that each want to launch enclaves, PARTEE would need to be extended. Potentially, the ARM CCA provides additional tools that could be used to achieve this [118].

Alternative Hardware Solutions: A few proposals targeting microcontrollers have designed hardware mechanisms for isolating critical software [151, 3, 4] while protecting their availability. Other works, TyTAN [29], TrustLite [113], ERTOS [206], and [122], delegate scheduling to the untrusted OS; thus, they do not provide availability.

Software-Based Attestation: New work [202] has shown how to perform pure software-based attestation, establishing a root of trust using a nearly perfectly optimized hashing function. At the cost of a dedicated core for attestation, a small embedded device with no other mechanisms to create enclaves can implement a reactive security system, sending continuous attestation reports. While this is a practical layer of security, because it is purely reactive, it cannot meet the needs of a time-sensitive and safety-critical CPS. By the time the device resets from an attack detection,

the damage would already be done—whereas PARTEE aims to allow critical software to continue executing securely until the system is in a safe state.

Dedicated Security Cores and Coprocessors: Previous work proposed the use of dedicated core(s) in a multicore system to act as a security monitor or dedicated enclave core. SecureCore [228] and Jailhouse [174] use nested paging to create memory protection. All I/O can be routed to the correct core using this approach allowing a trusted IO path. Unfortunately, SecureCore's and Jailhouse's use of an extra layer of address translation will likely have detrimental performance impacts on the translation cache [90]. Sanctuary [30] executes enclaves on a dedicated core and avoids the need for nested paging, but still relies on the Normal world OS to schedule enclaves. While separating security domains onto different cores offers potential temporal isolation, this approach comes at the cost of full resource utilization.

Likewise, another design might use a security coprocessor instead of a secure core or the Secure world. Some SoCs also provide real-time coprocessors for this purpose; however, it is usually not possible to modify a SoC to add a coprocessor. Adding another physical microchip adds additional, weight, power, and engineering costs.

Relevant CPS-specific security mechanisms: Other CPS-specific research projects have shown ways to isolate and share devices using the TrustZone or virtualization. The design of PROTC [139] shows how to construct a real-time drone while protecting access to some peripheral devices. Another work presents the design of Contego-TEE [89] which attempts to protect against malicious or dangerous control of actuators. We view these works as orthogonal and complementary to our approach of supporting general purpose, real-time enclaves and enclave communication for CPS. The VirtualDrone [227] architecture uses hardware-assisted virtualization to effectively create a TrustZone-like separation of security domains. This work uses a KVM-style hypervisor to isolate a guest OS which runs a complex control system for a drone. The I/O is routed through an I/O proxy and safety controller application running on the host OS.

2.11 Conclusion

Given the complex software and hardware needed to for robots to work, and the safety implications for their failure, we argued that critical applications be protected in enclaves. Unfortunately, past work on real-time enclaves has several major challenges around blocking IPC and partitioned software architectures. We proposed, and implemented, PARTEE a novel design for real-time enclaves which can communicate securely across fine-grained partitions.

Chapter 3

Ringmaster: How time-sensitive TrustZone enclaves can juggle untrusted OS services practically

Many safety-critical systems require 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 service is denied by the untrusted OS, 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 a unmanned aerial vehicle, Ringmaster achieved nearly 1GiB/sec of data into enclave on a Raspberry Pi4b, almost zero throughput overhead compared to non-enclave tasks. [1]

[1]This chapter is adapted from an in-preparation conference paper by Richard Habeeb, Hao Chen, Man-Ki Yoon, Ben Cifu, and Zhong Shao.

3.1 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 [115, 44, 28, 68, 172, 199, 224, 107, 156, 27]. 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 [54], 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 [36, 157], increasing exposure to threats. These systems frequently use system-on-a-chip (SoC) designs [102, 33, 181], co-locating *time-sensitive* programs with less critical ones, which introduces new security risks [88]. Despite prior research efforts, attacks continue to emerge [69, 124, 176, 160, 56, 38, 220, 186, 198, 26, 221, 25].

Trusted Execution Environments (TEEs) have been explored to isolate software in *enclaves* from a privileged host operating system [180], 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 [8], despite many years of quality research on TEEs [169], with numerous strong defense designs for real-time systems [183, 184, 185, 108, 167, 168, 148, 166, 143, 140, 162, 64, 170, 100, 218, 163]; 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 cryptographic operations [63].

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 which 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 [204, 47, 94, 53, 132, 22, 23, 12, 164, 80, 211, 193, 190, 1, 215] 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.

Enclaves with availability protection lack comprehensive OS support Designs with availability protections [183, 184, 185, 108, 167, 168, 148, 166, 143, 66, 212, 48, 140, 162, 64, 170, 3, 206, 100, 218, 163, 4] all have very limited OS services. This limits the scope of enclave functionality. By design, ARM TrustZone enclaves [8, 169] 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 [179, 162, 187, 203, 48, 150]. Thus, small changes to tasks require substantial effort, making this approach unappealing in practice.

We propose *Ringmaster,* a new design to enabling time-sensitive enclaves with fast, non-blocking access to Linux system calls. Ringmaster leverages the `io_uring` framework [49, 50] 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.

From a security perspective, the challenge of Ringmaster is to discover how to give enclaves

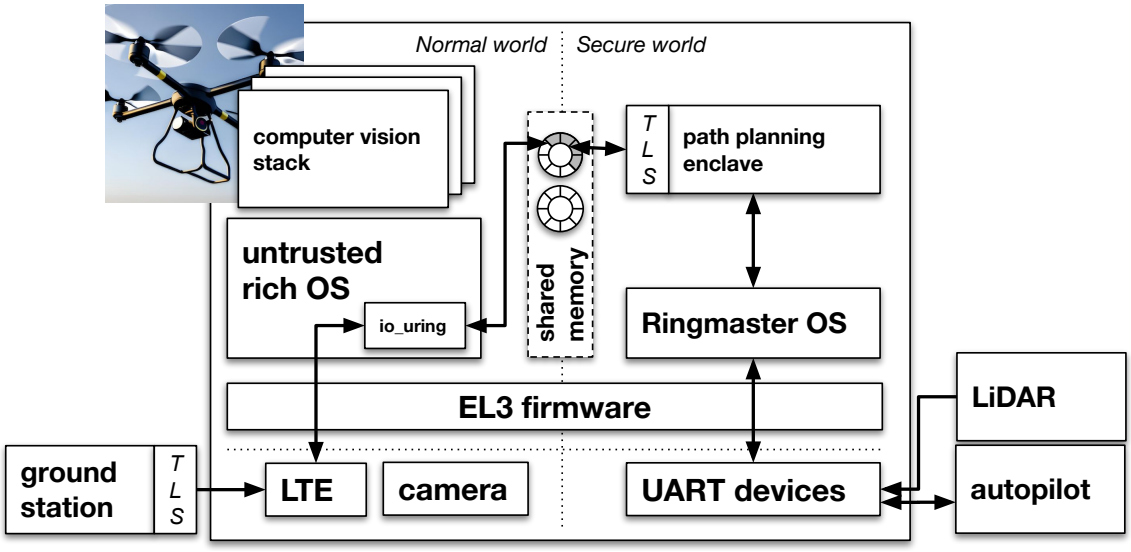

Figure 3.1: Example of Ringmaster on a drone with a flight controller enclave communicating over encrypted asynchronous `io_uring` operations with a remote operator; if the untrusted OS denies network service, the enclave will continue to stabilize and direct the drone.

clear and safe acce

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 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. We implemented Ringmaster on the ARM TrustZone as a TEE OS, however, the design likely generalizes to other hardware platforms or isolation primitives.

We evaluate Ringmaster by building an autonomous quadcopter (Fig. 3.1) with a time-sensitive path-planning enclave that communicates with a ground station using Linux network sockets. 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 adversarial conditions. The flight-control 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. 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 would massively reduce the critical task TCB.

Legacy POSIX (Portable Operating System Interface) applications can benefit from Ringmaster as well. Using timeouts and signals, such a program can be minimally updated to protect it from being blocked waiting infinitely. 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 [80].

Contributions:

- We design a new approach for initiating Linux system calls from independently managed enclaves with a zero-copy mechanism for transferring large arguments (§3.4). This includes handling of power and thermal management, mitigating the potential impacts of polling (§3.4.3).
- 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 (§3.5.1).
- We demonstrate a design that supports minimally modified POSIX applications with non-starvation guarantees (§3.6). Experimental results demonstrate comparable or better latency for system calls than past approaches for unmodified enclaves (§3.9.3).
- We deliver a prototype implementation on the Raspberry Pi4B, a demonstration on a drone platform (§3.8.1), and evaluations using unmodified GNU coreutils applications (§3.9.3).

Our experiments showcase high throughput I/O into an enclave, achieving full saturation of the Pi's gigabit Ethernet port and nearly 1GiB/s file system read and write speeds.[2]

3.2 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, and drivers, as well as third-party software.

While it may seem paradoxical to seek services from an untrusted OS, this is common in modern CPS, IoT devices, and robots. For example, drones often use the MAVLink protocol [116] to transmit waypoints from ground stations or companion computers [6, 171] 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 [92] while continuing real-time obstacle avoidance. For instance, a number of recent security analyses [160, 220, 25, 221, 26] reveal that the Tesla Autopilot system is directly connected to an Ethernet switch with internet connectivity.

In both cases, time-sensitive systems depend on OS services (*e.g.* from Automotive Grade Linux [205] or Real-Time Ubuntu [40] 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 §3.4.4, and comparisons with other related work in §3.10.

[2] We plan on making our artifact public at the time of publication.

3.3 Models & Assumptions

Our models for enclaves, platforms, and adversaries build upon Subramanyan et al.'s definitions [200], extending them to address time-sensitivity and availability.

3.3.1 Hardware Model

This work targets hardware platforms that provide privilege levels above kernel mode with a memory-management unit, although it is not tied to a specific architecture. The hardware requirements include:

- **Memory sharing and isolation.** Programmable control of OS memory access, for example, a TrustZone Address-Space Controller (TZASC) [10], RISC-V's physical-memory protection (PMP), or nested page tables (NPT).[3]
- **Device isolation.** Configurable OS access to memory-mapped I/O (MMIO) and secure interrupt delivery *e.g.* using ARM's Generic Interrupt Controller (GIC)[9].
- **A dedicated timer.** An unmaskable timer interrupt for preemption and scheduling of the OS and enclaves.
- **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 lack of programmable enclave page table management. Though our implementation is based on the ARMv8-A TrustZone, a hypervisor-based isolation approach, *e.g.* SMACCM [212, 48], or a RISC-V approach, as used by Keystone and ERTOS [122, 206], also fits our model.

3.3.2 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

[3] This potentially includes ARMv9's Granule Protection Table. [11].

privileges and can read or write to any physical 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.

Game 1: Timeliness. Suppose a victim enclave, *Alice* ($\mathcal{A}$), must write a valid message to a serial device, in a tight time bound $\mathcal{T}$. However, the message is stored by $\mathcal{E}$. Upon request, $\mathcal{E}$ chooses to send either a valid or invalid message. $\mathcal{A}$ has a function $f(m)$ that decides if a message is valid or not, 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 $\mathcal{T}/2$ time, even with the maximum number of cache misses, page faults, branch mispredictions, or other microarchitectural delays. In this game, 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 before $\mathcal{T}/2$, $\mathcal{A}$ will write it.

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.

3.3.3 Assumptions

Responses to rich I/O starvation or corruption are out of scope. Lack of data is a problem, and solutions are ultimately application-specific. The power of Ringmaster is that *enclaves can respond* to this scenario. Additionally, even without guaranteed availability for OS services, having access is often still advantageous (see §3.2). Likewise, malicious I/O should be handled via encryption, authentication, or an application-specific method. We discuss further solutions in §3.4.4.

System overhead is negligible. The overhead of Ringmaster 's system calls and interrupt handlers is negligible compared to scheduler timing. Additionally, we assume internal CPU operations like cache accesses, page-table walks and branch predictions are negligible for this security model. Mitigating contention for such microarchitectural resources (also including memory bandwidth) are being addressed in complementary work [229, 230, 234, 187].

Physical and microarchitectural side-channels are out of scope. We assume that the adversary does not have physical access to the SoC, and cannot sniff the memory or device buses, perform cold-boot attacks [84], power-analysis attacks [112], *etc.*. Microarchitectural side-channel attacks like ARMageddon, *etc.* [134, 232], are handled using emerging techniques [233, 72].

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.

3.4 Asynchronous System Call Design

Here we discuss Ringmaster's novel design for making arbitrary I/O system calls into Linux from an enclave.

Background on `io_uring`. Modern Linux kernels have an interesting system called `io_uring` [49, 50] which allows user-processes to make asynchronous I/O-related system calls. This mechanism, consisting of two ring-buffer queues mapped into user-space, has been demonstrated to support incredible rates of 122M I/O operations per second (around 60 GiB/s bandwidth) [16] through parallelism and bypassing trap overheads. 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.

3.4.1 Making System Calls from an Enclave

For Ringmaster, `io_uring` provides an exciting opportunity to change the hierarchical relationship between process and Linux kernel. If a process is enclaved in the TrustZone, in a VM, or on a co-processor, and 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.

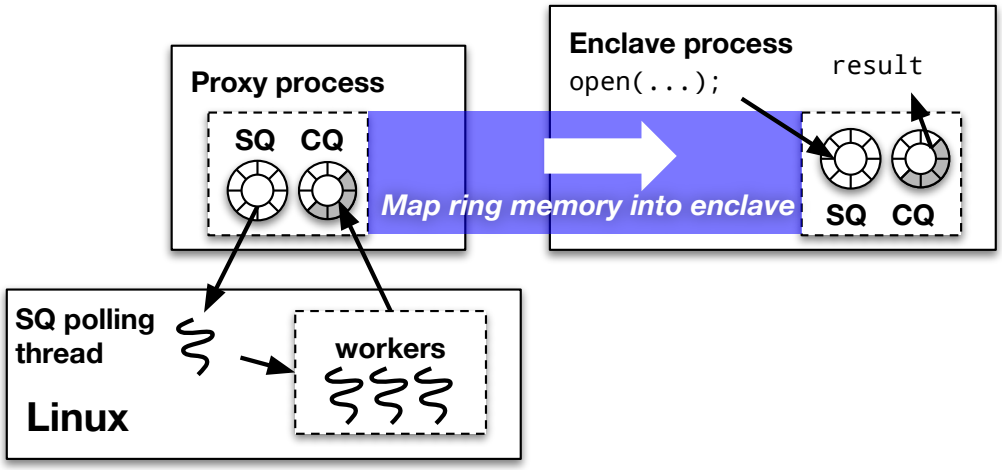

Figure 3.2: Illustration of how `io_uring` SQ and CQ memory could be mapped into an enclave, giving it access to I/O

Continuing with the TrustZone example, if the SQ and CQ are mapped into a TrustZone process, as shown in Fig. 3.2, then it could perform arbitrary I/O system calls *as if Linux was a remote file system and I/O service*. This is the basis of Ringmaster's design.

For Ringmaster, enclaves are processes running in the Secure world or Realm of the TrustZone or CCA, *etc.*; they are managed by a trusted OS, which handles scheduling, virtual memory, and other essential OS tasks. We will call this OS "Ringmaster OS" in this paper. As we will discuss further in §3.8.2, this is how enclaves can have 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. 3.3:

1. Each enclave has a proxy process 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. Once the SQE is received, Linux registers the queues' physical memory address with Ringmaster OS. Addi-

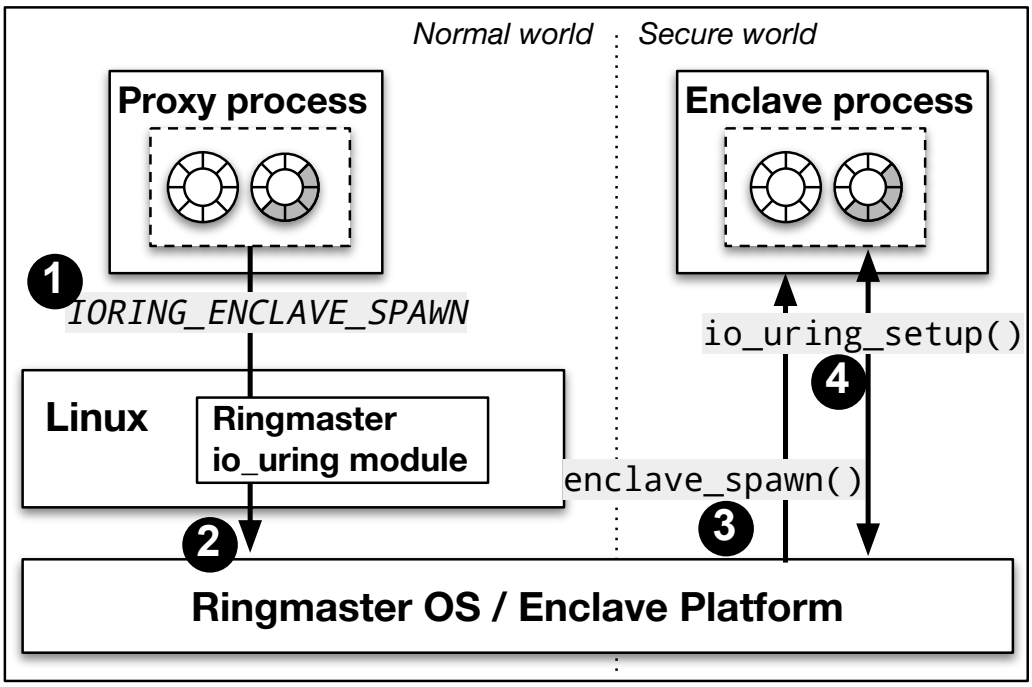

Figure 3.3: Diagram of Ringmaster's process for registering and mapping `io_uring` memory for an enclave

tionally, Linux optionally notifies Ringmaster OS to spawn a new enclave at this time, if it hasn't been started already.

3. If the enclave isn't 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.

3.4.2 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. Thus,

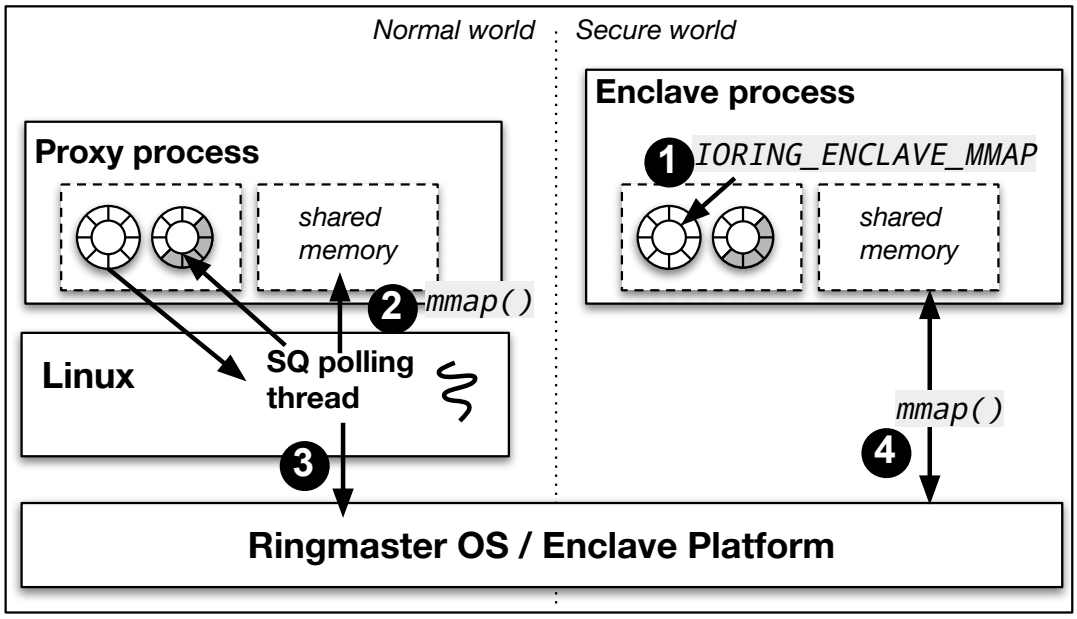

Figure 3.4: Diagram of Ringmaster's process for registering and mapping generic shared memory for an enclave

Ringmaster includes a two-part mechanism for establishing translatable regions of shared memory between proxy and enclave.

The first part of the mechanism is establishing shared memory between enclave and proxy via the following registration process, shown in Fig. 3.4:

1. The enclave first adds a new `IORING_OP_ENCLAVE_MMAP` entry to the SQ, which specifies the size of the request and 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.

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

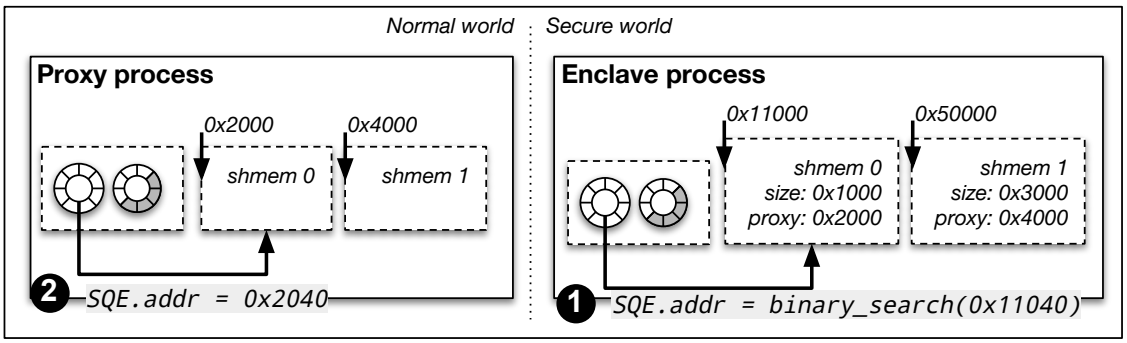

Figure 3.5: Diagram of Ringmaster's process for translating pointers into shared memory where address `0x11040` in the enclave is translated to `0x2040` in the proxy's address space

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 an ordered list of shared memory blocks, with the enclave's address of the memory, the proxy's address, and the block size—as shown in Fig. 3.5. A binary search with the pointer argument on the list allows the pointer to be translated from enclave to proxy. 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 everything 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 will see the SQE in the proxy's queue, and will service the `open` system call; once it is complete the file descriptor will be created in the proxy process, and the descriptor number will be put in the CQ as the return value.
5. The enclave now has opened the file and can use the descriptor number for further operations.

3.4.3 Power Management & Wake-Up Signals

Devices which use battery power or have limited thermal management typically need to allow cores to sleep periodically, *e.g.* using ARM's wait-for-interrupt (`WFI`) instruction; however, this would not be possible if Ringmaster entirely relied on polling the `io_uring` queues. Normal systems manage SQ-polling thread's energy consumption impact by setting a timeout, after which the thread will sleep. Once this happens the program needs to use an `io_uring_enter()` system call to wake up the thread again. For Ringmaster, once the thread sleeps the enclave would lose the ability to make further `io_uring` operations. Furthermore, we cannot directly use Overshadow-style [47, 80] forwarding because we do not want to block the enclave waiting for Linux. Our solution is twofold: (1) Linux SQ-polling thread wake-up notifications, and (2) leveraging a real-time scheduler.

We first designed a synchronous `io_uring_enter()` Ringmaster OS system call. This system call makes Ringmaster OS create a software-generated interrupt (with some information about the calling enclave written into a shared memory queue). The interrupt will remain masked, however, so control will not transfer immediately transfer to Linux; instead, it will return back to the enclave until it yields or its budget is exhausted. Once Linux receives the interrupt, the Ringmaster kernel module's handler will now forward the system call on behalf of the enclave (using techniques described by Overshadow [47, 80])—waking up the SQ-polling thread.

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 §3.7.2, 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 any `io_uring` completions, perform work, create submissions, and then yield the remaining budget (*e.g.* see Fig. 3.8). This common pattern real-time will allow unused budget to be used to either sleep or perhaps run Linux.

3.4.4 Handling malicious rich I/O

As demonstrated by SCONE [12], BlackBox [215], TrustShadow [80], Occlum [190], Graphene-SGX [211], and others, encryption and authentication of data passed and returned by a system call can mitigate many Iago attacks [45] relating to the file system and network.

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 [52, 104, 191, 98, 141]. 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 [43, 136, 147]. The Simplex approach has also been explored in detail, to validate untrusted ML outputs [189, 216, 228, 227, 139, 89, 155, 144].

3.5 Ringmaster Enclave API

The `io_uring` user-space library, `liburing` [15], implicitly trusts the OS in certain way, so we redesigned a new library which 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. For example, Ringmaster does not read the ring sizes from shared memory, except to confirm that they are the expected size. 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.

3.5.1 Dynamic Shared Memory Arenas

Because most `io_uring` operations require pointers, and thus for a Ringmaster enclave require shared memory, the reuse and management of shared memory is important 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 likely will not be randomly interleaved, so a heavy-weight `malloc` for shared memory is not always needed.

With these patterns in mind, we chose an arena-based design [86] for managing shared memory in a performant way. Ringmaster arenas are blocks of shared memory which can be used to quickly allocate and free multiple objects that have a similar life cycle, throughout the process of one or more related I/O operations. We illustrate in Fig. 3.6 how multiple arenas can be allocated from shared memory, and used to store objects associated with `io_uring` operations.

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 silently fills its pool of shared memory with a `IORING_OP_ENCLAVE_MMAP` request. When the completion arrives from Linux, it can fulfill the arena request asynchronously.

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. In this way, small objects, *e.g.* file paths, socket address structures, *etc.*, can be quickly allocated and freed with an associated arena once it has been set up. Further, objects do not even need to be popped in many cases, as they will all be freed when the arena is freed.

Early request optimization: Given the potential overhead of the additional `IORING_OP_-`

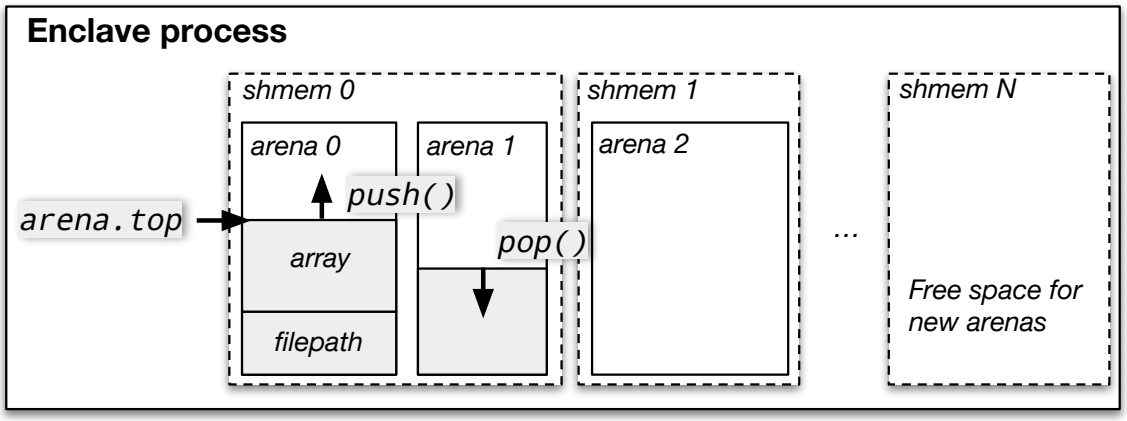

Figure 3.6: Diagram illustrating how Ringmaster shared memory arenas contain objects with similar life cycles

`ENCLAVE_MMAP` calls, we want to minimize the number of requests made, Ringmaster has a few techniques for doing this (however, the possibilities for optimizations are quite large here). First, in many cases, we can request an approximation of the needed arena sizes for an enclave 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 can overlap the request time with other initialization tasks. Once the initial large block arrives, it can be divided into various useful arena sizes, based on heuristics of common sizes. By doing this we avoid potential overhead of the binary search used to translate pointers.

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 which is 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 contents of the header to create 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 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 [121].

3.5.2 Protections via Abstraction

No direct access to the SQ and CQ We avoid giving the 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 index (of type `sqe_id_t` to the slot in the SQ. We avoid giving the programmer direct access to CQEs in shared ring memory; however, since CQEs are small, we opt to copy the data out of shared memory into a private `struct io_uring_cqe`. Thus the programmer gets access to a CQE pointer similar to `liburing`.

Protected `user_data` 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. So Ringmaster abstracts the true `user_data` field internally and provides a safe user-data field. We store the programmer's `user_data` in a private table, and use a value internal to the Ringmaster library which is bounds checked for safety. Additionally, this allows Ringmaster to mitigate double-completion attacks by forcing a monotonically increasing internal field value.

3.6 Ringmaster LibC: Legacy Applications

We designed a Ringmaster LibC (based on Musl LibC) which can be compiled with unmodified source code to run as an enclave. We modify the implementation of 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 as normal. Table 3.1 also shows how we divided up the calls.

Service / System Call	OS	Rationale
page-table management, `mmap`, `munmap`, `mprotect`, `mremap`, `sbrk`, `brk`, *etc.*	Ringmaster	Difficult to protect against Iago attacks [45], private data can be leaked through page access patterns [192]
scheduling, `yield()`, `gettimeofday()`	Ringmaster	Necessary for availability
randomness	Ringmaster	Linux-controlled randomness would allow 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 proxy's process ID
filesystem access, `open()`, `read()`, `write()`, `close()`, `unlink()`, `mkdir()`, `statx()`, *etc.*	Linux / `io_uring`	Complex filesystem drivers are already available
`socket()`, `bind()`, `connect()`, `accept()`, `recv()`, `send()`, *etc.*	Linux / `io_uring`	Complex network drivers and TCP/IP stack drivers already available in Linux; 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` [210] (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 [21] and inflate TCB

Table 3.1: 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`

3.6.1 Time-Sensitivity with a Synchronous API

Ringmaster allows a programmer to make some changes to legacy POSIX-style applications to prevent blocking attacks from Linux. Instead of waiting for the completion event, the programmer can define a call-specific timeout for any system call. After the timeout, `errno` will be set to `EAGAIN` or `ETIMEOUT` to indicate the timeout failure. Additionally, the programmer can register a `SIGALRM` signal, which will interrupt the spin-loop waiting for the completion.

3.6.2 Asynchronous Optimizations

Similar to previous work [207], we optimize Ringmaster LibC by putting `stdio` buffering in shared memory and making asynchronous writes and reads. When functions like `fwrite` are used, the `FILE` structure stores information used to buffer I/O before flushing to the kernel. We replace the default stdio buffers with shared memory arenas, once the buffering condition has been met (new line or full buffer), we write the arena asynchronously and return. Additionally, 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 we need to send.

3.7 Details on Ringmaster OS & Linux

Here we explain the design details of Ringmaster OS in order to clarify how Ringmaster meets its security claims.

3.7.1 Secure System Calls & Devices

Similar to early enclave work on Proxos [204], 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 Fig. 3.1.

3.7.2 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 [183, 168, 143, 218]; 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.

3.7.3 Starting Enclaves with Resource Donation

An adversary may try to launch many enclaves in order 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. With a budget-enforcing scheduler, Ringmaster OS ensures that time 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.

3.7.4 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, there are no operations at this time for `bind()` and `listen()`. For this reason, we implemented a number of `io_uring` operations (around 10) 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).

3.8 Security Analysis

In this section, we first perform an analysis of Ringmaster based on Game 1 and 2 defined in §3.3.2. Then, we evaluate a drone application, discussed in §3.2, with Ringmaster security.

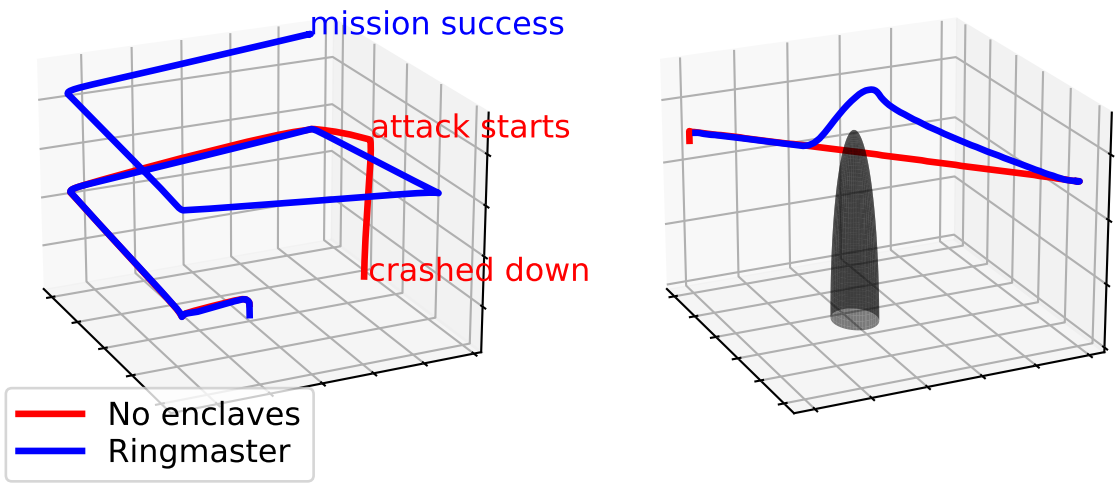

Figure 3.7: **Left:** Flight paths of same mission, showing how Ringmaster protects the safety-critical MAVLink device from being attacked by the compromised host OS, prevent arbitrary crash; **Right:** Flight paths from the obstacle avoidance experiment, showing how an adversarial OS cannot affect the enclave's path-planner, so it avoids a near-miss scenario.

3.8.1 Case study: A highly-secure drone

We implemented the drone shown in Fig. 3.1. We constructed a quad-copter platform with a CubePilot Orange autopilot device and a Raspberry Pi4b companion computer running a Linux 6.5 Buildroot OS.[4] 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 [116]. 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. 3.7).

[4]For this device nested page tables must be used for memory isolation because it does not have memory bus security.

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.

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 experiement: (1) no enclaves, *i.e.* no Ringmaster protection, and (2) with Ringmaster protections; results are shown on the right in Fig. 3.7. 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.

3.8.2 Detailed Analysis

In our adversary model's assumptions (§3.3.2, 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. 3.8. This code is showing a very 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. With this in mind, we can analyze our Ringmaster implementation against the two games defined in our adversary mode.

Game 1: Timeliness. 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 [218]. The enclave's physical memory is

```
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 3.8: Example Ringmaster enclave event loop

isolated from Linux via the ARM TZASC [10], nested paging, *etc.*; Ringmaster manages enclave page tables and page faults. Multicore synchronization locks in Ringmaster OS are MCS locks [153] with bounded wait times [106], 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 refuses 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.

Game 2: Confidentiality & Integrity. Linux could potentially try to mishandle system

	Latency (μs)						Overhead					
	Linux + GNU Libc			Ringmaster				T.S.[80]	B.B.[215]	V.G.[53]	I.T.[94]	Proxos[204]
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	2.5x	3.90x	55.8x	12.51x
open	9.93	14.31	934.94	12.67	16.78	721.52	1.17x	1.40x	1.5x	4.93x	7.95x	32.61x
read	1.48	1.63	84.02	2.37	3.02	585.04	1.85x	-	2.1x	-	-	13.06x
write	1.43	1.78	98.54	2.13	2.72	568.57	1.53x	-	2.1x	-	-	12.82x
stat	4.19	4.74	44.02	6.63	12.93	586.37	2.73x	-	2.9x	-	-	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 3.2: Table of LMBench Microbenchmarks, instrumented to run on Ringmaster, overhead is compared with reported results from related enclave research

calls to affect the enclave. For example, it could try to allocate too little shared memory or shared memory that overlaps with another enclave's private data. This is not possible, however, 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.

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 (§3.6) 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.

3.9 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 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 applications compiled with 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. Real-world system designers will have to tune the real-time parameters, core affinities, *etc.* to their specific context.

3.9.1 Latency Microbenchmarks

We opted to test the latency of Ringmaster by using a subset of the LMBench tests [152] as this was a framework commonly used by other enclave works. We compiled LMBench sources with Ringmaster LibC with minimal modifications so that we can compare with existing works' reported overheads. By doing this, our measurements also capture the overhead costs of (1) `user_data` protection, (2) arena allocation, (3) shared-memory address translation, (4) copies into or out of shared memory. We chose as many tests as we could, omitting any that required `fork()`, `exec()`, or heavy modifications.

Table 3.2 shows the results of our experiments. We observe comparable or better overhead for all benchmarks when compared the reported results of with related enclave solutions: TrustShadow (T.S.) [80], BlackBox (B.B.) [215], Virtual Ghost (V.G.) [53], and InkTag (I.T.) [94]. The latency overheads of Ringmaster are minimal due to the fact that we can 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 to do the four steps listed above, in addition to multicore cache coherence, and possible overhead latency from `io_uring` does add at least a few hundred nanoseconds to each system call.

Q1: Ringmaster has some latency overhead compared to synchronous Linux system calls, but it is comparable to, or better than, similar works.

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 3.3: Highly parallelized throughput tests for the network and file system, comparing a regular Linux progress to a Ringmaster enclave

3.9.2 Throughput Comparison with `liburing`

We designed benchmark applications which 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 here Ringmaster could achieve nearly zero overhead when compared to a regular process because there are no significant differences in the data paths, once enough shared memory has been established. Table 3.3 shows that in general we can receive large volumes of data into a process very quickly—fully saturating the gigabit Ethernet port of our Pi 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.

3.9.3 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 which were amenable for more rigorous performance analysis, `cat`, `tee`, `sha512sum`, and `base64`, and experimented with them. We measured their overhead with three different file sizes, normalized against the non-enclave version.

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 and compared to about 5ms on average with our current approach). For all

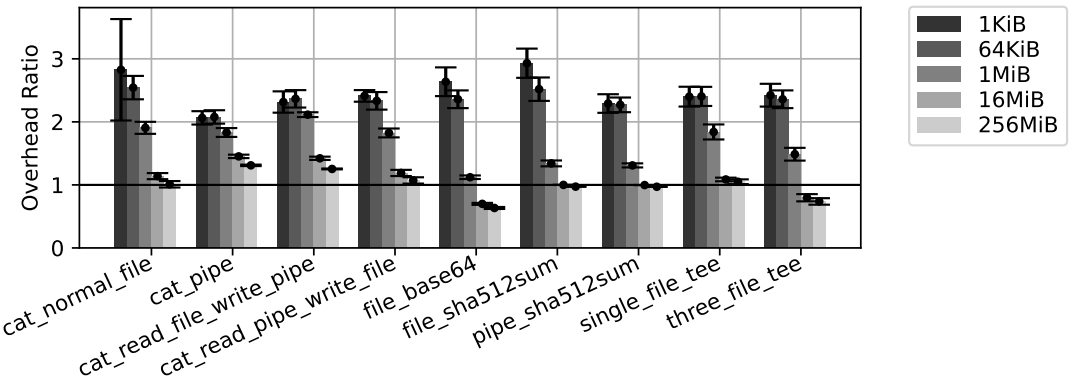

Figure 3.9: Benchmarks measuring overhead of unmodified GNU coreutils programs linked against Ringmaster's LibC.

tested programs programs we observe that we can get match or exceed baseline performance with longer running operations. In fact, the enclaved `base64`, `sha512`, and `tee` applications had statistically significant performance improvements (nearly a 50% increase for `base64`), due to the fact that they can utilize parallelism from our optimized 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. 3.10 (here lower numbers are better). First, we can see that pure computation has very low overhead with the `hanoi` test (the differences are likely due to compiler versions). The `socket` test, which sends data through TCP sockets measured almost no overhead for Ringmaster. We attribute this to the fact that, compared to the TCP/IP stack overhead, the enclave `io_uring` execution is not significant. The file system tests, `fsbuffer`, `fsdisk`, `fstime`, *etc.*, have 2-4x overhead on any tests involving writes; this appears to be an artifact of running `io_uring` operations fully synchronously. For these tests we do not use buffered I/O, so there was no parallelism optimization.

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

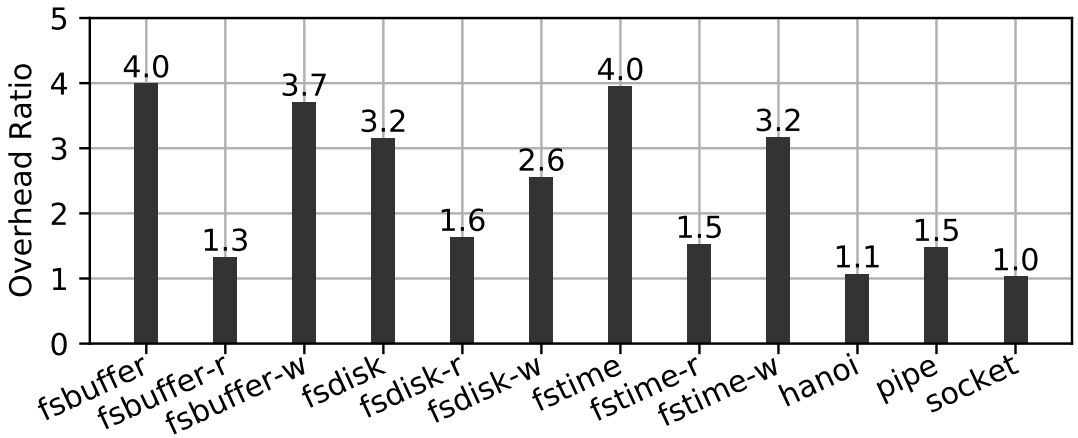

Figure 3.10: UnixBench Microbenchmarks, instrumented to run on Ringmaster's LibC, compared against native Linux.

3.10 Related Work

Asynchronous System Calls In Enclaves: Solutions such as SCONE [12], Eleos [164], and Occlum [190, 207] 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 (§3.4.2) 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 [99]) which is less common for CPS, and we are not confident that SGX will ever be able 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 programs that useR-

ingmaster LibC, this work presents the insight and an implementation that can support timeouts needed to ensure availability.

Proxos: Proxos [204] 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 timing-sensitive enclaves (see Fig. 3.2).

In theory, Proxos could be combined with FlexSC[195], SCONE [12], Eleos [164], or Occlum [190, 207]. 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 [47, 94, 22, 23, 12, 211, 80, 164, 193, 190, 207, 1, 215], Ringmaster's major unique contributions are mechanisms to support availability (though with some modifications). While next-gen Occlum [207] utilizes `io_uring` to reduce SGX switch overhead, it does not address real-time requirements or inter-address-space memory sharing. TrustShadow [80] forwards system calls from TrustZone to Linux but cannot ensure availability due to its dependency on Overshadow-style page table management [47] 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 [149], like Jailhouse [174], PikeOS [101], VxWorks [222], Xen [19, 223], seL4 in SMACCM [111, 91, 48], Bao [150], and the TrustZone-assisted hypervisors [183, 108, 168, 148, 143] represent the dominant isolation technique for CPS. However, they have some challenging trade offs for real-world systems preventing their widespread traction. First, to communicate between the RTOS and general-purpose VM, they need a VirtIO-like system [162, 187]. This incurs significant engineering overhead to establish data flows, possibly requiring the entire system hypervisor to be rebuilt for a single change to a VM's task. Secondly, virtualization simply for isolation may lead to excessive overheads for some embedded systems [209, 90]. Projects like Jailhouse trade the cost of VM exits for the usage of an entire core, often 25-50% of the processing power on SoCs.

Secure I/O for Enclaves: LDR demonstrates how safely reuse Linux drivers in the TrustZone by dividing their functionality between worlds [225]; if paired with Ringmaster, this is a potential solution to the malicious I/O data problem. Works like StrongBox [62] and Graviton [217] 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 [85] and RT-TEE [218] protect device access have challenges scaling for broader applicability.

Existing TrustZone TEE solutions: OP-TEE [133], and other TrustZone TEE solutions are unfortunately not well suited to time-sensitive applications because they act as passive services—waiting for invocation from the host. While RT-TEE [218] demonstrated how to use a secure timer to scheduler OP-TEE tasks, OP-TEE contains too many potential timing vulnerabilities (*e.g.* spin-locks around exhaustible resources) to make it a practical platform for availability.

3.11 Conclusion

Modern safety-critical CPS have paradoxically 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 which allowed protected enclaves to have access to rich OS features while protecting it from many timing attacks. Our results showed that an strong isolation plus an asynchronous approach to rich I/O allowed an enclaves to have new timing guarantees which were previously not possible—allowing for much greater security for robotics.

Chapter 4

Future Work and Discussion

4.1 Future Work

There are several future avenues for PARTEE's and Ringmaster's development and research not discussed in the previous chapters; we discuss key directions below.

4.1.1 Formal verification

Formal verification of the combined prototype of PARTEE and Ringmaster would provide strong security assurances for claims that are now only supported via claims about trusted-computing-based (TCB) size. While PARTEE OS is a redesigned version of CertiKOS, it retains the layer-based design pattern to a large extent. This means that a new formal verification campaign will likely be able to take advantage of improved proof-automation techniques in the field [55, 130, 226], in addition to advances in verified compilers unlocking verification for various complex concurrent programs [231], and linking between different verified compilation units [114, 196, 77]. Additionally, previous work has shown how to formally verify real-time schedulers [137, 138] for CertiKOS. Moreover, Komodo and others, has already shown that it is possible to build formally-verified TEEs [70, 46] (of the type that do not provide any availability guarantees).

New challenges: There are some new potential challenges for verification that future work will need to solve. First, we have redesigned the abstraction layers used in the original CertiKOS layer-based design. For example, we have decoupled some of the key OS kernel abstractions: separating memory quotas from individual processes. Second, we have allowed for dynamic kernel

object allocation using these per-partition memory quotas. Previous implementations of CertiKOS avoid dynamic object allocation to a large extent, but this greatly limits usability and runtime functionality. Additionally, PARTEE's partitioned dynamic object allocation allows for more simple system configuration, while also preventing cross-partition DoS attacks. Third, device queues, and asynchronous device interactions are a key new feature, but has not been verified before. Fourth, we need to be able to reason about low-level synchronization primitive instructions under the ARM memory model, which will affect kernel thread interactions when it comes to PARTEE. Finally, we have been seeing explosive growth in the Rust community; and major forces are advocating for a future where C code is phased out in favor of Rust [59]. The PARTEE and Ringmaster OS implementation may gain community traction and ease of verification if rewritten in Rust [120, 13, 93].

4.1.2 Opportunities with ARMv9's CCA architecture

The ARMv9 architecture, when it is available, will support the new Confidential Compute Architecture (CCA), which is an expansion of the ARM TrustZone; this approach likely has several advantages and open questions which pertain to PARTEE and Ringmaster. For example, the CCA now fully separates the EL3 privilege level (called "root"), the highest privilege level used for the firmware or "secure monitor," from the Secure-world OS [11]. Previously, there was no way of isolating memory accesses between these two processor modes; however, now with the new *granule protection table* (GPT), as part of the CCA specification, memory access control can be dynamically configured for memory regions. This solves another major problem where compromises in the Secure-world TEE OS would allow privilege escalations to supervisor mode in Linux (previously the Secure-world kernel could access any physical memory). Thus, PARTEE's combined firmware and Secure-world TEE OS would need to be redesigned. For performance and security reasons it may make more sense to keep a combined approach, but ideally we would want formally verify the TEE OS to avoid security issues. At a high-level the new Realm world seems to be another Secure world; however it has the explicit purpose of running enclave VMs using a verified Realm Management Monitor (RMM) [129, 131]. Additionally, it may make sense to use the new Realm world to allow for driver enclaves or VM enclaves, we think it is an open question how these

hardware primitives could be used outside of their stated purpose.

4.1.3 TEE-assisted smart contracts and side-channel hardening

While both PARTEE and Ringmaster used CPS and robotics applications as demos, we believe there are numerous emerging potential directions in the cloud and distributed computing space. First, because these works have focused on time-sensitive contexts, they could very easily apply to blockchains, which can be very sensitive to transaction latency and ordering. For decentralized exchanges, *maximum extractable value* (MEV) bots try to exploit race conditions, auction pricing, and multi-trade atomicity in order to perform arbitrage between different cryptocurrencies by *e.g.* front running or back running transactions make millions [57, 159, 175, 103]. Thus, low latency has become an important aspect of the cryptocurrency arms race (similar to traditional financial institutions with extremely low-latency data centers on Wall street) [32]. Furthermore, TEEs have become a highly popular approach to optimizing slow distributed computations [125, 208, 39]. Therefore, we believe PARTEE and Ringmaster's real-time scheduling, with guaranteed budgets and periods, along with the ability to perform kernel bypassed network I/O from a TEE, make it a prime candidate for this space. While this approach may not be able to prevent denial of network service (if the untrusted host controls the network card), PARTEE can provide soft real-time guarantees to enclaves to minimize latency and ensure confidentiality.

If used in this context, PARTEE and Ringmaster could provide better resistance to side-channel attacks due to our usage of the ARM TrustZone. For instance, we could mitigate many caching and branch-predication attacks in software using known techniques [233, 134, 71]. Furthermore, due to PARTEE's page-table management happening in the Secure world, page-fault attacks prevalent on SGX [192] are automatically mitigated as well.

4.2 Concluding Thoughts

This work focused on building a new software platform for time-sensitive TEEs, with strong potential to be formally verified in future work. While there has been strong past research on formally verified operating systems in the past [110, 78], modern software and hardware com-

plexity and market pressures demands large third-party software stacks. Thus, my research was focused on enclaves or TEEs as a practical alternative to the end-to-end, clean-slate approach to verified software. My colleagues and I built PARTEE and Ringmaster, which together make this new platform. In summary, this work provided evidence and designs supporting the following hypotheses:

- *Current modern TEE OSes fundamentally are not designed for availability or timing guarantees.* I provided an analysis of availability challenges for modern TEE OSes using three example attacks (§2.3).

- *Partitioning a TEE/enclave helps ensure temporal isolation and availability.* We designed and implemented PARTEE, a novel TEE OS designed from the ground up to mitigate denial-of-service attacks on TEE OS kernel memory, time, devices, *etc.* by partitioning them according to security domain (§2.5). Through our experiments we demonstrated that this design can provide enclaves with more reliable response times and latencies.

- *Asynchronous publishing and subscribing for enclaves enable modular, data-centric, and time-sensitive designs.* By making PARTEE's primitive IPC interface asynchronous and publish/-subscribe, the timing of incoming messages and behaviors of enclaves is decoupled by default. Thus, enclaves do not need to be blocked by message passing, and the system can be flexibly extended to allow multiple readers and writers. Service-style IPC can be implemented on top of this primitive if needed. The end result is this: Where previously it was difficult to run arbitrary robotics or CPS applications, now they can run performantly on PARTEE (§2.6).

- *Asynchronous host OS system calls help enable enclave availability properties.* If the host OS cannot by default block an enclave via the system call interface, then the enclave can achieve good CPU availability (assuming the enclave is independently scheduled) and even secure I/O availability in CPS/robotics applications.

- *Asynchronous host OS system calls for enclaves can out-perform synchronous ones for ARM systems.* Due to the costs of switching between worlds, performing system calls asyn-

chronously over shared memory allows fewer world switches. This is because other cores can execute the system calls while the enclave can continue executing—not to mention batching and parallelization optimizations that can occur as well (§3.4). I present evaluations using unmodified GNU coreutils applications (§3.9.3). Our experiments showcase high throughput I/O into an enclave, achieving full saturation of the Pi's gigabit Ethernet port and nearly 1GiB/s file system read and write speeds.

- *Ringmaster's design allows legacy POSIX applications to run in enclaves, with potential availability benefits for new or modified applications.* Experimental results demonstrate comparable or better latency for system calls than past approaches for unmodified enclaves (§3.6, §3.9.3).

- *PARTEE and Ringmaster can be evaluated on working systems.* I showcase PARTEE's and Ringmaster security, usability, and performance with our drone implementation on a Raspberry Pi4b (§2.7, §3.8.1), with additional support for the NVIDIA Jetson TX2.

After the peer review process of these sections are completed, I will be very excited to release this work as an open-source project for developers to use to secure their systems. I hope to continue to explore how PARTEE and Ringmaster can evolve to needs of engineers and hopefully make a larger impact on safety, reliability, and security of our critical systems.

Bibliography

[1] Adil Ahmad, Juhee Kim, Jaebaek Seo, Insik Shin, Pedro Fonseca, and Byoungyoung Lee. CHANCEL: efficient multi-client isolation under adversarial programs. In *Network and Distributed Systems Security Symposium*, NDSS '21, Virtual, February 2021. The Internet Society.

[2] Fritz Alder, Gianluca Scopelliti, Jo Van Bulck, and Jan Tobias Mühlberg. About time: On the challenges of temporal guarantees in untrusted environments. In *Proceedings of the 6th Workshop on System Software for Trusted Execution*, SysTEX '23, pages 27–33, Rome, Italy, 2023. ACM New York, NY, USA.

[3] Fritz Alder, Jo Van Bulck, Frank Piessens, and Jan Tobias Mühlberg. Aion: Enabling open systems through strong availability guarantees for enclaves. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, pages 1357–1372, Virtual Event, Repbulic of Korea, November 2021. ACM New York, NY, USA.

[4] Esmerald Aliaj, Ivan De Oliveira Nunes, and Gene Tsudik. GAROTA: Generalized active Root-Of-Trust architecture (for tiny embedded devices). In *31st USENIX Security Symposium*, USENIX Security '22, pages 2243–2260, Boston, MA, August 2022. USENIX Association.

[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. ARM Trusted Firmware. Available: `https://github.com/ARM-software/arm-trusted-firmware`.

[8] ARM. Building a secure system using TrustZone technology. Technical report, ARM, 2009.

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

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

[11] ARM. *Learn the architecture — Realm Management Extension*, 2024.

[12] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L. Stillwell, et al. SCONE: Secure Linux containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, pages 689–703, Savannah, GA, November 2016. USENIX Association.

[13] Vytautas Astrauskas, Aurel Bílý, Jonáš Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. The Prusti Project: Formal verification for Rust. In Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez, editors, *NASA Formal Methods*, pages 88–108, Cham, 2022. Springer International Publishing.

[14] Atlassian. Incident management for high-velocity teams: Cost of downtime. Available: `https://www.atlassian.com/incident-management/kpis/cost-of-downtime`.

[15] Jens Axboe. liburing. Available: `https://github.com/axboe/liburing`.

[16] Jens Axboe. That's it. 10M IOPS, one physical core. Available: `https://twitter.com/axboe/status/1452689372395053062`.

[17] David Axe. Ukrainian marines hacked a russian drone to locate its base—then blew up the base with artillery. Available: `https://www.forbes.com/sites/davidaxe/2023/11/30/ukrainian-marines-hacked-a-russian-drone-to-locate-its-base-then-blew-up-the-base-with-artillery/`, November 2023.

[18] Gaurav Banga, Peter Druschel, and Jeffrey C Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, OSDI '99, New Orleans, Louisiana, USA, 1999. USENIX Association.

[19] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[20] Andrew Barrington, Steven Feldman, and Damian Dechev. A scalable multi-producer multi-consumer wait-free ring buffer. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 1321–1328, Salamanca, Spain, April 2015. ACM New York, NY, USA.

[21] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. A `fork()` in the road. In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems*, HotOS XVII, pages 14–22, Bertinoro, Italy, May 2019.

[22] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with Haven. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, pages 267–283, Broomfield, CO, October 2014. USENIX Association.

[23] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with Haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):1–26, 2015.

[24] Michael Bechtel and Heechul Yun. Denial-of-service attacks on shared cache in multicore: Analysis and prevention. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 357–367, 2019.

[25] David Berard and Vincent Dehors. I feel a draft. opening the doors and windows 0-click RCE on the Tesla Model3. In *Hexacon*, Vancouver, BC, October 2022.

[26] David Berard and Vincent Dehors. 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`.

[27] Deval Bhamare, Maede Zolanvari, Aiman Erbad, Raj Jain, Khaled Khan, and Nader Meskin. Cybersecurity for industrial control systems: A survey. *Computers & Security*, 89(101667), February 2020.

[28] Tamara Bonaci, Junjie Yan, Jeffrey Herron, Tadayoshi Kohno, and Howard Jay Chizeck. Experimental analysis of denial-of-service attacks on teleoperated robotic systems. In *Proceedings of the ACM/IEEE Sixth International Conference on Cyber-Physical Systems*, ICCPS '15, page 11–20, Seattle, Washington, 2015. ACM New York, NY, USA.

[29] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. TyTAN: Tiny trust anchor for tiny devices. In *Proceedings of the 52nd ACM/EDAC/IEEE Design Automation Conference*, DAC '15, pages 1–6, San Francisco, CA, USA, June 2015.

[30] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. SANCTUARY: ARMing TrustZone with user-space enclaves. In *Network and Distributed Systems Security Symposium*, NDSS '19, San Diego, CA, February 2019. The Internet Society.

[31] Felix Bruns, Shadi Traboulsi, David Szczesny, Elizabeth Gonzalez, Yang Xu, and Attila Bilgic. An evaluation of microkernel-based virtualization for embedded real-time systems. In *2010 22nd Euromicro Conference on Real-Time Systems*, ECRTS '10, pages 57–65. IEEE, 2010.

[32] BSO. Beyond the exchange: How ultra-low latency powers the entire crypto ecosystem. Available: `https://www.bso.co/all-insights/how-ultra-low-latency-powers-the-entire-crypto-ecosystem`.

[33] Alan Burns and Robert Davis. Mixed criticality systems—a review. Technical Report 13, Department of Computer Science, University of York, February 2022.

[34] Alan Burns and Robert I Davis. A survey of research into mixed criticality systems. *ACM Computing Surveys (CSUR)*, 50(6):1–37, 2017.

[35] Marcel Busch, Philipp Mao, and Mathias Payer. GlobalConfusion: TrustZone trusted application 0-days by design. In *33rd USENIX Security Symposium*, USENIX Security '24, pages 5537–5554, 2024.

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

[37] Pedro Cabrera. Parrot drones hijacking. Available: `https://www.rsaconference.com/Library/presentation/USA/2018/parrot-drones-hijacking`, April 2018.

[38] Zhiqiang Cai, Aohui Wang, and Wenkai Zhang. 0-days & mitigations: Roadways to exploit and secure connected BMW cars. In *Black Hat USA 2019*, Las Vegas, NV, 2019.

[39] Lucas Martin Calderon. Achieving superior blockchain security with SGX and ZK proofs. Available: `https://coinsbench.com/achieving-superior-blockchain-security-with-sgx-and-zk-proofs-702cd182db7e`, November 2023.

[40] Canonical. A CTO's guide to real-time Linux. Technical report, Canonical, March 2023.

[41] David Carrero. The linux kernel surpasses 40 million lines of code: A historic milestone in open-source software. Available: `https://www.stackscale.com/blog/linux-kernel-surpasses-40-million-lines-code/`.

[42] David Cerdeira, José Martins, Nuno Santos, and Sandro Pinto. ReZone: Disarming TrustZone with TEE privilege reduction. In *31st USENIX Security Symposium*, USENIX Security '22, pages 2261–2279, 2022.

[43] Raghu Changalvala and Hafiz Malik. Lidar data integrity verification for autonomous vehicle using 3D data hiding. In *2019 IEEE Symposium Series on Computational Intelligence*, SSCI '19, pages 1219–1225. IEEE, 2019.

[44] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, Tadayoshi Kohno, et al. Comprehensive experimental analyses of automotive attack surfaces. In *20th USENIX Security Symposium*, USENIX Security '11. USENIX Association, August 2011.

[45] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call API is a bad untrusted RPC interface. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 253—-264, Houston, Texas, USA, March 2013. ACM New York, NY, USA.

[46] Hongbo Chen, Haobin Hiroki Chen, Mingshen Sun, Kang Li, Zhaofeng Chen, and XiaoFeng Wang. A verified confidential computing as a service framework for privacy preservation. In *32nd USENIX Security Symposium*, USENIX Security '23, pages 4733–4750, Anaheim, CA, August 2023. USENIX Association.

[47] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan RK Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. *ACM SIGOPS Operating Systems Review*, 42(2):2–13, March 2008.

[48] Darren Cofer, John Backes, Andrew Gacek, Daniel DaCosta, Michael Whalen, Ihor Kuz, Gerwin Klein, Gernot Heiser, Lee Pike, Adam Foltzer, Michal Podhradsky, Douglas Stuart, Jason Grahan, and Brett Wilson. Secure mathematically-assured composition of control models. Technical report, Air Force Research Laboratory (RITA), September 2017.

[49] Johnathan Corbet. Ringing in a new asynchronous I/O API. Available: `https://lwn.net/Articles/776703/`, January 2019.

[50] Johnathan Corbet. The rapid growth of io_uring. `https://lwn.net/Articles/810414/`, January 2020.

[51] Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Paper 2016/086, 2016.

[52] Andrei Costin, Aurélien Francillon, et al. 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.

[53] John Criswell, Nathan Dautenhahn, and Vikram Adve. Virtual ghost: Protecting applications from hostile operating systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, Salt Lake City, Utah, USA, February 2014. ACM New York, NY, USA.

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

[55] Łukasz Czajka and Cezary Kaliszyk. Hammer for Coq: Automation for dependent type theory. *Journal of Automated Reasoning*, 61(1):423–453, 2018.

[56] Thijs Alkemade Daan Keuper. The connected car: Ways to get unauthorized access and potential implications. Technical report, Computest, 2018.

[57] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. arXiv preprint arXiv:1904.05234, 2019.

[58] DARPA. Crash: Clean-slate design of resilient, adaptive, secure hosts. Available: `https://www.darpa.mil/research/programs/clean-slate-design-of-resilient-adaptive-secure-hosts`.

[59] DARPA. Tractor: Translating all c to rust. Available: `https://www.darpa.mil/research/programs/translating-all-c-to-rust`.

[60] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs. In *13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 185–192. IEEE, 2010.

[61] DeepSpec. The science of deep specification.

[62] Yunjie Deng, Chenxu Wang, Shunchang Yu, Shiqing Liu, Zhenyu Ning, Kevin Leach, Jin Li, Shoumeng Yan, Zhengyu He, Jiannong Cao, et al. StrongBox: A GPU TEE on ARM endpoints. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, pages 769–783, Los Angeles, CA, USA, November 2022. ACM New York, NY, USA.

[63] DJI. Drone security white paper (version 3.0). Technical report, DJI, April 2024.

[64] Pan Dong, Alan Burns, Zhe Jiang, and Xiangke Liao. 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*, RTCSA '18, pages 59–64, Hakodate, Japan, August 2018. IEEE.

[65] Eclipse Foundation. Iceoryx: Lock-free queue. Available: `https://github.com/eclipse-iceoryx/iceoryx/blob/main/doc/design/lockfree_queue.md`.

[66] Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. 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*, WiSec '17, pages 99–110, Boston, Massachusetts, July 2017. ACM New York, NY, USA.

[67] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.

[68] Daniel J Fagnant and Kara Kockelman. Preparing a nation for autonomous vehicles: Opportunities, barriers and policy recommendations. *Transportation Research Part A: Policy and Practice*, 77:167–181, 2015.

[69] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32.Stuxnet dossier. Technical report, Symantec Corp., February 2011.

[70] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 287–305. ACM New York, NY, USA, October 2017.

[71] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time protection: The missing os abstraction. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, Dresden, Germany, 2019. ACM New York, NY, USA.

[72] Lukas Giner, Stefan Steinegger, Antoon Purnal, Maria Eichlseder, Thomas Unterluggauer, Stefan Mangard, and Daniel Gruss. Scatter and split securely: Defeating cache contention and occupancy attacks. In *2023 IEEE Symposium on Security and Privacy*, IEEE S&P '23, pages 2273–2287. IEEE, 2023.

[73] GlobalPlatform. *GlobalPlatform Device Technology TEE Client API Specification Version 1.0*, July 2010.

[74] GlobalPlatform. *GlobalPlatform Technology TEE Internal Core API Specification Version 1.1.2.50*, June 2018.

[75] GlobalPlatform. *GlobalPlatform Technology TEE System Architecture Version 1.2*, November 2018.

[76] Liang Gu, Alexander Vaynberg, Bryan Ford, Zhong Shao, and David Costanzo. CertiKOS: A certified kernel for secure cloud computing. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, Shanghai, China, 2011. ACM New York, NY, USA.

[77] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, page 595–608, Mumbai, India, 2015. ACM New York, NY, USA.

[78] Ronghui Gu, Zhong Shao, Hao Chen, Jieung Kim, Jérémie Koenig, Xiongnan (Newman) Wu, Vilhelm Sjöberg, and David Costanzo. Building certified concurrent OS kernels. *Communications of the ACM*, 62(10):89–99, September 2019.

[79] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, pages 653–669, Savannah, GA, November 2016. USENIX Association.

[80] Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. TrustShadow: Secure execution of unmodified applications with ARM TrustZone. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '17, pages 488–501, Niagara Falls, New York, USA, June 2017. ACM New York, NY, USA.

[81] Tarek Guesmi, Rojdi Rekik, Salem Hasnaoui, and Houria Rezig. Design and performance of DDS-based middleware for real-time control systems. *International Journal of Computer Science and Network Security (IJCSNS)*, 7(12):188–200, 2007.

[82] Liwei Guo and Felix Xiaozhu Lin. Minimum viable device drivers for ARM TrustZone. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 300–316, Rennes, France, 2022. ACM New York, NY, USA.

[83] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization*, WWC-4, pages 3–14, Austin, TX, USA, December 2001. IEEE.

[84] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest we remember: Cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009.

[85] Seungkyun Han and Jinsoo Jang. MyTEE: Own the trusted execution environment on embedded devices. In *Network and Distributed Systems Security Symposium*, NDSS '23, San Diego, CA, February 2023. The Internet Society.

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

[87] Norm Hardy. The Confused Deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, 1988.

[88] Monowar Hasan, Ashish Kashinath, Chien-Ying Chen, and Sibin Mohan. Sok: Security in real-time systems. *ACM Computing Surveys*, 56(9), April 2024.

[89] Monowar Hasan and Sibin Mohan. 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*, IoT S&P'19, page 8–14, London, United Kingdom, 2019. ACM New York, NY, USA.

[90] Gernot Heiser. The role of virtualization in embedded systems. In *Proceedings of the 1st workshop on Isolation and Integration in Embedded Systems*, IIES '08, pages 11–16, Glasgow, UK, April 2008.

[91] Gernot Heiser, Lucy Parker, Peter Chubb, Ivan Velickovic, and Ben Leslie. Can we put the "S" into IoT? In *IEEE 8th World Forum on Internet of Things*, WF-IoT '22, pages 1–6. IEEE, 2022.

[92] Mario Herger. 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/`, November 2020.

[93] Son Ho and Jonathan Protzenko. Aeneas: Rust verification by functional translation. *Proceedings of the ACM on Programming Languages*, 6(ICFP), August 2022.

[94] Owen S Hofmann, Sangman Kim, Alan M Dunn, Michael Z Lee, and Emmett Witchel. InkTag: Secure applications on an untrusted operating system. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 265–278, Houston, Texas, USA, March 2013. ACM New York, NY, USA.

[95] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. vTZ: Virtualizing ARM TrustZone. In *26th USENIX Security Symposium*, USENIX Security '17, pages 541–556, Vancouver, BC, August 2017. USENIX Association.

[96] Jinsoo Jang and Brent Byunghoon Kang. 3rdParTEE: Securing third-party IoT services using the Trusted Execution Environment. *IEEE Internet of Things Journal*, 9(17):15814–15826, 2022.

[97] Dongxu Ji, Qianying Zhang, Shijun Zhao, Zhiping Shi, and Yong Guan. MicroTEE: Designing TEE OS based on the microkernel architecture. In *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, TrustCom '19, pages 26–33, 2019.

[98] Wei Jia, Zhaojun Lu, Haichun Zhang, Zhenglin Liu, Jie Wang, and Gang Qu. Fooling the eyes of autonomous vehicles: Robust physical adversarial examples against traffic sign recognition systems. In *Network and Distributed Systems Security Symposium*, NDSS '22, San Diego, CA, April 2022. The Internet Society.

[99] Yuekai Jia, Shuang Liu, Wenhao Wang, Yu Chen, Zhengde Zhai, Shoumeng Yan, and Zhengyu He. HyperEnclave: An open and cross-platform trusted execution environment. In *2022 USENIX Annual Technical Conference*, USENIX ATC '22, pages 437–454, Carlsbad, CA, July 2022. USENIX Association.

[100] Zhe Jiang, Pan Dong, Ran Wei, Qingling Zhao, Yankai Wang, Dizhong Zhu, Yan Zhuang, and Neil Audsley. PSpSys: A time-predictable mixed-criticality system architecture based on ARM TrustZone. *Journal of Systems Architecture*, 123, 2022.

[101] Robert Kaiser and Stephan Wagner. Evolution of the pikeos microkernel. In *First International Workshop on MicroKernels for Embedded Systems*, MIKES '07, January 2007.

[102] Ali Kani. 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/`, September 2022.

[103] Harsh Kasana. The dark forest: A builder's guide to mev protection. Available: `https://medium.com/@harshkasana05/the-dark-forest-a-builders-guide-to-mev-protection-2f422fba9a1d`, July 2025.

[104] Andrew J Kerns, Daniel P Shepard, Jahshan A Bhatti, and Todd E Humphreys. Unmanned aircraft capture and control via GPS spoofing. *Journal of Field Robotics*, 31(4):617–636, 2014.

[105] Arslan Khan, Hyungsub Kim, Byoungyoung Lee, Dongyan Xu, Antonio Bianchi, and Dave Jing Tian. M2MON: Building an MMIO-based security reference monitor for unmanned vehicles. In *30th USENIX Security Symposium*, USENIX Security '21, pages 285–302. USENIX Association, August 2021.

[106] Jieung Kim, Vilhelm Sjöberg, Ronghui Gu, and Zhong Shao. Safety and liveness of MCS lock—layer by layer. In *15th Asian Symposium on Programming Languages and Systems*, APLAS '17, pages 273–297, Suzhou, China, November 2017. Springer.

[107] Kyounggon Kim, Jun Seok Kim, Seonghoon Jeong, Jo-Hee Park, and Huy Kang Kim. Cybersecurity for autonomous vehicles: Review of attacks and defense. *Computers & Security*, 103, 2021.

[108] Se Won Kim, Chiyoung Lee, MooWoong Jeon, Hae Young Kwon, Hyun Woo Lee, and Chuck Yoo. Secure device access for automotive software. In *2013 International Conference on Connected Vehicles and Expo*, ICCVE '13, pages 177–181. IEEE, 2013.

[109] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: The Linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230, Ottawa, Dntorio, Canada, July 2007.

[110] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1), February 2014.

[111] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220. ACM New York, NY, USA, October 2009.

[112] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *19th Annual International Cryptology Conference*, CRYPTO '99, Santa Barbara, CA, USA, August 1999. Springer-Verlag.

[113] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. TrustLite: A security architecture for tiny embedded devices. In *Proceedings of the 9th European Conference on Computer Systems*, EuroSys '14, Amsterdam, Netherlands, April 2014. ACM New York, NY, USA.

[114] Jérémie Koenig and Zhong Shao. Compcerto: compiling certified open c components. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 1095–1109, Virtual, Canada, 2021. ACM New York, NY, USA.

[115] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, et al. Experimental security analysis of a modern automobile. In *2010 IEEE Symposium on Security and Privacy*, IEEE S&P '10, pages 447–462, Oakland, CA, USA, May 2010. IEEE.

[116] Anis Koubâa, Azza Allouch, Maram Alajlan, Yasir Javed, Abdelfettah Belghith, and Mohamed Khalgui. Micro Air Vehicle Link (MAVLink) in a nutshell: A survey. *IEEE Access*, 7:87658–87680, 2019.

[117] Niclas Kühnapfel, Christian Werling, and Hans Niklas Jacob. Recovering critical data from tesla autopilot using voltage glitching. In *37th Chaos Communication Congress*, 37C3, Hamburg, Germany, December 2023.

[118] Mark Kuhne, Supraja Sridhara, Andrin Bertschi, Nicolas Dutly, Srdjan Capkun, and Shweta Shinde. Aster: Fixing the android TEE ecosystem with ARM CCA. arXiv preprint arXiv:2407.16694, 2024.

[119] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, (2):125–143, 1977.

[120] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. Verus: A practical foundation for systems verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 438–454, Austin, TX, USA, 2024. ACM New York, NY, USA.

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

[122] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20. ACM New York, NY, USA, April 2020.

[123] Patrick PC Lee, Tian Bu, and Girish Chandranmenon. A lock-free, cache-efficient shared ring buffer for multi-core architectures. In *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '09, pages 78–79, 2009.

[124] Robert M. Lee, Michael J Assante, and Tim Conway. Analysis of the cyber attack on the ukrainian power grid. Technical report, SANS ICS, E-ISAC, Washington, DC, 2016.

[125] Rujia Li, Qin Wang, Qi Wang, David Galindo, and Mark Ryan. Sok: Tee-assisted confidential smart contracts. In *Proceedings on Privacy Enhancing Technologies*, PETS '22, 2022.

[126] Shih-Wei Li, John S Koh, and Jason Nieh. Protecting cloud virtual machines from hypervisor and host operating system exploits. In *28th USENIX Security Symposium*, USENIX Security '19, pages 1357–1374, 2019.

[127] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. Formally verified memory protection for a commodity multiprocessor hypervisor. In *30th USENIX Security Symposium*, USENIX Security '21, pages 3953–3970, 2021.

[128] Wenhao Li, Yubin Xia, Long Lu, Haibo Chen, and Binyu Zang. Teev: Virtualizing trusted execution environments on mobile platforms. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '19, pages 2–16. ACM New York, NY, USA, 2019.

[129] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Design and verification of the Arm Confidential Compute Architecture. In *16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '22, pages 465–484, Carlsbad, CA, July 2022. USENIX Association.

[130] Xupeng Li, Xuheng Li, Wei Qiang, Ronghui Gu, and Jason Nieh. Spoq: Scaling machine-checkable systems verification in Coq. In *17th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '23', pages 851–869, Boston, MA, July 2023. USENIX Association.

[131] Xupeng Li Li, Xuheng Li, Christoffer Dall, Gu Ronghui, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Enabling realms with the arm confidential compute architecture. Available: `https://www.usenix.org/publications/loginonline/enabling-realms-arm-confidential-compute-architecture`, 2023.

[132] Yanlin Li, Jonathan McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. MiniBox: A two-way sandbox for x86 native code. In *2014 USENIX Annual Technical Conference*, USENIX ATC '14, pages 409–420, Philadelphia, PA, June 2014. USENIX Association.

[133] Linaro. Open portable trusted execution environment. Available: `https://github.com/OP-TEE/optee_os`.

[134] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium*, USENIX Security '16, pages 549–564, Austin, TX, August 2016. USENIX Association.

[135] Justin Littlefield-Lawwill and Larry Kinnan. System considerations for robust time and space partitioning in integrated modular avionics. In *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, pages 1–B. IEEE, 2008.

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

[137] Mengqi Liu, Lionel Rieg, Zhong Shao, Ronghui Gu, David Costanzo, Jung-Eun Kim, and Man-Ki Yoon. Virtual timeline: a formal abstraction for verifying preemptive schedulers with temporal isolation. *Proceedings of the ACM on Programming Languages*, December 2019.

[138] Mengqi Liu, Zhong Shao, Hao Chen, Man-Ki Yoon, and Jung-Eun Kim. Compositional virtual timelines: Verifying dynamic-priority partitions with algorithmic temporal isolation. *Proceedings of the ACM on Programming Languages*, 6:60–88, October 2022.

[139] Renju Liu and Mani Srivastava. PROTC: PROTeCting drone's peripherals through ARM TrustZone. In *Proceedings of the 3rd Workshop on Micro Aerial Vehicle Networks, Systems,*

and Applications, DroNet '17, pages 1–6, Niagara Falls, New York, USA, June 2017. ACM New York, NY, USA.

[140] Yin Liu, Kijin An, and Eli Tilevich. 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*, GPCE '18, pages 175–187, Boston, MA, USA, November 2018. ACM New York, NY, USA.

[141] Ziwei Liu, Zhongqi Miao, Xiaohang Zhan, Jiayun Wang, Boqing Gong, and Stella X Yu. Large-scale long-tailed recognition in an open world. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, CVPR '19, pages 2537–2546. Computer Vision Foundation, June 2019.

[142] Lookwerks. HACMS: High Assurance Cyber Military Systems. Available: `http://loonwerks.com/projects/hacms.html`.

[143] Pierre Lucas, Kevin Chappuis, Michele Paolino, Nicolas Dagieu, and Daniel Raho. VOSYSmonitor, a low latency monitor layer for mixed-criticality systems on ARMv8-A. In Marko Bertogna, editor, *29th Euromicro Conference on Real-Time Systems*, volume 76 of *ECRTS '17*, pages 6:1–6:18, Dagstuhl, Germany, 2017. Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[144] Baiting Luo, Shreyas Ramakrishna, Ava Pettet, Christopher Kuhn, Gabor Karsai, and Ayan Mukhopadhyay. 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)*, ICCPS '23, page 177–186, San Antonio, TX, USA, 2023.

[145] David Mack. The art of shipping early and often. Available: `https://www.ycombinator.com/blog/tips-ship-early-and-often/`, 2016.

[146] Ivano Malavolta, Grace Lewis, Bradley Schmerl, Patricia Lago, and David Garlan. How do you architect your robots? State of the practice and guidelines for ROS-based systems. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '20, pages 31–40, Seoul, South Korea, 2020.

[147] Michele Marazzi, Stefano Longari, Carminati Michele, and Stefano Zanero. Securing LiDAR communication through watermark-based tampering detection. In *Symposium on Vehicles Security and Privacy*, VehicleSec '24, 2024.

[148] José Martins, João Alves, Jorge Cabral, Adriano Tavares, and Sandro Pinto. μRTZVisor: A secure and safe real-time hypervisor. *Electronics*, 6(4), 2017.

[149] José Martins and Sandro Pinto. Shedding light on static partitioning hypervisors for ARM-based mixed-criticality systems. In *29th Real-Time and Embedded Technology and Applications Symposium*, RTAS '23, pages 40–53. IEEE, 2023.

[150] José Martins, Adriano Tavares, Marco Solieri, Marko Bertogna, and Sandro Pinto. Bao: A lightweight static partitioning hypervisor for modern multi-core embedded systems. In *Workshop on Next Generation Real-Time Embedded Systems*, NG-RES '20. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[151] Ramya Jayaram Masti, Claudio Marforio, Aanjhan Ranganathan, Aurélien Francillon, and Srdjan Capkun. Enabling trusted scheduling in embedded systems. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 61–70, Orlando, FL, USA, December 2012.

[152] Larry W McVoy, Carl Staelin, et al. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, USENIX ATC '96, pages 279–294, San Diego, CA, USA, 1996.

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

[154] Charlie Miller. Remote exploitation of an unaltered passenger vehicle. In *Black Hat USA*, Las Vegas, NV, August 2015.

[155] Patrick Musau, Nathaniel Hamilton, Diego Manzanas Lopez, Preston Robinette, and Taylor T. Johnson. On using real-time reachability for the safety assurance of machine learning controllers. In *2022 IEEE International Conference on Assured Autonomy*, ICAA '22, pages 1–10, 2022.

[156] Ben Nassi, Ron Bitton, Ryusuke Masuoka, Asaf Shabtai, and Yuval Elovici. SoK: Security and privacy in the age of commercial drones. In *2021 IEEE Symposium on Security and Privacy*, IEEE S&P '21, pages 1434–1451, 2021.

[157] Alex Nehmy. 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`.

[158] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. TrustZone explained: Architectural features and use cases. In *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, pages 445–451, 2016.

[159] Dalmas Ngetich. Ross ulbricht lost $12 million to an mev bot on solana: Here's what happened. Available: `https://99bitcoins.com/news/ross-ulbricht-lost-12-million-to-an-mev-bot-on-solana-heres-what-happened/`, January 2025.

[160] Sen Nie, Ling Liu, and Yuefeng Du. Free-fall: Hacking tesla from wireless to CAN bus. In *Black Hat USA 2017*, Las Vegas, NV, 2017.

[161] NVIDIA. End-to-end solutions for autonomous vehicles. Available: `https://developer.nvidia.com/drive`.

[162] André Oliveira, José Martins, Jorge Cabral, Adriano Tavares, and Sandro Pinto. TZ-VirtIO: Enabling standardized inter-partition communication in a TrustZone-assisted hypervisor. In *2018 IEEE 27th International Symposium on Industrial Electronics*, ISIE '18, pages 708–713, Cairns, QLD, Australia, June 2018. IEEE.

[163] Daniel Oliveira, Tiago Gomes, and Sandro Pinto. uTango: An open-source TEE for IoT devices. *IEEE Access*, 10:23913–23930, 2022.

[164] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: ExitLess OS services for SGX enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 238–253, Belgrade, Serbia, April 2017.

[165] Gerardo Pardo-Castellote. OMG data-distribution service: Architectural overview. In *23rd International Conference on Distributed Computing Systems Workshops*, ICDCSW '03, pages 200–206. IEEE, 2003.

[166] Sandro Pinto, André Oliveira, Jorge Pereira, Jorge Cabral, João Monteiro, and Adriano Tavares. Lightweight multicore virtualization architecture exploiting ARM TrustZone. In *43rd Annual Conference of the IEEE Industrial Electronics Society*, IECON '17, pages 3562–3567, Beijing, China, October 2017. IEEE.

[167] Sandro Pinto, Jorge Pereira, Tiago Gomes, Mongkol Ekpanyapong, and Adriano Tavares. Towards a TrustZone-assisted hypervisor for real-time embedded systems. *IEEE Computer Architecture Letters*, 16(2):158–161, October 2016.

[168] Sandro Pinto, Jorge Pereira, Tiago Gomes, Adriano Tavares, and Jorge Cabral. LTZVisor: TrustZone is the key. In Marko Bertogna, editor, *29th Euromicro Conference on Real-Time Systems*, volume 76 of *ECRTS '17*, pages 4:1–4:22, Dagstuhl, Germany, 2017. Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

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

[170] Sanndro Pinto, Hugo Araujo, Daniel Oliveira, Jose Martins, and Adriano Tavares. Virtualization on TrustZone-enabled microcontrollers? voilà! In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '19, pages 293–304, Montreal, QC, Canada, April 2019. IEEE.

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

[172] Davide Quarta, Marcello Pogliani, Mario Polino, Federico Maggi, Andrea Maria Zanchettin, and Stefano Zanero. An experimental security analysis of an industrial robot controller. In *2017 IEEE Symposium on Security and Privacy*, IEEE S&P '17, pages 268–286, San Jose, CA, May 2017. IEEE.

[173] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Software*. Kobe, Japan, 2009.

[174] Ralf Ramsauer, Jan Kiszka, Daniel Lohmann, and Wolfgang Mauerer. Look mum, no vm exits!(almost). arXiv preprint arXiv:1705.06932, 2017.

[175] Dan Robinson. Ethereum is a dark forest. Available: `https://www.paradigm.xyz/2020/08/ethereum-is-a-dark-forest`, August 2020.

[176] Nils Rodday. Hacking a professional drone. In *Black Hat Asia 2016*, Singapore, March 2016.

[177] John Rushby. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Technical report, NASA, 1999.

[178] John M. Rushby. Design and verification of secure systems. *ACM SIGOPS Operating Systems Review*, 15(5):12–21, dec 1981.

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

[180] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: What it is, and what it is not. In *Proceedings of the 2015 IEEE Trustcom/BigDataSE/ISPA*, Trustcom '15, pages 57–64, Helsinki, Finland, 2015. IEEE.

[181] Selma Saidi, Rolf Ernst, Sascha Uhrig, Henrik Theiling, and Benoît Dupont de Dinechin. The shift to multicores in real-time and safety-critical systems. In *2015 International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '15, pages 220–229, Amsterdam, Netherlands, October 2015. IEEE.

[182] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[183] Daniel Sangorrin, Shinya Honda, and Hiroaki Takada. Dual operating system architecture for real-time embedded systems. In *6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, OSPERT '10, July 2010.

[184] Daniel Sangorrin, Shinya Honda, and Hiroaki Takada. Integrated scheduling for a reliable dual-OS monitor. *IPSJ Transactions on Advanced Computing Systems*, 5(2):99–110, 2012.

[185] Daniel Sangorrin, Shinya Honda, and Hiroaki Takada. Reliable and efficient dual-OS communications for real-time embedded virtualization. *Computer Software*, 29(4):182–198, 2012.

[186] Nico Schiller, Merlin Chlosta, Moritz Schloegel, Nils Bars, Thorsten Eisenhofer, Tobias Scharnowski, Felix Domke, Lea Schönherr, and Thorsten Holz. Drone security and the mysterious case of DJI's DroneID. In *Network and Distributed Systems Security Symposium*, NDSS '23, San Diego, CA, February 2023. The Internet Society.

[187] Gero Schwäricke, Rohan Tabish, Rodolfo Pellizzoni, Renato Mancuso, Andrea Bastoni, Alexander Zuepke, and Marco Caccamo. A real-time virtio-based framework for predictable inter-vm communication. In *2021 IEEE Real-Time Systems Symposium*, RTSS '21, pages 27–40. IEEE, 2021.

[188] Gianluca Scopelliti, Sepideh Pouyanrad, Job Noorman, Fritz Alder, Christoph Baumann, Frank Piessens, and Jan Tobias Mühlberg. End-to-end security for distributed event-driven enclave applications on heterogeneous TEEs. *ACM Transactions on Privacy and Security*, 26(3), June 2023.

[189] Lui Sha et al. Using simplicity to control complexity. *IEEE Software*, 18(4):20–28, 2001.

[190] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 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*, ASPLOS '20, pages 955–970, Lausanne, Switzerland, March 2020.

[191] Hocheol Shin, Dohyun Kim, Yujin Kwon, and Yongdae Kim. Illusion and dazzle: Adversarial optical channel exploits against lidars for automotive applications. In *19th International Conference on Cryptographic Hardware and Embedded Systems*, CHES '17, pages 445–467, Taipei, Taiwan, September 2017. Springer.

[192] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, page 317–328, Xi'an, China, 2016. ACM New York, NY, USA.

[193] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-TCB Linux applications with SGX enclaves. In *Network and Distributed Systems Security Symposium*, NDSS '17, San Diego, CA, USA, February 2017. The Internet Society.

[194] Ian Smith, Anthony Voelim, David Niemi, Jon Tombs, Ben Smith, Rick Grehan, and Tom Yager. Unixbench: the original BYTE UNIX benchmark suite. Available: `https://github.com/kdlucas/byte-unixbench`.

[195] Livio Soares and Michael Stumm. FlexSC: Flexible system call scheduling with Exception-Less system calls. In *9th USENIX Symposium on Operating Systems Design and Implementation*, Vancouver, BC, October 2010. USENIX Association.

[196] Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. CompCertM: Compcert with c-assembly linking and lightweight modular verification. *Proceedings of the ACM on Programming Languages*, 4(POPL), December 2019.

[197] Bailey Srimoungchanh, J. Garrett Morris, and Drew Davidson. Assessing UAV sensor spoofing: More than a GNSS problem. In *2024 Annual Computer Security Applications Conference*, ACSAC '24, pages 1032–1046, 2024.

[198] Tim Starks and David DiMolfetta. Downed U.S. drone points to cyber vulnerabilities. Available: https://www.washingtonpost.com/politics/2023/03/16/downed-us-drone-points-cyber-vulnerabilities/, March 2023.

[199] Ioannis Stellios, Panayiotis Kotzanikolaou, Mihalis Psarakis, Cristina Alcaraz, and Javier Lopez. A survey of IoT-enabled cyberattacks: Assessing attack paths to critical infrastructures and services. *IEEE Communications Surveys & Tutorials*, 20(4):3453–3495, 2018.

[200] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas Devadas, and Sanjit A Seshia. A formal foundation for secure remote execution of enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2435–2450, 2017.

[201] Darius Suciu, Stephen McLaughlin, Laurent Simon, and Radu Sion. Horizontal privilege escalation in trusted applications. In *29th USENIX Security Symposium*, USENIX Security '20. USENIX Association, August 2020.

[202] Sebastian Surminski, Christian Niesler, Ferdinand Brasser, Lucas Davi, and Ahmad-Reza Sadeghi. RealSWATT: Remote software-based attestation for embedded devices under realtime constraints. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, pages 2890–2905, Virtual Event, Repbulic of Korea, November 2021. ACM New York, NY, USA.

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

[204] Richard Ta-Min, Lionel Litty, and David Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 279–292, Seattle, WA, USA, November 2006. USENIX Association.

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

[206] Alex Thomas, Stephan Kaminsky, Dayeol Lee, Dawn Song, and Krste Asanovic. ERTOS: Enclaves in real-time operating systems. In *Fifth Workshop on Computer Architecture Research with RISC-V*, CARRV '21, Virtual, June 2021.

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

[208] TOKI. Blockchain × TEE: Why various forefront projects are adopting TEE. Available: https://medium.com/@tokifinance/blockchain-projects-adapting-tee-bed9550db9c5, 2024.

[209] Sebouh Toumassian, Rico Werner, and Axel Sikora. Performance measurements for hypervisors on embedded ARM processors. In *2016 International Conference on Advances in Computing, Communications and Informatics*, ICACCI '16, pages 851–855. IEEE, 2016.

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

[211] Chia-Che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *USENIX Annual Technical Conference*, USENIX ATC '17, pages 645–658, Santa Clara, CA, July 2017. USENIX Association.

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

[213] Chris Urmson, Joshua Anhalt, Drew Bagnell, Christopher Baker, Robert Bittner, MN Clark, John Dolan, Dave Duggins, Tugrul Galatali, Chris Geyer, et al. Autonomous driving in urban environments: Boss and the urban challenge. *Journal of Field Robotics*, 25(8):425–466, 2008.

[214] Jo Van Bulck, Jan Tobias Mühlberg, and Frank Piessens. VulCAN: Efficient component authentication and software isolation for automotive control networks. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, ACSAC '17, page 225–237, Orlando, FL, USA, 2017. ACM New York, NY, USA.

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

[216] Prasanth Vivekanandan, Gonzalo Garcia, Heechul Yun, and Shawn Keshmiri. A simplex architecture for intelligent and safe unmanned aerial vehicles. In *22nd International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '16, pages 69–75, Daegu, Korea (South), August 2016. IEEE.

[217] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted execution environments on GPUs. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 681–696, Carlsbad, CA, October 2018. USENIX Association.

[218] Jinwen Wang, Ao Li, Haoran Li, Chenyang Lu, and Ning Zhang. RT-TEE: Real-time system availability for cyber-physical systems using ARM TrustZone. In *2022 IEEE Symposium on Security and Privacy*, IEEE S&P '22, pages 352–369, San Francisco, CA, May 2022. IEEE.

[219] Waymo. Waymo safety imapct. Available: `https://waymo.com/safety/impact/`.

[220] Ralf-Philipp Weinmann and Benedikt Schmotzle. TBONE–a zero-click exploit for Tesla MCUs. Technical report, ComSecuris, 2020.

[221] Christian Werling, Niclas Kühnapfel, Hans Niklas Jacob, and Oleg Drokin. Jailbreaking an electric vehicle in 2023. In *Black Hat USA*, Las Vegas, NV, June 2023.

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

[223] Sisu Xi, Meng Xu, Chenyang Lu, Linh T. X. Phan, Christopher Gill, Oleg Sokolsky, and Insup Lee. Real-time multi-core virtual machine scheduling in xen. In *Proceedings of the 14th International Conference on Embedded Software*, EMSOFT '14, New Delhi, India, 2014. ACM New York, NY, USA.

[224] Jean-Paul Yaacoub, Hassan Noura, Ola Salman, and Ali Chehab. Security analysis of drones systems: Attacks, limitations, and recommendations. *Internet of Things*, 11, 2020.

[225] Huaiyu Yan, Zhen Ling, Haobo Li, Lan Luo, Xinhui Shao, Kai Dong, Ping Jiang, Ming Yang, Junzhou Luo, and Xinwen Fu. LDR: Secure and efficient Linux driver runtime for embedded TEE systems. In *Network and Distributed Systems Security Symposium*, NDSS '24, San Diego, CA, February 2024. The Internet Society.

[226] Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36 of *NuerIPS '23*, pages 21573–21612. Curran Associates, Inc., 2023.

[227] Man-Ki Yoon, Bo Liu, Naira Hovakimyan, and Lui Sha. VirtualDrone: virtual sensing, actuation, and communication for attack-resilient unmanned aerial systems. In *Proceedings of the 8th International Conference on Cyber-Physical Systems*, ICCPS '17, pages 143–154, Pittsburgh, Pennsylvania, April 2017. ACM New York, NY, USA.

[228] Man-Ki Yoon, Sibin Mohan, Jaesik Choi, Jung-Eun Kim, and Lui Sha. 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*, RTAS '13, pages 21–32, Philadelphia, PA, USA, June 2013.

[229] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *19th Real-Time and Embedded Technology and Applications Symposium*, RTAS '14, pages 155–166. IEEE, 2014.

[230] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. 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*, RTAS '13, pages 55–64, Philadelphia, PA, USA, June 2013.

[231] Ling Zhang, Yuting Wang, Yalun Liang, and Zhong Shao. CompCertOC: Verified compositional compilation of multi-threaded programs with shared stacks. In *2025 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '23, Seoul, South Korea, June 2025. ACM New York, NY, USA.

[232] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y Thomas Hou. TruSense: Information leakage from TrustZone. In *IEEE Conference on Computer Communications*, INFOCOM '18, pages 1097–1105, Honolulu, HI, April 2018. IEEE.

[233] Shijun Zhao, Qianying Zhang, Yu Qin, Wei Feng, and Dengguo Feng. SecTEE: A software-based approach to secure enclave architecture using TEE. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 1723–1740, London, UK, November 2019. ACM New York, NY, USA.

[234] Alexander Zuepke, Andrea Bastoni, Weifan Chen, Marco Caccamo, and Renato Mancuso. MemPol: Policing core memory bandwidth from outside of the cores. In *29th Real-Time and Embedded Technology and Applications Symposium*, RTAS '23, pages 235–248. IEEE, 2023.

Appendix A

Additional design and implementation details for PARTEE

A.1 PARTEE System Design Details

Memory Protection: User-space virtual-memory access control is sufficient for isolating enclaves from each other spatially; however, for ARM systems, after stage 1 and 2 translations, there are no further MMU mechanisms to prevent the Normal world from accessing Secure world physical memory (until ARMv9). ARM provides a bus-security mechanism, the TrustZone Address Space Controller (TZASC), to configure access to physical memory regions [10]. Some SoC systems, like the NVIDIA Tegra X2 (TX2), have custom bus security mechanisms. The system control fabric (SCF) on the TX2 has a "carveout" of configurable memory regions called "apertures" which can be assigned to some security domain. These bus security devices do not integrate with the cache hierarchy, so the controller only protects when memory requests actually make it to the bus. ARM's solution to this problem is to allow MMU page tables to have a Secure world bit, indicating that the translation should result in Secure-world owned physical memory. If this bit is selected in the page tables, the cache will treat these physical addresses separately (by setting a bit in the physical address) and the bus controller can distinguish between Secure and Normal requests.

Spatial Isolation Without Bus Security: All ARMv7-A and ARMv8-A chips we have seen have the TrustZone extension; however, some (*e.g.* Raspberry Pi 3 and 4) do not have any off-core bus security, so the MMU is the only way to protect memory from being accessed by kernel mode. A simple way to protect TrustZone memory for these devices is to apply similar techniques used by HypSec and seKVM [126, 127]. This works by running a trusted *Corevisor* in `EL2`, which prevents the host Linux's KVM from accessing its nested page tables (NPTs). In our case, the NPTs just need to be configured to remove any entries for Secure-world memory, and we do not need to run a guest VM, so no VM-switch overhead is ever incurred. All other aspects of PARTEE remain the same. We used the seKVM artifact in this way to ensure security for our Raspberry Pi4B.

Interrupt Security: Interrupts are needed for both worlds, but the host OS should not be able to interrupt PARTEE or modify its interrupt configurations. ARM's Generic Interrupt Controller (GIC) [9] has the capability to configure interrupts by world and is used widely. We configure a

small subset of interrupts for the Secure world OS (*e.g.* a timer, I/O devices, DMA etc.), and allocate the remaining for the Normal world. We configure the SCR_EL3 register to effectively disallow the host OS from masking interrupts.

A.1.1 Integrated design: Firmware and Secure OS (EL3/S-EL1):

Due to the way EL3 software is not protected in the MMU, there is no meaningful security difference between EL3 and S-EL1 (until ARMv9), so we combine them for PARTEE. Essentially, world context switches are a special extension on regular context switches, so much of our trap handling is simply reused. We reduce the TCB by observing that traps entering from the Normal world can be a built on generic trap handling infrastructure: all traps need to save general-purpose registers to a trap frame, and traps from the Normal world also need to save the EL1 registers and restore some EL1 registers. In addition to EL3 trap handling, the design needs to handle power state operations that are managed by the firmware, *e.g.* sleeping cores, or powering off the system. The advantage of integrating the designs here allows PARTEE's policy to apply to power state, enforcing access control to these elements.

With this design we also optimize the trap handling over the ARM-TF by observing that entry traps do not need to restore the state of the general-purpose registers and do not need to restore the state of all Secure world kernel registers. Whereas if the EL3 firmware is separate, it must swap all registers. Furthermore, the floating-point registers, do not need to be saved or restored during a trap, but can be swapped lazily if an enclave uses them by trapping on floating point instructions. Finally, similar optimizations are made for trap returns; the end result reduces the number of reads and writes from about 1.9 KB of memory to 0.7 KB (about a 13% reduction in timer interrupt overhead alone). We measured the world switch latency by forcing consecutive secure timer interrupts into EL3 from the Normal world. The latency is then measured as a delta between timer interrupt handler invocations. This has the benefit of letting us normalize any differences in the cache from interrupt to interrupt. We first measure a configuration of PARTEE with the ARM-TF as the EL3 firmware; this took on average 7520 cycles. With the optimized no-ARM-TF implementation we observed 6663 cycles (n=100). This gives a modest performance improvement

	Enter		Return		
Register	**S**	**R**	**S**	**R**	**Rationale**
sp_el0	✓	✓		✓	Restored to fixed value.
ttbr1_el1	✓			✓	Swapped with enclave cswitch.
spsr_el1	✓	→		✓	Clobbered on next trap.
elr_el1	✓	→		✓	Clobbered on next trap.
esr_el1	✓	→		✓	Clobbered on next trap.
far_el1	✓	→		✓	Clobbered on next trap.
afsr0_el1	✓			✓	Unused.
afsr1_el1	✓			✓	Unused.
tpidr_el0	✓			✓	Swapped with enclave cswitch.
tpidrro_el0	✓			✓	Swapped with enclave cswitch.
x0	✓			✓	Clobbered.
...	...			...	...
x30	✓			✓	Clobbered.
q0					Swapped on enclave use.
...					...
q31					Swapped on enclave use.

Figure A.1: Analysis of world switch optimizations made by avoiding saving (S) or restoring (R) registers.

of about 13% for timer interrupt handling with a world switch, which is likely similar for other interrupts.

Our analysis of each optimized ARMv8 register involved in world switching is given in Fig. A.1. In this table a ✓ indicates that a register is saved (**S**) or restored (**R**) on trap enter or trap return during a world switch for PARTEE. A → indicates that a register's value is forwarded, *i.e.* the content's of the EL3 version of the register copied into the parallel EL1 register. For the ARM-TF all registers are saved and restored at both points, so absence of a ✓ in the table indicates an optimization.

A.2 PARTEE's Wait-Free Broadcasting Ring Buffer

The basic building block of communication in PARTEE are *shared-memory* fixed-size ring buffers queues. PARTEE's ring-buffer protocol is a lock-free data structure, which means that all threads in the protocol will complete in a bounded number of instructions. We will use this data structure between user and kernel and between partitions, so we will assume threads can be malicious. With any shared-memory protocol, a malicious thread can corrupt the shared variables; however, in that case this ring buffer will simply return an error or a corrupt, but valid message.

To the best of our knowledge, this exact data structure has not been discussed before, although some variations can be found in the literature [119, 123, 20] and in off-the-shelf products [65]. The real innovation is how we use this data structure in a kernel-mediated way in §2.6; however, there are some unique aspects to its design as is. In particular, this is a publisher-dequeue, or "broadcasting" style of ring buffer; the publishers will always remove the oldest message from the queue before adding a new one. This means subscribers or readers will internally track their own reading progress relative to the current "tail" index, as messages are now broadcast as public record of the past N messages. As we will see in the following section, this allows PARTEE to reduce the maximum number of copies needed for any message to one. In summary, this protocol has the following properties:

- Publishers can safely clobber the oldest elements of the ring, even if a subscriber is actively reading it. The subscriber will gracefully abort in this case.
- Publishing is transactional, no partial or corrupted messages will be written when enclaves follow the protocol.
- Publishing is unlikely to fail on race conditions, *i.e.* the caller does not need to loop until publishing succeeds.
- All loops are either bounded by the ring size or message size, and hence operations have bounded run times.
- Subscribers do not remove elements from the ring; they track their own read elements.
- A stalled or crashed publisher will not infinitely block subscribers, and ring corruption

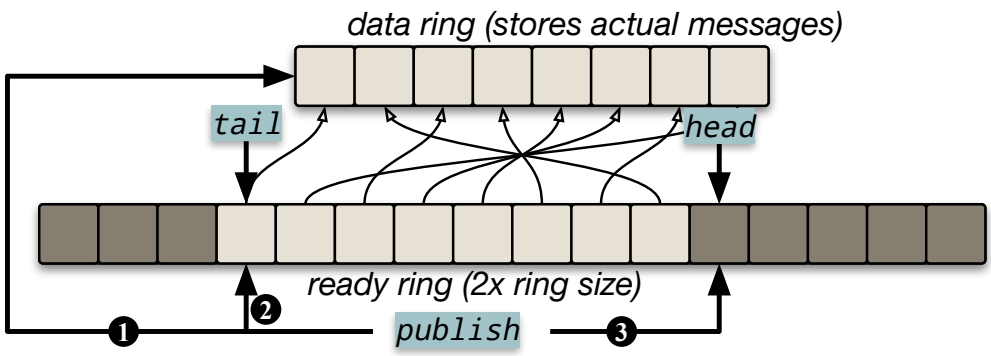

Figure A.2: A diagram of PARTEE's Broadcasting Ring Buffer and an overview of the steps to publish; the dark cells of the buffer indicate that the pointer is `NULL`.

should not cause infinite blocking or further memory corruption.

The Wait-free Broadcasting Publish Protocol: The ring buffer is made up of two arrays as shown in Fig. A.2; at a high level, this division enables transactional publishing. The *Data Ring* array stores the actual messages. It is a `ring_size` array of `slot_size` elements (or "slots"), each with a variable size message up to `slot_size` bytes. The *Ready Ring* array stores timestamped indices into the Data Ring; notably, the Ready Ring is always twice the size of the data ring. This data structure must also store the `ready_tail` and `ready_head`, which are the *monotonically increasing* indices into the Ready Ring. The timestamped indices and monotonically increasing indices are what we use to resolve classic ABA problems [60]. Moreover, we assume that the Data Ring is larger than the number of publishers. This will ensure that simultaneous contention will always be resolved. We go over the steps of the protocol, shown in Fig. A.2, (code in Figs. **??**), below and explain how we achieve our desired properties:

❶ The publisher dequeues the oldest Ready Ring item.

(a) The publisher atomically reads and increments the ready ring tail using an `atomic_fetch_add` operation (Fig. A.4 line 25). Atomicity ensures concurrent publisher dequeues will yield a unique element.

(b) The publisher atomically exchanges `INVALID_INDEX` into the read tail index of the Ready Ring. If this exchange reveals that the array already contained `INVALID_INDEX`, then we retry ❶ up to `n_slots` times. However, this exchange will only "fail" if the

ready ring wraps around entirely between this exchange and the previous `atomic_fetch_add`.

❷ The publisher now has exclusive access to the Data Ring slot previously referenced by `ready_tail`. The index is stored as a timestamped index, so the timestamp bits must be masked-away to read the index. The publisher writes the message into the ring.

❸ Finally the publisher enqueues a the data slot's index + a fresh timestamp.

(a) The publisher reads and atomically increments `ready_head` using an `atomic_fetch_add` operation (Fig. A.3 line 12). Atomicity ensures concurrent publisher enqueues will yield a unique element.

(b) Because the Ready Ring is twice the size of the Data Ring, if all publishers simultaneously attempt to enqueue at the same time, each should be overwriting a `INVALID_INDEX` slot. To verify this, the publisher performs an `atomic_compare_exchange` operation, which confirms that the slot was indeed invalid before writing to it. On failure, retry ❸ up to `n_slots` times, though this will only happen in the unlikely wrapping case.

The Ring Buffer Read Protocol: In our design, the oldest message is dropped on `publish()`; we chose this approach for three main reasons. First, for CPS it is generally not important to have stale data (*e.g.* sensor data) from, for example, 10 seconds ago, and very important to have the latest data. Second, we can guarantee, using the PARTEE Rules and max publish rates in the Topic Firewall from §2.6, that the Data Ring will be large enough to hold all messages published between budget refills of any reader. Therefore, the subscribers will always be scheduled before the ring buffer wraps around. The protocol is as follows:

1. The reader performs two `atomic_load` instructions to read the `ready_tail` then the `ready_head` into local variables (we believe the former may be able to be relaxed). This tail is compared with the reader's internal tail, which is updated only if the `ready_tail` is larger.
2. The reader scans through the array from the internal tail to the local head, looking for slots

not marked `INVALID_ID` by using an `atomic_load` on each element.

3. If the element is valid and is not a stale timestamp, it begins to read the Data Ring at that index.
4. After reading, the reader performs one more `atomic_load` on the `ready_ring` at the same index as before. If the load matches the original timestamp, we have successfully read the entire message.

Optionally, the reader can track read messages using a local ring buffer. This allows for out-of-order reads to happen and can be useful in some cases. Otherwise, slow or stalled publishers could be skipped over by updating the local tail upon successful read. If order is absolutely important, the reader can be configured to stop scanning once it finds the first invalid slot.

```c
int
partee_topic_publish(
        struct partee_publisher * pub,
        struct partee_msg * msg)
{
    if(pub == NULL || msg == NULL || msg->slot == NULL) return -1;

    for(size_t retries = 0; retries < pub->rings.n_slots; retries++)
    {
        /* allocate a spot in the ready ring */
        /* this head ptr is NULL, gating readers */
        size_t head = atomic_fetch_add_explicit(
                pub->rings.ready.head_shared,
                1,
                memory_order_seq_cst);

        /* update the data_timestamp with this cycle */
        size_t data_timestamp = (msg->slot_timestamp & (pub->rings.n_slots - 1)) +
            (head & ~(pub->rings.n_slots - 1));

        /* We should be writing to an unused slot */
        size_t expected = INVALID_TIMESTAMP;
        if(atomic_compare_exchange_strong_explicit(
            partee_get_ready_slot(&pub->rings, head),
            &expected,
            data_timestamp,
            memory_order_seq_cst, memory_order_relaxed))
        {
            pub->n_allocated--;
            return 0;
        }

        /* if the ready slot at head is not INVALID, then the ready ring wrapped
         * around, we should retry, as this is unlikely unless some publisher
         * thread is flooding the ring with messages.
         */
    }

    output("%s: Publishing failed.\n", pub->topic_name);

    /* if we failed to publish (unlikely), we free the slot to a non-shared queue */
    if(partee_free_unpublished(pub, msg))
    {
        return -2;
    }

    return -3;
}
```

Figure A.3: Code fragments of PARTEE publisher shared-memory protocol, showing that no unbounded loops are present due to race conditions.

```c
int
partee_msg_alloc(
        struct partee_publisher *pub,
        struct partee_msg *msg)
{
    if(pub == NULL || msg == NULL) { return -1; }
    if(pub->n_allocated >= pub->max_allocations)
    {
        output("%s: Too many messages allocated %zu / %zu.\n", pub->topic_name,
                pub->n_allocated, pub->max_allocations);
        msg->slot = NULL;
        msg->rings = NULL;
        msg->data_size_cached = 0;
        return -2;
    }

    /* unlikely, but check if we have any previous failed publishes */
    if(partee_check_unpublshed(pub, msg))
    {
        return 0;
    }

    for(size_t retries = 0; retries < pub->rings.n_slots; retries++)
    {
        size_t ready_tail = atomic_fetch_add_explicit(
            pub->rings.ready.tail_shared,
            1,
            memory_order_seq_cst);

        /* Now we try to pop the data slot from the ready ring */
        msg->slot_timestamp = atomic_exchange_explicit(
            partee_get_ready_slot(&pub->rings, ready_tail),
            INVALID_TIMESTAMP,
            memory_order_seq_cst);

        /* we either successfully acquired a data_slot, or the ready ring
         * wrapped around, due to fast publishers */
        if(msg->slot_timestamp == INVALID_TIMESTAMP)
        {
            continue;
        }

        msg->slot = partee_get_data_slot(&pub->rings, msg->slot_timestamp);
        msg->rings = &pub->rings;
        msg->data_size_cached = 0;
        pub->n_allocated++;

        return 0;
    }

    msg->slot = NULL;
    msg->rings = NULL;
    return -3;
}
```

Figure A.4: Code fragments of PARTEE publisher shared-memory protocol, showing that no unbounded loops are present due to race conditions.