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

Richard Habeeb*, Hao Chen*, Man-Ki Yoon†, Zhong Shao*
*Yale University, †North Carolina State University

***Abstract*—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.

***Index Terms*—TrustZone, TEE, Robotics, CPS, Availability**

1. Introduction

Next-gen robotics and cyber-physical systems (CPS) increasingly rely on machine learning and large language models integrated with frameworks like the Robot Operating System (ROS) atop multicore system-on-chip (SoC) devices [1]–[3].[1] These systems feature highly distributed software architectures composed of many third-party processes communicating over a publish/subscribe framework [4], [5]. 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 [6]–[13]. With increased complexity and Internet connectivity, many are at risk of privilege escalation once an adversary gains a foothold [14]–[20]. 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), often called *enclaves*, offer a lightweight solution by isolating software from an untrusted host operating system (OS) [21], [22], but these are generally not designed to provide the availability guarantees needed by CPS [23]. While some TEEs can provide a basis for *CPU availability* (access to CPU time, *e.g.* using ARM TrustZone [24] with hierarchical scheduling [25]–[27]), achieving stronger availability for critical applications is tricky on a modern TEE. Of the platforms that need a TEE OS (like OP-TEE [28]), 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 [29]–[43], 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 [28], [29], [37], [44], 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 [28] heavily uses a kernel heap for numerous types of objects which can be drained by the untrusted OS or a faulty enclave (demonstrated in §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 §3). Furthermore, TEE IPC does not integrate well ROS-like patterns, leaving enclaves isolated.

1. ROS is not a traditional OS, it is a user-space framework and middleware built on top of Ubuntu, allowing for rapid development.

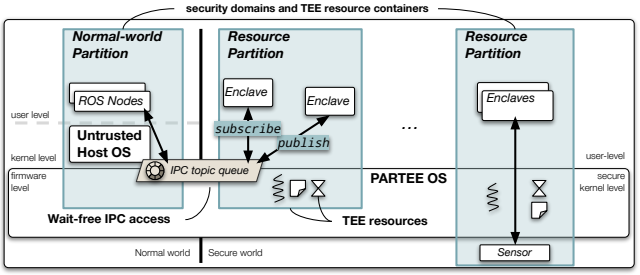

Figure 1. PARTEE prevents DoS attacks by dividing TEE resources into containers called *availability partitions* which also act as security domains and hierarchical scheduling partitions; IPC availability is protected and made practical for distributed ROS architectures via a wait-free broadcasting mechanism.

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 (§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 §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. 1, PARTEE provides a new type of resource container [45] called an *availability partition* (or just "partition") which divides up TEE OS services and objects, physical memory, CPU time, device I/O, and IPC access into isolated security domains. Critical applications run in enclaves, and each enclave belongs to one of these partitions with dedicated resources. Partitioning enables system developers to clearly define the security domain for each potential 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. Thus, when compared with directly running on Ubuntu/ROS, PARTEE offers a massively reduced trusted computing base (TCB) for critical tasks.

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, and 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 [25], [26], [46]. 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 sources. It must support numerous complex consumer services using unreliable, insecure connections like cellular, Wifi, and Bluetooth. This device also provides 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 [14], [19], [47]. PARTEE paves the way for the protection of this type of critical use case by partitioning time-sensitive subsystems into lightweight enclaves with 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 a malicious message. Summarizing, our contributions are:

- *A novel availability-partitioning TEE OS:* We built PARTEE to protect enclave availability from the start by considering DoS attacks at each layer of its design (§5), ensuring timely access to CPU, critical I/O, IPC, and dynamic memory.
- *A fast enclave interface supporting the reliability needs of CPS:* We developed an optimized enclave IPC system that can be seamlessly integrated with ROS-like publish-subscribe middleware; it ensures IPC availability between honest partitions, message integrity, and fine-grained access controls via partitioning (§6).
- *Real-world implementation and attacks*: We analyze the availability challenges for modern TEE OSes using three example attacks (§3) and test our implementation of PARTEE on two popular platforms: the NVIDIA Jetson TX2 and Raspberry Pi4B, using our attack tools. We further test PARTEE's security and performance by constructing a drone implementation on the Pi (§7, 9).

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 [48], [49]. For instance, recent security analyses showed that Tesla's full self-driving autopilot runs Linux and is connected to both the Internet (for data collection) and the vehicle's controller area network (CAN) bus of critical vehicle control systems [50], [51]. Many others have shown the potential for privilege escalation once a foothold is gained on a CPS [14], [15], [17]–[20].

Our drone implementation (show in Fig. 10 in §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 §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 [52], if one subsystem is compromised or has a fault, the other systems ideally should remain as unaffected as possible [53]; thus, a simple binary division of security domains will not always provide strong security. The ARINC 653 standard [54]–[56] and ongoing research on partitioning hypervisors are based on this premise [57]. One notable example is the Boss self-driving architecture [58], 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 [59]–[62]. 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 §4.2). This allows us to model potentially malicious third-party enclaves.

3. Availability Challenges for TEE OSes

How are TEEs constructed which makes it difficult to modify them to support the problems outlined in §1? To answer this question, we look at the highly popular OP-TEE [28] 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 via remote procedure call (RPC) and waits for it to return. This is one of nine essential RPCs part 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.

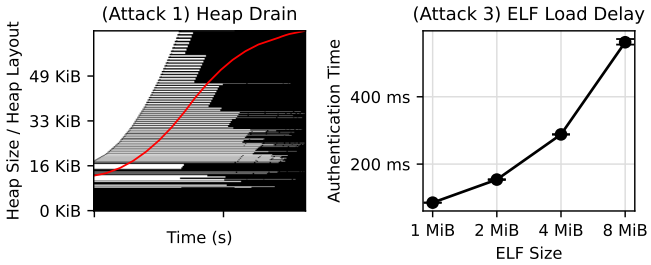

Figure 2. **Left:** Heap space over time during a heap-drain attack on OP-TEE OS; black space is allocated memory and the red-line indicates overall capacity. **Right:** Measurements of malicious ELF loading time for OP-TEE by file size, showing linear scaling between size and hash time.

Even though many have proposed ways to use the ARM TrustZone for real-time applications [26], [46], [63]–[66], only one recent work, RT-TEE [25], has availability measures for modern TEEs specifically [21]. 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 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 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

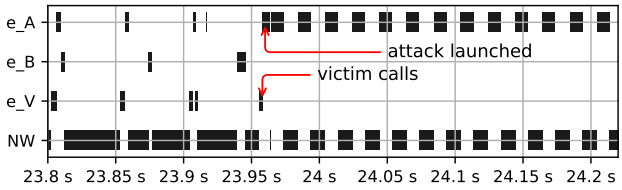

Figure 3. Task trace of an execution-blocking attack implementation on RT-TEE [25] where victim $e_{\mathcal{V}}$ calls adversary $e_{\mathcal{A}}$, which never returns, causing $e_{\mathcal{V}}$ to miss its deadline. Periodic budget refills simply resume $e_{\mathcal{A}}$.

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 [67]–[69]) 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 synchronously (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, for 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 [62]. 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 [25]). Fig. 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's ELF loader to act as a confused deputy [70]: 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.Fig. 2 shows our measurements for how long this hashing process could take, scaling linearly with the ELF size. The delay is due to hashing and copying overheads of data for 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.

4. PARTEE Goals and Security Models

Using our motivating attacks above, we define several goals beyond prior work, which are later analyzed in §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.*

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 [71] platforms as 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 [72]. Interrupts can be configured to be owned by a certain world and can be disabled during the non-owning world's execution.

4.2. Adversary Model

Using our informal attacks from §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 (§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 §6 for in-depth design discussions on this issue, and see §7 for how these affected our drone implementation. We now describe PARTEE's challenge to this adversary in the following sections.

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 [71]) and can improve world switch latency (see §A). Normal hypervisor mode (`EL2`) is not needed for PARTEE (as long as the hardware model in §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 [45], 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. 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 PARTEE at runtime to ensure isolation; they specify the

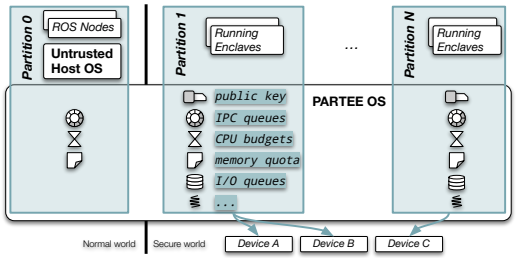

Figure 4. PARTEE's partitioning system design where TEE OS managed resources and objects are partitioned among running enclaves.

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 CPU time partitioned?*

PARTEE OS uses well-known TrustZone techniques [25], [26], [46], [64] 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).

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 [72]), see §A for details.

5.1. Availability Partitioning in the TEE OS

What types of resources must be available for an enclave? To answer this question systematically, we use the TEE 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 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. 5. In general this approach to design could apply to any OS that should provide availability; however, for a TEE OS, we have additional challenges related to interactions and resource sharing with the host OS.

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 [73], [74] with carefully bounded critical sections to ensure lock-acquire ordering.

Asynchronous device I/O: PARTEE offers 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.

Using the rules, partitions are configured with access to hardware devices; however, for shared devices one partitions could DoS another by saturating PARTEE's I/O drivers with time-consuming operations. To prevent long-running device I/O operations from blocking a core, we implemented generic I/O queuing system calls as the primary interface for critical devices from enclaves. Outgoing I/O is queued up in each partition for each device (using each partition's own memory). For devices shared by multiple partitions, the OS divides 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. 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

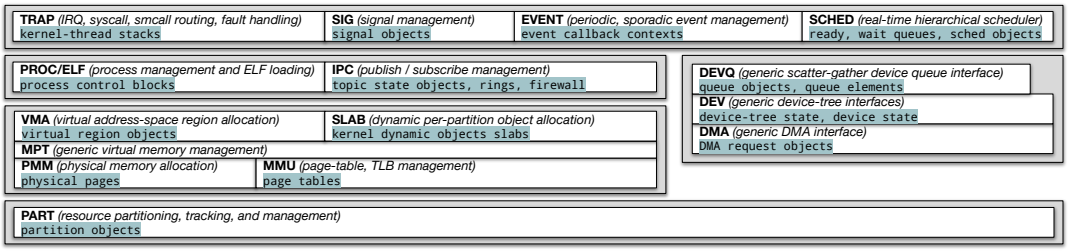

Figure 5. Breakdown of core PARTEE OS design layers showing what enclave availability-relevant state is managed at each abstraction to prevent DoS

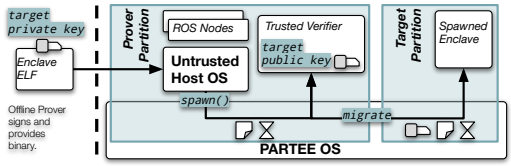

Figure 6. Resources needed to authenticate and spawn an enclave in a different partition must be provided by the "prover" partition (or parent) until the signature check is complete.

after authentication is the enclave charged to the destination partition.

6. Availability, Integrity, Performance, and Usability for TEE Communication

For communication among enclaves and the rest of the system, PARTEE mediates asynchronous message passing over shared memory to provide: strong availability and integrity properties, seamless integration with ROS [1], and reliably low latencies. At a high level, PARTEE uses a DDS-like communication style, where enclaves publish and subscribe to named topics [75], [76] via a TEE-OS-mediated IPC protocol which prevents 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 builds upon well-known implementations; the design is provided in §B.

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. The backup or alert operation logic that are required for graceful failure should be implemented within enclaves, and any devices required by these operations must be made accessible via a PARTEE driver; thus, the backup logic has no timing or resource dependencies on the untrusted host OS and can execute gracefully if everything else fails.

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 appending 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 [77]. Research is ongoing to investigate the problem of maliciously spoofed sensor data or ML model robustness attacks [78], [78]–[82]. The Simplex approach has been explored to validate complex controller outputs [83]–[90]. Other approaches use physics modeling or hypothesis testing to identify anomalous spoofed data [77]. In some cases, data from trusted devices could be authenticated [91], [92]. For instance, if LiDAR data is published from an untrusted ROS node, it is possible to detect tampering via watermarking [93]–[95]; 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 §B 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.

The 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

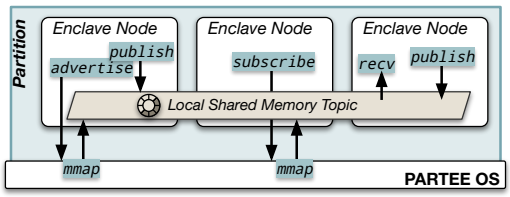

Figure 7. PARTEE's intra-partition publish-subscribe over pure shared memory incurs no TEE OS overhead for local message transfers.

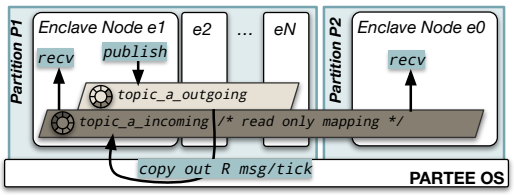

Figure 8. PARTEE's inter-partition publish-subscribe protocol enforces security efficiently by rate limiting with a single copy from partition-local outgoing to global incoming ring buffers

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 size. If the quota cannot fulfill the queue size, then the enclave's request system call (*e.g.* `advertise`/`subscribe` calls to create queues) returns a failure. 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. With this optimization, the developer makes a trade-off: improved performance for less runtime safety; for example, a faulting enclave could corrupt messages of others in the same partition. To help this situation, PARTEE takes a best-effort approach by ensuring that the shared-memory protocol is non-blocking and wait-free (see §B) and enclaves do not share address spaces; thus, faulting enclaves still cannot block each other or corrupt each other's code and private data.

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. 8. All enclaves in a partition share an outgoing ring per topic, and

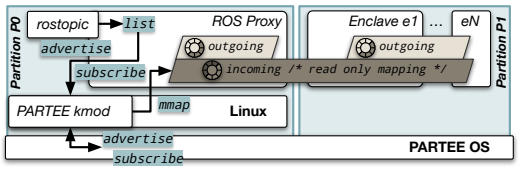

Figure 9. Diagram of how PARTEE integrates with ROS, automatically fetching topic lists from ROS and creating PARTEE topics, and *vice versa*, and publishing messages back and forth

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:

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

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

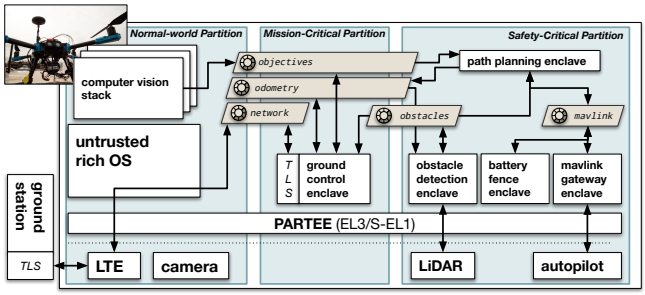

Figure 10. Overview of highly-secure PARTEE drone implementation where control tasks are divided into partitioned enclave processes. Enclaves publish and subscribe to isolated ROS-compatible topics to transfer data, and they interface with their partition's devices—thereby protecting availability, integrity, and confidentiality for enclaves and IPC.

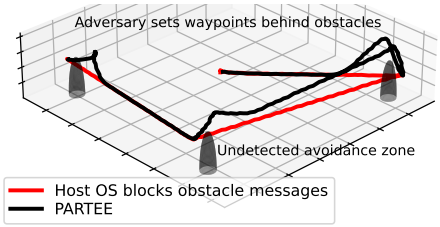

Figure 11. Comparison of drone flight paths: Insecure version where the host OS blocks obstacle-avoidance IPC, and a secure version using PARTEE, which guarantees IPC delivery. The insecure drone does not detect the obstacle and does not avoid flying too close.

7. Case Study: Secure Partitioned Drone

We tested PARTEE on a drone software architecture (overviewed in Fig. 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 [10], [12], [18], [96], [97]. The drone uses an off-the-shelf quad-copter frame with a Pixhawk CubePilot Orange autopilot device and a Raspberry Pi4B companion computer. 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).

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 [98]). This enclave spawns children to handle concurrent network connections.

The MAVLink-gateway (enclave) task interfaces with the autopilot via MAVLink [99] (via a PARTEE UART driver), and proxies the packets. This interface directly controls the drone and provides odometry data. This task is *safety critical* since it can arbitrarily control the drone's movements.

The obstacle-detection (enclave) task reads a LiDAR sensor to detect obstacles (and publish detections). It uses odometry to smooth noise from the LiDAR from frame-to-frame and coalesce detections. It is *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 MAVLINK gateway (over a "mavlink" topic). This enclave is also safety critical.

The battery-fence (enclave) task reads the battery levels and publishes a return-to-landing command before the battery would be exhausted on return.

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. 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. 11 shows two flight traces. In the 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 hit the obstacle. The result is that the drone did not detect or avoid obstacles, flying through the no-fly danger radius around them. The black trace shows exactly the same source code, recompiled to run inside PARTEE enclaves. The *path-planner* and *obstacle-avoidance* enclaves continue to run, regardless of the actions of Linux.

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 §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.

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 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 §4.

TABLE 1. SOURCE LINE COUNTS FOR THE PARTEE AND OP-TEE TRUSTED COMPUTING BASES.

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

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 allocations 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. 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 §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 §B, 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 §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.

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. Impacts of PARTEE on Host OS: To evaluate the effect of PARTEE on Normal world performance, we use both UnixBench [100] and MiBench [101] 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. We chose this setup so we could assess the impacts of interrupting Linux on the caches. Fig. 12 shows the results. Note that the PARTEE rules allow the CPU budget of the Normal world on to be capped for availability purposes, but for this we allowed 100% utilization for the Normal world (when the cap is lowered, we observed that the benchmark performance would proportionally degrade).

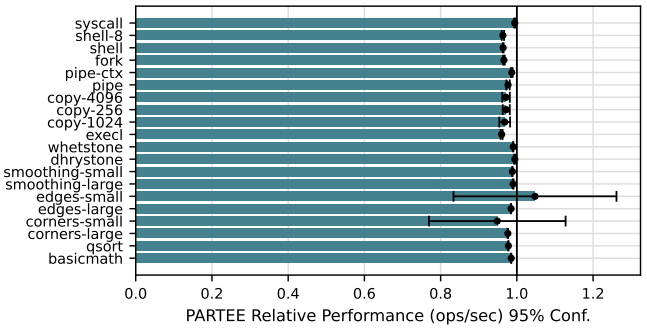

Figure 12. **Host OS performance with PARTEE relative to the baseline (larger is better):** Impacts (measured as throughput) of PARTEE on UnixBench and MiBench applications executed on Linux

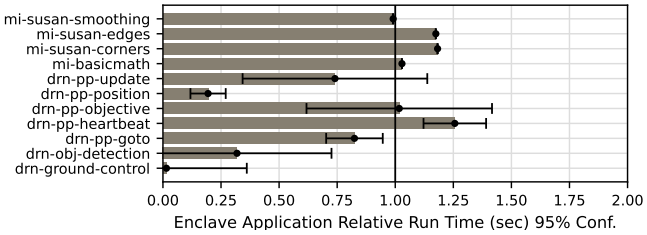

Figure 13. **Enclave application run times normalized against non-enclave run times (smaller is better):** Benchmarks run in enclaves demonstrate overheads for memory-intensive operations, but communication-intensive drone functions typically had improved performance over the host OS due to PARTEE's IPC.

We find that our prototype implementation incurs an overhead of 0.5-4%; our analysis is that this can be largely eliminated with additional engineering work. From our testing, this overhead is due to a speculative execution attack mitigation that occurs on traps from the Normal world; in our prototype we are interrupting periodically using the secure timer. We measured the world switch time at about 5 μs, which is about 0.5% of the timer interval, but each interrupt resets the branch predictor, so an additional loss in performance occurs; we confirmed this with a quick test to remove the predictor reset on timer interrupts which lowered the maximum overhead to around 1%, with most benchmarks having non-statistically significant overhead. Thus, this overhead could be largely eliminated if we upgraded PARTEE to use a *tickless* scheduler—eliminating the extra interrupts and branch predictor resets.

Q2. Impacts of PARTEE on application performance: 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 [101] and ported them to each be PARTEE enclaves. The results are shown in Figure 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 PARTEE could improve on native performance, for several

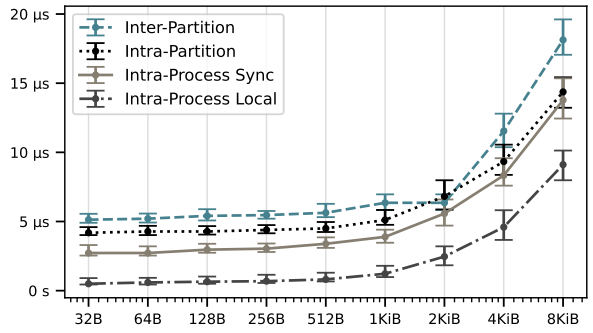

Figure 14. **Single-message latency:** Comparison of PARTEE's end-to-end latency for inter-partition, intra-partition, and intra-process scenarios; PARTEE's local-topic optimization lowers copying overhead scaling (n=128).

communication-heavy operations. This is due to PARTEE's use of shared memory broadcasting; shared memory in general is not always available for ROS middleware and 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, Q4. Message latency: Our end-to-end message transfer time is shown in Fig. 14. This experiment measures 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.

10. Related Work

Static-partitioning and TrustZone-assisted hypervisors: Some high-assurance CPS employ temporal and spatial partitioning to isolate mixed-criticality workloads [49], [53], [55], protecting one or more real-time OS (RTOS) from a rich untrusted OS. Solutions such as static partitioning hypervisors [54], [57], [102]–[105] or TrustZone-assisted designs [26], [27], [46], [63]–[66], [106]–[111] 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 VM-switch trade offs [112], [113]. For example, Jailhouse [103] trades the cost of VM exits for the usage of an entire core, 25% of the Raspberry Pi4B's CPU power, and μRTZVisor [64] requires

a full cache flush on world switches. (2) *Over-privileged tasks.* Many RTOSes are not designed for security, and will run all tasks in a single address space in kernel mode. For the ARM TrustZone, a faulty or hostile task could then own the entire system [59]–[62]. (2) *Engineering overheads.* Tasks in different partitions must use inter-VM communication [110], [114], [115]. Hence, for ROS-based systems, ad-hoc protocols are necessary to transport published data across VMs and real-time OS interfaces. Ultimately, these hypervisors can provide isolation, but they can also suffer from steep performance and engineering overheads.

SMACCM and seL4: The SMACCM architecture [116], [117] for the DARPA HACMS project [118], aims to build a highly-secure drone with from many layer of formally verified software. If we focus on SMACCM's seL4-based [119] 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 partitioned dynamic memory, dynamic spawning of enclaves, publish-subscribe IPC and topic creation, asynchronous device I/O, and other essential features. Without PARTEE, this architecture is vulnerable to IPC blocking attacks.

Hardware 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 [120]–[123] 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 version of seL4 that is hardened against microarchitectural timing channels [119], [123].

Alternative hardware TEEs: Using a hardware-software co-design approach, past works have built real-time enclaves on small embedded devices which have some availability protections [124]–[126]. Notably, Aion [125] showed how to run enclaves on top of an untrusted OS (RIOT OS) by running the scheduler inside an enclave. While Aion protects shared device resources by also placing them in enclaves; if an enclave application needs to make any other type of system call—like IPC managed by the untrusted OS, then the enclave would be at risk of DoS. Because the problem of enclave availability can go well beyond scheduling (as shown in §3), we chose a wholistic approach with PARTEE. Other works: TyTAN [127], TrustLite [128], ERTOS [129], and Keystone [130] delegate scheduling to the untrusted OS; thus, these four cannot provide availability if the OS is compromised (even though they are providing integrity protections for real-time tasks).

Trusted I/O: One area for future research for PARTEE is improving enclave access to dedicated devices. Recent work on the Linux Device Runtime (LDR) [131], 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. GPUReplay [132], MyTEE [133], RT-TEE [25], and others [134], [135] also have potential solutions that complement PARTEE OS to enable more complex I/O.

Protections against malicious enclaves: Recent works vTZ [136], TEEV [59], 3rdParTEE [137], ReZone [61] 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 [138].

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.

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

References

[1] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng *et al.*, "ROS: an open-source robot operating system," in *ICRA Workshop on Open Source Software*. Kobe, Japan, 2009.

[2] A. 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/, Sep. 2022.

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

[4] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Computing Surveys (CSUR)*, vol. 35, no. 2, pp. 114–131, 2003.

[5] I. Malavolta, G. Lewis, B. Schmerl, P. Lago, and D. 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*, ser. ICSE-SEIP '20, Seoul, South Korea, 2020, pp. 31–40.

[6] T. Bonaci, J. Yan, J. Herron, T. Kohno, and H. J. 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*, ser. ICCPS '15. Seattle, Washington: ACM New York, NY, USA, 2015, p. 11–20.

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

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

[9] I. Stellios, P. Kotzanikolaou, M. Psarakis, C. Alcaraz, and J. Lopez, "A survey of IoT-enabled cyberattacks: Assessing attack paths to critical infrastructures and services," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 4, pp. 3453–3495, 2018.

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

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

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

[13] D. Bhamare, M. Zolanvari, A. Erbad, R. Jain, K. Khan, and N. Meskin, "Cybersecurity for industrial control systems: A survey," *Computers & Security*, vol. 89, no. 101667, Feb. 2020.

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

[15] T. A. Daan Keuper, "The connected car: Ways to get unauthorized access and potential implications," Available: https://www.computest.nl/documents/9/The_Connected_Car._Research_Rapport_Computest_april_2018.pdf, Computest, Tech. Rep., 2018.

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

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

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

[19] D. Berard and V. 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.

[20] ——, "I feel a draft. opening the doors and windows 0-click RCE on the Tesla Model3," in *Hexacon*, Vancouver, BC, Oct. 2022.

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

[22] D. Feng, Y. Qin, W. Feng, W. Li, K. Shang, and H. Ma, "Survey of research on confidential computing," *IET Communications*, vol. 18, no. 9, pp. 535–556, 2024. [Online]. Available: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/cmu2.12759

[23] F. Alder, G. Scopelliti, J. Van Bulck, and J. T. Mühlberg, "About time: On the challenges of temporal guarantees in untrusted environments," in *Proceedings of the 6th Workshop on System Software for Trusted Execution*, ser. SysTEX '23. Rome, Italy: ACM New York, NY, USA, 2023, pp. 27–33.

[24] B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin, "TrustZone explained: Architectural features and use cases," in *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, 2016, pp. 445–451.

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

[26] D. Sangorrin, S. Honda, and H. Takada, "Dual operating system architecture for real-time embedded systems," in *6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, ser. OSPERT '10, Jul. 2010.

[27] ——, "Integrated scheduling for a reliable dual-OS monitor," *IPSJ Transactions on Advanced Computing Systems*, vol. 5, no. 2, pp. 99–110, 2012.

[28] Linaro, "Open portable trusted execution environment," Available: https://github.com/OP-TEE/optee_os.

[29] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 2, pp. 2–13, Mar. 2008.

[30] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. 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*, ser. ASPLOS '13. Houston, Texas, USA: ACM New York, NY, USA, Mar. 2013, pp. 265–278.

[31] J. Criswell, N. Dautenhahn, and V. 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*, ser. ASPLOS '14. Salt Lake City, Utah, USA: ACM New York, NY, USA, Feb. 2014.

[32] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry, "MiniBox: A two-way sandbox for x86 native code," in *2014 USENIX Annual Technical Conference*, ser. USENIX ATC '14. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 409–420. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/li_yanlin

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

[34] ——, "Shielding applications from an untrusted cloud with Haven," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 3, pp. 1–26, 2015.

[35] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'keeffe, M. L. Stillwell *et al.*, "SCONE: Secure Linux containers with Intel SGX," in *12th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '16. Savannah, GA: USENIX Association, Nov. 2016, pp. 689–703. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov

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

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

[38] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A practical library OS for unmodified applications on SGX," in *USENIX Annual Technical Conference*, ser. USENIX ATC '17. Santa Clara, CA: USENIX Association, Jul. 2017, pp. 645–658. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai

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

[40] D. Ji, Q. Zhang, S. Zhao, Z. Shi, and Y. 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)*, ser. TrustCom '19, 2019, pp. 26–33.

[41] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia, and S. 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*, ser. ASPLOS '20, Lausanne, Switzerland, Mar. 2020, pp. 955–970.

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

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

[44] V. Costan and S. Devadas, "Intel SGX explained," Cryptology ePrint Archive, Paper 2016/086, 2016. [Online]. Available: https://eprint.iacr.org/2016/086

[45] G. Banga, P. Druschel, and J. 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*, ser. OSDI '99. New Orleans, Louisiana, USA: USENIX Association, 1999.

[46] S. Pinto, J. Pereira, T. Gomes, A. Tavares, and J. Cabral, "LTZVisor: TrustZone is the key," in *29th Euromicro Conference on Real-Time Systems*, ser. ECRTS '17, M. Bertogna, Ed., vol. 76, Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 4:1–4:22. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2017/7153

[47] C. Miller, "Remote exploitation of an unaltered passenger vehicle," in *Black Hat USA*, Las Vegas, NV, Aug. 2015. [Online]. Available: https://ioactive.com/pdfs/IOActive_Remote_Car_Hacking.pdf

[48] A. Burns and R. I. Davis, "A survey of research into mixed criticality systems," *ACM Computing Surveys (CSUR)*, vol. 50, no. 6, pp. 1–37, 2017.

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

[50] N. Kühnapfel, C. Werling, and H. N. Jacob, "Recovering critical data from tesla autopilot using voltage glitching," in *37th Chaos Communication Congress*, ser. 37C3, Hamburg, Germany, Dec. 2023.

[51] C. Werling, N. Kühnapfel, H. N. Jacob, and O. Drokin, "Jailbreaking an electric vehicle in 2023," in *Black Hat USA*, Las Vegas, NV, Jun. 2023.

[52] J. Saltzer and M. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[53] J. M. Rushby, "Design and verification of secure systems," *ACM SIGOPS Operating Systems Review*, vol. 15, no. 5, pp. 12–21, dec 1981.

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

[55] J. Rushby, "Partitioning in avionics architectures: Requirements, mechanisms, and assurance," NASA, Tech. Rep., 1999.

[56] J. Littlefield-Lawwill and L. Kinnan, "System considerations for robust time and space partitioning in integrated modular avionics," in *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*. IEEE, 2008, pp. 1–B.

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

[58] C. Urmson, J. Anhalt, D. Bagnell, C. Baker, R. Bittner, M. Clark, J. Dolan, D. Duggins, T. Galatali, C. Geyer *et al.*, "Autonomous driving in urban environments: Boss and the urban challenge," *Journal of Field Robotics*, vol. 25, no. 8, pp. 425–466, 2008.

[59] W. Li, Y. Xia, L. Lu, H. Chen, and B. Zang, "Teev: Virtualizing trusted execution environments on mobile platforms," in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '19. ACM New York, NY, USA, 2019, pp. 2–16.

[60] D. Suciu, S. McLaughlin, L. Simon, and R. Sion, "Horizontal privilege escalation in trusted applications," in *29th USENIX Security Symposium*, ser. USENIX Security '20. USENIX Association, Aug. 2020.

[61] D. Cerdeira, J. Martins, N. Santos, and S. Pinto, "ReZone: Disarming TrustZone with TEE privilege reduction," in *31st USENIX Security Symposium*, ser. USENIX Security '22, 2022, pp. 2261–2279.

[62] M. Busch, P. Mao, and M. Payer, "GlobalConfusion: TrustZone trusted application 0-days by design," in *33rd USENIX Security Symposium*, ser. USENIX Security '24, 2024, pp. 5537–5554.

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

[64] J. Martins, J. Alves, J. Cabral, A. Tavares, and S. Pinto, "μRTZVisor: A secure and safe real-time hypervisor," *Electronics*, vol. 6, no. 4, 2017.

[65] P. Dong, A. Burns, Z. Jiang, and X. 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*, ser. RTCSA '18. Hakodate, Japan: IEEE, Aug. 2018, pp. 59–64.

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

[67] *GlobalPlatform Device Technology TEE Client API Specification Version 1.0*, GlobalPlatform, Jul. 2010.

[68] *GlobalPlatform Technology TEE Internal Core API Specification Version 1.1.2.50*, GlobalPlatform, Jun. 2018.

[69] *GlobalPlatform Technology TEE System Architecture Version 1.2*, GlobalPlatform, Nov. 2018.

[70] N. Hardy, "The Confused Deputy: (or why capabilities might have been invented)," *ACM SIGOPS Operating Systems Review*, vol. 22, no. 4, pp. 36–38, 1988.

[71] ARM, "ARM Trusted Firmware," Available: https://github.com/ARM-software/arm-trusted-firmware.

[72] *ARM CoreLink TZC-400 TrustZone Address Space Controller Technical Reference Manual*, ARM, 2014. [Online]. Available: https://developer.arm.com/documentation/ddi0504/c/

[73] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems (TOCS)*, vol. 9, no. 1, pp. 21–65, 1991.

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

[75] G. Pardo-Castellote, "OMG data-distribution service: Architectural overview," in *23rd International Conference on Distributed Computing Systems Workshops*, ser. ICDCSW '03. IEEE, 2003, pp. 200–206.

[76] T. Guesmi, R. Rekik, S. Hasnaoui, and H. Rezig, "Design and performance of DDS-based middleware for real-time control systems," *International Journal of Computer Science and Network Security (IJCSNS)*, vol. 7, no. 12, pp. 188–200, 2007.

[77] B. Srimoungchanh, J. G. Morris, and D. Davidson, "Assessing UAV sensor spoofing: More than a GNSS problem," in *2024 Annual Computer Security Applications Conference*, ser. ACSAC '24, Honolulu, HI, USA, 2024, pp. 1032–1046.

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

[79] A. Costin, A. 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, Jul. 2012.

[80] A. J. Kerns, D. P. Shepard, J. A. Bhatti, and T. E. Humphreys, "Unmanned aircraft capture and control via GPS spoofing," *Journal of Field Robotics*, vol. 31, no. 4, pp. 617–636, 2014.

[81] W. Jia, Z. Lu, H. Zhang, Z. Liu, J. Wang, and G. Qu, "Fooling the eyes of autonomous vehicles: Robust physical adversarial examples

against traffic sign recognition systems," in *Network and Distributed Systems Security Symposium*, ser. NDSS '22. San Diego, CA: The Internet Society, Apr. 2022.

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

[83] L. Sha *et al.*, "Using simplicity to control complexity," *IEEE Software*, vol. 18, no. 4, pp. 20–28, 2001.

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

[85] M.-K. Yoon, S. Mohan, J. Choi, J.-E. Kim, and L. 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*, ser. RTAS '13, Philadelphia, PA, USA, Jun. 2013, pp. 21–32.

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

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

[88] M. Hasan and S. 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*, ser. IoT S&P'19. London, United Kingdom: ACM New York, NY, USA, 2019, p. 8–14.

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

[90] B. Luo, S. Ramakrishna, A. Pettet, C. Kuhn, G. Karsai, and A. 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)*, ser. ICCPS '23, San Antonio, TX, USA, 2023, p. 177–186.

[91] J. Van Bulck, J. T. Mühlberg, and F. Piessens, "VulCAN: Efficient component authentication and software isolation for automotive control networks," in *Proceedings of the 33rd Annual Computer Security Applications Conference*, ser. ACSAC '17. Orlando, FL, USA: ACM New York, NY, USA, 2017, p. 225–237.

[92] G. Scopelliti, S. Pouyanrad, J. Noorman, F. Alder, C. Baumann, F. Piessens, and J. T. Mühlberg, "End-to-end security for distributed event-driven enclave applications on heterogeneous TEEs," *ACM Transactions on Privacy and Security*, vol. 26, no. 3, Jun. 2023.

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

[94] J. Liu and J.-M. Park, ""seeing is not always believing": Detecting perception error attacks against autonomous vehicles," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2209–2223, 2021.

[95] M. Marazzi, S. Longari, C. Michele, and S. Zanero, "Securing LiDAR communication through watermark-based tampering detection," in *Symposium on Vehicles Security and Privacy*, ser. VehicleSec '24, 2024.

[96] DJI, "Drone security white paper (version 3.0)," Available: https://www.dji.com/trust-center/resource/white-paper, DJI, Tech. Rep., Apr. 2024.

[97] P. Cabrera, "Parrot drones hijacking," Available: https://www.rsaconference.com/Library/presentation/USA/2018/parrot-drones-hijacking, Apr. 2018.

[98] D. 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/, Nov. 2023.

[99] A. Koubâa, A. Allouch, M. Alajlan, Y. Javed, A. Belghith, and M. Khalgui, "Micro Air Vehicle Link (MAVLink) in a nutshell: A survey," *IEEE Access*, vol. 7, pp. 87 658–87 680, 2019.

[100] I. Smith, A. Voelim, D. Niemi, J. Tombs, B. Smith, R. Grehan, and T. Yager, "Unixbench: the original BYTE UNIX benchmark suite," Available: https://github.com/kdlucas/byte-unixbench.

[101] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization*, ser. WWC-4. Austin, TX, USA: IEEE, Dec. 2001, pp. 3–14.

[102] R. Kaiser and S. Wagner, "Evolution of the pikeos microkernel," in *First International Workshop on MicroKernels for Embedded Systems*, ser. MIKES '07, Jan. 2007.

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

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

[105] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[106] D. Sangorrin, S. Honda, and H. Takada, "Reliable and efficient dual-OS communications for real-time embedded virtualization," *Computer Software*, vol. 29, no. 4, pp. 182–198, 2012.

[107] S. Pinto, J. Pereira, T. Gomes, M. Ekpanyapong, and A. Tavares, "Towards a TrustZone-assisted hypervisor for real-time embedded systems," *IEEE Computer Architecture Letters*, vol. 16, no. 2, pp. 158–161, Oct. 2016.

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

[109] P. Lucas, K. Chappuis, M. Paolino, N. Dagieu, and D. Raho, "VOSYSmonitor, a low latency monitor layer for mixed-criticality systems on ARMv8-A," in *29th Euromicro Conference on Real-Time Systems*, ser. ECRTS '17, M. Bertogna, Ed., vol. 76, Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 6:1–6:18. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2017.6

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

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

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

[113] F. Bruns, S. Traboulsi, D. Szczesny, E. Gonzalez, Y. Xu, and A. Bilgic, "An evaluation of microkernel-based virtualization for embedded real-time systems," in *2010 22nd Euromicro Conference on Real-Time Systems*, ser. ECRTS '10. IEEE, 2010, pp. 57–65.

[114] G. Schwäricke, R. Tabish, R. Pellizzoni, R. Mancuso, A. Bastoni, A. Zuepke, and M. Caccamo, "A real-time virtio-based framework

for predictable inter-vm communication," in *2021 IEEE Real-Time Systems Symposium*, ser. RTSS '21. IEEE, 2021, pp. 27–40.
[115] SYSGO, "PikeOS — VirtIO," Available: https://www.sysgo.com/virtio.
[116] UNSW Trustworthy Systems Group, "SMACCM: TS in the DARPA HACMS Program," Available: https://trustworthy.systems/projects/OLD/SMACCM/.
[117] D. Cofer, J. Backes, A. Gacek, D. DaCosta, M. Whalen, I. Kuz, G. Klein, G. Heiser, L. Pike, A. Foltzer, M. Podhradsky, D. Stuart, J. Grahan, and B. Wilson, "Secure mathematically-assured composition of control models," Air Force Research Laboratory (RITA), Tech. Rep., Sep. 2017.
[118] Lookwerks, "HACMS: High Assurance Cyber Military Systems," Available: http://loonwerks.com/projects/hacms.html.
[119] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish *et al.*, "seL4: Formal verification of an OS kernel," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. ACM New York, NY, USA, Oct. 2009, pp. 207–220.
[120] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. 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*, ser. RTAS '13, Philadelphia, PA, USA, Jun. 2013, pp. 55–64.
[121] M. Bechtel and H. 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)*, 2019, pp. 357–367.
[122] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, "PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms," in *19th Real-Time and Embedded Technology and Applications Symposium*, ser. RTAS '14. IEEE, 2014, pp. 155–166.
[123] Q. Ge, Y. Yarom, T. Chothia, and G. Heiser, "Time protection: The missing os abstraction," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. Dresden, Germany: ACM New York, NY, USA, 2019.
[124] R. J. Masti, C. Marforio, A. Ranganathan, A. Francillon, and S. Capkun, "Enabling trusted scheduling in embedded systems," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC '12, Orlando, FL, USA, Dec. 2012, pp. 61–70.
[125] F. Alder, J. Van Bulck, F. Piessens, and J. T. Mühlberg, "Aion: Enabling open systems through strong availability guarantees for enclaves," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. Virtual Event, Repbulic of Korea: ACM New York, NY, USA, Nov. 2021, pp. 1357–1372.
[126] E. Aliaj, I. D. O. Nunes, and G. Tsudik, "GAROTA: Generalized active Root-Of-Trust architecture (for tiny embedded devices)," in *31st USENIX Security Symposium*, ser. USENIX Security '22. Boston, MA: USENIX Association, Aug. 2022, pp. 2243–2260. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/aliaj
[127] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl, "TyTAN: Tiny trust anchor for tiny devices," in *Proceedings of the 52nd ACM/EDAC/IEEE Design Automation Conference*, ser. DAC '15, San Francisco, CA, USA, Jun. 2015, pp. 1–6.
[128] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite: A security architecture for tiny embedded devices," in *Proceedings of the 9th European Conference on Computer Systems*, ser. EuroSys '14. Amsterdam, Netherlands: ACM New York, NY, USA, Apr. 2014.
[129] A. Thomas, S. Kaminsky, D. Lee, D. Song, and K. Asanovic, "ERTOS: Enclaves in real-time operating systems," in *Fifth Workshop on Computer Architecture Research with RISC-V*, ser. CARRV '21, Virtual, Jun. 2021.
[130] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. ACM New York, NY, USA, Apr. 2020.
[131] H. Yan, Z. Ling, H. Li, L. Luo, X. Shao, K. Dong, P. Jiang, M. Yang, J. Luo, and X. Fu, "LDR: Secure and efficient Linux driver runtime for embedded TEE systems," in *Network and Distributed Systems Security Symposium*, ser. NDSS '24. San Diego, CA: The Internet Society, Feb. 2024.
[132] H. Park and F. X. Lin, "GPUReplay: a 50-KB GPU stack for client ML," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. Lausanne, Switzerland: ACM New York, NY, USA, 2022, p. 157–170. [Online]. Available: https://doi.org/10.1145/3503222.3507754
[133] S. Han and J. Jang, "MyTEE: Own the trusted execution environment on embedded devices," in *Network and Distributed Systems Security Symposium*, ser. NDSS '23. San Diego, CA: The Internet Society, Feb. 2023.
[134] L. Guo and F. X. Lin, "Minimum viable device drivers for ARM TrustZone," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. Rennes, France: ACM New York, NY, USA, 2022, p. 300–316.
[135] A. Khan, H. Kim, B. Lee, D. Xu, A. Bianchi, and D. J. Tian, "M2MON: Building an MMIO-based security reference monitor for unmanned vehicles," in *30th USENIX Security Symposium*, ser. USENIX Security '21. USENIX Association, Aug. 2021, pp. 285–302. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/khan-arslan
[136] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vTZ: Virtualizing ARM TrustZone," in *26th USENIX Security Symposium*, ser. USENIX Security '17. Vancouver, BC: USENIX Association, Aug. 2017, pp. 541–556. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/hua
[137] J. Jang and B. B. Kang, "3rdParTEE: Securing third-party IoT services using the Trusted Execution Environment," *IEEE Internet of Things Journal*, vol. 9, no. 17, pp. 15 814–15 826, 2022.
[138] M. Kuhne, S. Sridhara, A. Bertschi, N. Dutly, S. Capkun, and S. Shinde, "Aster: Fixing the android TEE ecosystem with ARM CCA," 2024, arXiv preprint arXiv:2407.16694.
[139] S.-W. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui, "Formally verified memory protection for a commodity multiprocessor hypervisor," in *30th USENIX Security Symposium*, ser. USENIX Security '21, 2021, pp. 3953–3970.
[140] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Transactions on Software Engineering*, no. 2, pp. 125–143, 1977.
[141] P. P. Lee, T. Bu, and G. 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*, ser. ANCS '09, 2009, pp. 78–79.
[142] A. Barrington, S. Feldman, and D. Dechev, "A scalable multi-producer multi-consumer wait-free ring buffer," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, ser. SAC '15. Salamanca, Spain: ACM New York, NY, USA, Apr. 2015, pp. 1321–1328.
[143] Eclipse Foundation, "Iceoryx: Lock-free queue," Available: https://github.com/eclipse-iceoryx/iceoryx/blob/main/doc/design/lockfree_queue.md.
[144] D. Dechev, P. Pirkelbauer, and B. 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*. IEEE, 2010, pp. 185–192.

Appendix A. PARTEE's System Implementation

Memory protection: User-space virtual-memory access control is sufficient for isolating enclaves from each other

spatially; however, for ARMv8, after stage 1 and 2 translations, there are no further MMU mechanisms to prevent the Normal world from accessing Secure world memory (until ARMv9). ARM's optional bus-security mechanism, the TrustZone Address Space Controller (TZASC), can configure access to physical memory regions [72], but some SoCs, like the NVIDIA Tegra X2 (TX2), have custom bus security mechanisms.[2]

Integrated design of Firmware and Secure OS: 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. 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. We optimize the trap handling over observing that 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. Floating-point registers are swapped lazily only when enclaves use them. 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.

Appendix B. PARTEE's wait-free broadcasting ring buffer

The basic building block of communication in PARTEE are *shared-memory* fixed-size ring buffers queues. To the best of our knowledge, this exact data structure has not been discussed before, although some variations can be found in the literature [140]–[142] and in off-the-shelf products [143]. The primary innovation is how we use this data structure in a kernel-mediated way in §6; however, there are some unique aspects to its design as is. 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—triggering a graceful return.
- Publishing is transactional; no partial or corrupted messages can be written when following 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 malicious publisher can never infinitely block subscribers with ring corruption.

The wait-free broadcasting publish protocol: The ring buffer is made up of two arrays as shown in Fig. 15; at a high level, this division enables transactional publishing. The *Data Ring* array stores message content. It is a `ring_size` array of `slot_size` elements (or "slots"), each with a size 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 [144]. Moreover, we require that the Data Ring is larger than the number of publishers, ensuring that simultaneous contention will always be resolved. We go over the steps of the protocol, shown in Fig. 15.

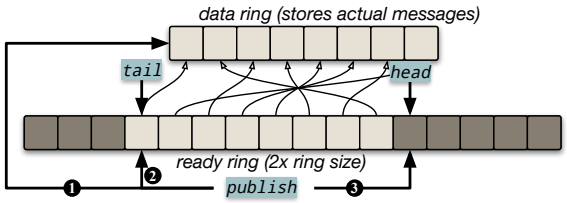

Figure 15. A diagram of PARTEE's Broadcasting Ring Buffer and an overview of the steps to publish; the dark cells of the buffer indicate that the pointer is `NULL`.

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. 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. Retries only occur if the ready ring wraps around entirely between the two atomics.

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. 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.

2. Since the Pi4B does not have memory bus security, we instrumented SeKVM [139] (with no guest) to ensure isolation.