PARTEE: Trustworthy Real-Time Containers for the Physical AI Era

Richard Habeeb, Sophia Yang, Zhong Shao

Abstract

Physical AI systems—ranging from autonomous vehicles and humanoid robots to medical devices and industrial automation—are rapidly transitioning from experimental prototypes to real-world deployments. This shift introduces acute security, reliability, and regulatory challenges: physical AI must remain trustworthy even under partial compromise, while meeting strict real-time constraints and supporting modern software stacks.

We present PARTEE, an on-device, firmware platform that makes trust a first-class system property for physical AI. PARTEE introduces lightweight, container-like partitions built on hardware-backed trusted execution environments (TEEs), enabling unmodified Linux applications to run securely with strong isolation, real-time availability guarantees, and hardware-rooted attestation. Unlike traditional TEEs or partitioning hypervisors, PARTEE is designed for the performance, usability, and middleware requirements of modern robotics and IoT systems. We describe PARTEE's architecture, partition model, and availability mechanisms, and demonstrate its effectiveness through real-world case studies, including autonomous drones and real-time medical data processing. We propose that trustworthy physical AI systems can be built without sacrificing development velocity through PARTEE, which provides a practical foundation for secure, reliable deployment at scale.

Contents

[1]

[1] Version 1.0: February 4th, 2026

1 Introduction

1.1 The Catalyst: Physical AI is Here.

Physical AI—autonomous vehicles, IoT devices, and *embodied AI* technology—is at an inflection point, moving rapidly into real-world deployment. Autonomous taxis are maturing, with Waymo clocking in just shy of half a million weekly rides.[2] Unitree's G1 humanoids are performing synchronized backflips at live concerts. Even highly regulated domains are crossing thresholds: the FDA is approving and exempting AI-enabled medical devices at an unprecedented pace.[3] As founders and developers, this technology is no longer the fun projects and moonshot ideas we used to imagine; it is here, now.

The same forces making physical AI possible are compressing the timeline to get security and reliability right. It is easier than ever to write code, and the pressure to ship is imminent, creating inevitable tension between getting to market and building trustworthy infrastructure. We have seen this play out at 39C3, where researchers exhibited multiple ways to remotely take over Unitree's G1,[4] via Internet and Bluetooth, affecting other robots in range.[5] These robots could not be updated to support basic defenses, which are reportedly planned for later versions.

1.2 The Challenge: Trust at Speed

We believe *trust* will be the primary asset driving and maintaining adoption of new physical AI technology. Trust—security, privacy, and reliability—is uniquely challenging to establish in physical AI systems, and getting it wrong can be devastating.[6] Unlike cloud systems, physical AI interacts directly with our real world, operating in homes, hospitals, and public spaces. Failures have immediate consequences.

Ever-expanding attack surface. Security teams cannot fully assess and validate the entire engineering stack for physical AI devices. On-device hardware—*graphics processing units (GPUs)* and *neural processing units (NPUs)*—are integrated into a multicore *system-on-a-chip (SoC)*, running large operating systems with millions of lines of third-party driver code enabled by default.[7] When combined with third-party libraries, runtimes, and frameworks, this sprawling internet-connected codebase presents an ever-expanding attack surface. Further, AI systems are routinely targeted by well-funded actors capable of performing insider[8] and supply chain attacks across software and hardware.[9]

Data-driven privacy risks. Modern physical AI systems depend on large-scale data collection to improve performance. As Richard Sutton's influential 2019 essay, "The Bitter Lesson," observes, scaling compute and data consistently outperforms clever algorithms [14]. While

[2] *Delivering more for our riders in a year of incredible growth [Link]*
[3] *FDA exempts more wearable, AI features from oversight [Link]*
[4] *Skynet Starter Kit, From Embodied AI Jailbreak to Remote Takeover of Humanoid Robots [Link]*
[5] *Exploit Allows for Takeover of Fleets of Unitree Robots—Security researchers find a wormable vulnerability [Link]*
[6] *Video: Humanoid robot fires at YouTuber after prompt twist, sparking AI safety fears [Link]*
[7] *Linux Kernel Surpasses 40 Million Lines of Code [Link]*
[8] *Former employees behind Tesla data breach [Link]*
[9] *Shai-Hulud 2.0: Guidance for detecting, investigating, and defending against the supply chain attack [Link]*

ubiquitous AI chat models consume and benefit from effectively endless, continuously refreshed data on the Internet, robotics systems have no such advantage. Developers must actively collect data from the physical world, making privacy an architectural challenge. Tesla, *e.g.*, has turned every driver into a source of training data and a beta tester for Autopilot, illustrating both the power and the risk of this approach.[10]

The same data required to make these systems work well is also a liability. Data breaches are costly: government fines routinely reach millions[11, 12] and lost revenue from canceled contracts cascades over time.[13] Protecting user data while continuing to collect it at scale is central to building trustworthy physical AI.

Real-time constraints. Physical AI is fundamentally constrained by its need for *real-time* responsiveness and *availability* to achieve trust. Robotics, autonomous vehicles, and medical devices must be so reliable that users forget they can fail. Unlike traditional IT and cloud security models, where a 50 ms delay goes unnoticed, an AV that lags for 50 ms for a security check is an AV that crashes.

Trust is expensive. If building trust is so essential to product adoption, why do teams retrofit security, privacy, and availability onto their architectures late in development? In practice, achieving this level of guarantees demands investment into scarce and expensive embedded systems expertise, rigorous and long testing cycles, and early commitments to hardware and software that conflict with rapid product iteration. As a result, developers frequently defer trust, and discover too late that retrofitting a good architecture is far harder than building it in from day one.

1.3 The Solution: PARTEE

How, then, can developers establish a trusted system foundation while sprinting at this new market pace? We propose **PARTEE**. PARTEE makes trust a first-class system property in physical AI. Enforced at the hardware boundary, resilient under partial compromise, and compatible with real-time operation, PARTEE resolves the tradeoff of shipping quickly and building trustworthy robotics and IoT products.

The main architectural tool for protecting critical software is *isolation*, *i.e.* running sensitive code in a hardware-protected environment that remains secure even if the rest of the system is compromised. This is what a *trusted execution environment (TEE)* provides. TEEs are standard in mobile devices for secure payments and biometrics, and are increasingly deployed in cloud computing. Yet, for robotics and IoT, TEEs remain underutilized because existing frameworks were not designed for the performance, availability, and real-time requirements of these workloads.

PARTEE addresses the traditional shortcomings for TEEs by using *container-like* isolation [6]. Rather than relying on air gaps or *virtual machines (VMs)*, PARTEE rethinks system

[10] *How Tesla Turned Every Driver Into a Data Source [Link]*

[11] *FTC Says Ring Employees Illegally Surveilled Customers, Failed to Stop Hackers from Taking Control of Users' Cameras [Link]*

[12] *FTC Takes Action Against Security Camera Firm Verkada over Charges it Failed to Secure Videos, Other Personal Data and Violated CAN-SPAM Act [Link]*

[13] *'Democracy in action': Cities react to privacy concerns by canceling Flock surveillance contracts [Link]*

partitioning by combining hardware-backed TEEs with lightweight containers. This class of system was largely infeasible until recent years. Today, a convergence of capable hardware (*e.g.*, ARM TrustZone on major SoCs), urgent market pressure, and repeated demonstrations of systemic vulnerabilities has created a window for architectural innovation.

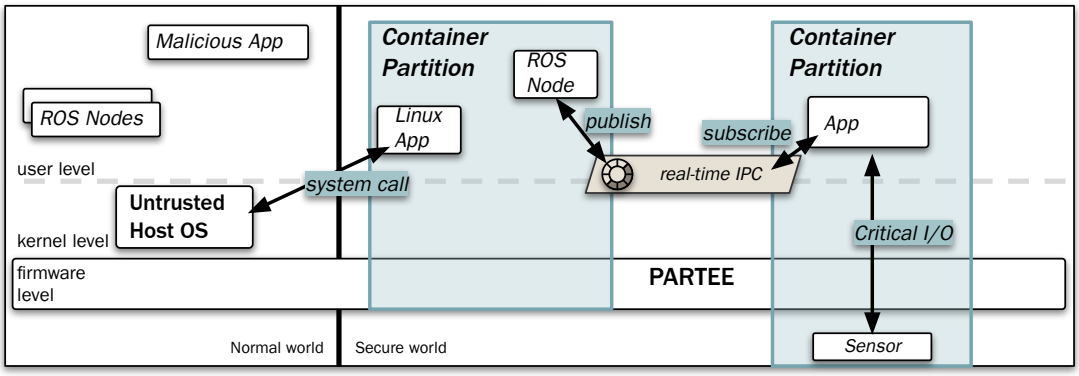

Figure 1: *Overview of PARTEE's system architecture: The most critical applications are executed in partitions; TEEs which act like resource containers, ensuring that even under host OS compromise, the critical software is architecturally isolated.*

What is PARTEE? PARTEE is an on-device, low-level firmware platform that allows developers to deploy *unmodified* critical software into transparent, hardware-backed TEEs, with support for the software architectures and real-time constraints of physical AI. Developers can run existing Linux applications without major changes, avoiding the costs and risks of retrofitting or redesigning their systems.

While most physical AI software sits at a high level, atop Linux-based stacks, the security and reliability gap demands security innovation at the lowest layers of the system. By integrating directly with robotics middleware, *e.g.* ROS, and enforcing isolation and availability at the firmware boundary—even under host OS compromise—PARTEE enables teams to ship faster without sacrificing security, availability, or regulatory credibility.

Who is PARTEE for? PARTEE is built for companies shipping physical AI systems who face steep security and safety requirements but cannot afford to slow down. Further, PARTEE's highly configurable design does not require an all-or-nothing security model: developers can selectively isolate the most critical components into TEE partitions, and layer PARTEE alongside existing hypervisors where appropriate. Our first version targets mid-size and large embedded ARM SoCs, *e.g.* NVIDIA IGX systems, in sectors including automotive, humanoid robotics, warehouse automation, medical devices, and aviation.

Key technical features of PARTEE

- Lightweight TEE containers without full virtualization overheads (§2, 3.2), around 0.5% to 4% overhead on the system overall, and even enabling 1.5x performance *improvements* on average for I/O bound TEE workloads

- Support for existing software: unmodified Linux applications, and planned legacy OP-TEE trusted app support (§3)
- High-speed integrations with middleware like *Robot Operating System (ROS)* (§3.3)
- Transparent, protected, and low-overhead file system and network access across partitions using asynchronous I/O (§3.1)
- Availability protections for time-sensitive or real-time software in the face of root-level *denial-of-service (DoS)* attacks (§2.2)
- A pathway towards formal verification: PARTEE is built with a future of formal verification in mind (§5.1)

The following sections describe PARTEE's technical architecture with its core abstraction: the partition.

2 PARTEE Partitions: Real-Time TEE Containers

PARTEE introduces container-like *partitions*: lightweight, hardware-backed TEEs designed to protect the most critical software in a physical AI system. For robots, this typically includes control loops responsible for motion, balance, and actuator safety; for medical or industrial systems, it may include real-time sensor pipelines or safety controllers. PARTEE partitions are designed to provide strong isolation, bounded resource usage, and real-time availability—even when the host operating system is compromised.

Physical AI systems span multiple trust domains: control loops, perception pipelines, middleware, and application logic often have different security, availability, and performance requirements. PARTEE reflects this reality: developers can selectively isolate the software components that require strong guarantees, leaving the rest of the system unchanged.

The implications of real-time on security: Unlike traditional TEE deployments, PARTEE partitions are explicitly designed to preserve availability under adversarial conditions. Physical systems impose strict *real-time* constraints: the most critical software is also the most time-sensitive. Responding to sensor input or handling low-latency network traffic must occur within predictable time bounds, making them vulnerable to denial-of-service (DoS) attacks from compromised OS components or faulty software. PARTEE enforces per-partition CPU, memory, and I/O budgets at the firmware level, ensuring that safety-critical software continues to execute within defined timing bounds regardless of host OS behavior.

2.1 Motivating Challenges for Partitioning

Where traditional TEEs fall short for Robotics and IoT: While TEEs provide strong primitives for confidentiality and integrity, existing TEE deployments fall short for robotics and IoT systems because they do not address availability, containment, and usability at the system level.

- **Lack of availability or real-time guarantees:** Existing TEE frameworks were not designed for workloads that must meet strict timing deadlines [2]. A secure enclave that

blocks on memory allocation, garbage collection, or host OS services cannot support safety-critical control loops.

- **Engineering complexity and operational costs:** TEEs require significant expertise to build correctly: developers must manage specialized toolchains, custom *software development kits (SDKs)*, and *public key infrastructure (PKI)* certificates and *attestation* chains. Traditional TEEs frameworks can be error prone [3], requiring more rigorous and involved testing and validation in order to avoid opening new security vulnerabilities.
- **Limited functionality:** Existing TEE environments are optimized for discrete, stateless operations (cryptographic signing, key storage). Robotics workloads need persistent state, sensor access, and inter-process communication—capabilities that traditional TEE models do not easily accommodate. Generally, physical AI depends on the file system, network, and hardware accelerator as well, and such access can be challenging from TEEs.

Traditional Partitioning Hypervisors: If existing TEE frameworks fall short for physical AI workloads, why not rely on traditional partitioning approaches instead? Partitioning hypervisors have long been used in mixed-criticality systems to isolate safety-critical software from less trusted components.[14] In automotive and avionics systems, *e.g.*, they are used to partition a single system-on-chip into multiple *VMs* with statically assigned resources.

These *partitioning hypervisors*—originally called separation kernels [13]—provide strong assurances that each control system will have access to resources such as memory and CPU time, while remaining isolated from other software components. While effective in narrowly scoped, static safety-critical domains, this model is poorly suited to modern physical AI systems due to its:

- **Inability to run modern robotics applications.** Today's robotics systems run on Ubuntu, use Python, and depend on network APIs and filesystem access. Partitioning hypervisors require control software to be custom-built for isolated VMs running minimal *real-time operating systems (RTOSs)*, which lack access to the rich tooling developers expect. As a result, debugging and observability are significantly impaired, and building many common applications becomes infeasible due to missing OS features.
- **High engineering and operational overhead.** VMs typically require control software to run in uncommon *RTOSs*, in addition to inter-VM communication infrastructure. This increases system complexity, raises the cost of development and maintenance, and complicates debugging, deployment, and long-term evolution of the software stack.
- **Performance and flexibility limitations.** Hardware virtualization introduces unavoidable overhead from the hypervisor and virtualized I/O, which can be problematic for latency-sensitive workloads. Further, hypervisors typically enforce static resource partitions defined at the early design stage. While appropriate for some domains with fixed functionality, *e.g.*, avionics, this rigidity is poorly matched to the dynamic workload of robotics and IoT systems.

[14] Prior to hypervisors, isolation, *e.g.* on older cars, was achieved by physically separating control systems ("air-gapping") from other less-critical software onto dedicated chips.

Taken together, these limitations suggest that the core problem is not isolation itself, but the abstraction at which it is applied. Physical AI systems require isolation mechanisms that preserve access to modern Linux-based software stacks, impose minimal overhead on latency-sensitive workloads, and allow developers to selectively isolate the components that demand strong guarantees. Static, VM-based partitioning fails to meet these requirements.

Our insight: In the physical AI era, *container-level* partitioning within hardware-backed TEEs provides strong isolation and availability guarantees without sacrificing usability, flexibility, or development velocity.

PARTEE partitions at the *container* level, allowing teams to build trust as a lightweight, default property of the system.

2.2 What do PARTEE partitions do?

PARTEE partitions conceptually extend Linux Control Groups (`cgroups`), which create LXC or Docker containers, into hardware-backed TEEs. Like `cgroups`, PARTEE's partitioning enforces:

- **Resource limiting and accounting:** Each partition is given a maximum memory quota, CPU budget, and I/O bandwidth, which is tracked and enforced by PARTEE.
- **Prioritization and scheduling:** Partitions are scheduled according to defined CPU and I/O priorities, enabling predictable responsiveness.

Additionally, PARTEE enforces and enables:

- **Fine-grained access control:** Devices and shared *inter-process communication (IPC)* channels can be assigned to partitions, preventing untrusted software from interfering with sensor and actuator access.
- **Authentication and attestation:** Each partition can support a unique authentication key, enabling isolation across different privilege levels, vendors, or stakeholders.
- **Confidential and available IPC:** Access control to ROS-compatible communication topics is enforced at the partition boundary to prevent DoS attacks (see §3.3).

When combined with Linux namespaces, PARTEE's partitions enable lightweight confidential containers that can also prevent software-based DoS attacks. At a high level, PARTEE uses the TrustZone primitives to isolate CPU time and memory for each `cgroup`-like partition. To achieve real-time availability, PARTEE uses a secure timer to preempt and schedule Linux on each core, ensuring each partition gets a dedicated budget of time per second. For technical details on how PARTEE implements partitioning, see our research paper [6]. We discuss how I/O and system calls are handled without breaking availability in §3.1.

2.2.1 How do you use partitions?

PARTEE is designed to make the development and usage of TEEs as easy as possible, so we follow traditional Unix-like patterns where possible. TEE applications can be run in a partition using a few ways:

- **Automatically, at boot time:** Application code is bundled with the PARTEE engine (which runs as firmware before anything else on the system), and started during boot in a target partition with preconfigured settings.
- **Dynamically, using a shell command:** Applications that can be launched via the command line can be launched in a target partition by simply prefixing the command with a proxy command, `partee`, similar to the Docker CLI's `docker`. The application needs to be bundled with a configuration manifest and cryptographically signed for authentication—more on this in §2.3. The `partee` command forwards the application bundle, command-line parameters, and environment variables to the PARTEE engine firmware.
- **Remotely, via an API:** PARTEE provides a service to remotely launch and perform *attestation*—more in §2.3.

2.2.2 How do you manage partitions?

PARTEE provides two ways for partitions to be created, depending on system requirements. For traditional avionics and automotive, dynamic memory allocation is not allowed after initialization—this is standard across numerous safety-critical industries. In this case, developers must predefine partitions using a boot-time configuration file; PARTEE's tool chain signs and bundles the configuration with the PARTEE firmware, and at boot time the firmware creates the partitions and locks down the system, preventing further changes. For more flexible systems that support easy testing and debugging, developers can elect to dynamically create partitions after initialization. If allowed, at runtime applications with appropriate access can dynamically split partitions through a local programming interface (a system call), donating some subset of resources to the new "child" partition.

2.3 Partitioned Root-of-Trust and Attestation

How can engineers be confident that a deployed system is working with TEEs correctly? One can imagine that a hacker could deploy an application without TEE security rather than using PARTEE; the system could appear normal, but would be under adversary control. The key defense against this is via a hardware-backed *root-of-trust* and *attestation*, specifically *Attested TLS*. The specific implementation depends on hardware capabilities; some SoCs provide on-chip secure storage which can only be accessed by the bootloader and firmware, while others require external *trusted platform modules (TPMs)*. At a high level the root-of-trust allows for secure boot—starting the system in a secure, known state with authenticated firmware and OS images. Additionally, the root-of-trust allows for attestation, which proves to the robot or device developers that the software is executing in a TEE.

Partitioned authentication and attestation with PARTEE: One advantage is that PARTEE can support multiple stakeholders with different privilege levels, creating a least-privilege approach to security. Once a hardware root-of-trust is configured, PARTEE can authenticate and attest applications to each partition individually. The hardware root-of-trust is used to authenticate PARTEE's firmware and data, and then PARTEE authenticates TEE applications on a per-partition basis using each partition's public key. *I.e.*, when developers create a partition, they also create an X.509 certificate and provide it for that partition. When remotely configuring a robot, PARTEE uses the Attested TLS protocol to prove that the TEE is constructed and launched correctly. The attestation includes signed measurements of memory and initial application load parameters using the ephemeral attestation keys created at boot-time.

Because attestation is performed at the partition level, PARTEE enables multi-stakeholder systems where different vendors, suppliers, or software components are independently authenticated and isolated—without requiring a single monolithic trust domain.

3 High-Performance Unmodified Applications

One of the primary barriers to adopting TEEs in physical AI systems has been the lack of a usable programming environment compatible with modern software stacks. PARTEE addresses this by enabling many different kinds of unmodified applications to execute inside TEE partitions, opposed to limited trustlets, while still allowing developers to build custom applications that directly leverage PARTEE's real-time and isolation guarantees.

3.1 How to make system calls from a TEE

TEEs have a few technical distinctions from a typical application runtime environment. Most importantly, they do not usually have a way to make *system calls*, *i.e.* the API functions of the Normal-world host OS. In the case of ARM TrustZone approaches, the TEE application's system call interface is actually the trusted OS running in the Secure world, which has very limited functionality. For instance, Linux programs commonly use system calls like `read()` or `write()` to interact with files, sockets, and pipes, but these are not provided by OP-TEE [9] and other trusted OS implementations. The other distinctions are related to what role the Secure-world trusted OS plays versus the Normal-world host OS, particularly with respect to memory management and scheduling.

Leveraging asynchrony: PARTEE provides an asynchronous I/O interface to TEE applications to enable them to request system calls from the host OS. Because the host OS might be adversarial, using an asynchronous approach means that the application will not be "blocked" if the OS maliciously never returns. PARTEE builds upon Linux's `io_uring` features to provide this asynchronous interface, but it abstracts away the risky security details (like sharing memory between the TEE and Linux). The result is that applications can read and write to files, pipes, or the network using an `io_uring`-like interface. To ensure end-to-end I/O integrity and confidentiality, standard encryption techniques, like *TLS*, are available to prevent the host OS from corrupting or reading TEE data sent asynchronously. We found several deep performance optimizations are possible when using asynchronous I/O, allowing performance improvements for I/O heavy workloads; full details of this approach are in our paper [7].

3.2 How to use unmodified applications and TEE containerization

PARTEE implements a transparent library OS layer between the application and the Normal world OS (similar to Intel SGX Enclaves, PARTEE is able to leverage Gramine for this [15]). The library OS layer effectively sorts system calls and I/O into two directions: to be handled by PARTEE directly or to be handled by the Normal world OS. From a programmer's perspective the routing is handled automatically, and PARTEE can either signal or return a timeout if the call runs out of time.

3.3 Using publish/subscribe architectures

For many robotics architectures, communication between distributed software components can be critically important for real-time responsiveness. PARTEE provides an internal TEE-to-TEE cross-partition *inter-process communication (IPC)* mechanism using mediated shared memory. This mechanism guarantees that messages can be reliably and securely published and received between partitions, even under adversarial settings. PARTEE is designed to allow bridging and synchronizing between *Robot Operating System (ROS)* topics, to PARTEE topics. The result is that PARTEE is a deeply ROS-compatible TEE platform.

4 Examples and Case Studies

4.1 Quadcopter validation under OS compromise

We validated PARTEE on a quadcopter drone built as part of DARPA's VSPELLS program, using a software architecture representative of modern embodied AI systems [6, 7]. The drone performs autonomous tasks, *e.g.*, scanning terrain, identifying objectives, and coordinating with a remote ground station. We partitioned the system into two isolated TEE containers: (1) a real-time flight planner and sensor processing pipeline, and (2) a ground-control interface service handling waypoints and task logic. The host Linux OS runs normally in the Normal World, handling networking, logging, and non-critical services.

Under simulated attack, with an adversary holding root access to Linux, the drone's flight controller and planner remained isolated and operational. Rather than losing control and crashing, the drone detected the anomaly and autonomously returned to its home position. This architecture translates directly to humanoid robotics, where balance controllers and joint actuators require the same guarantees: isolation from compromise, guaranteed execution, and graceful degradation under attack.

An unexpected benefit: hardware consolidation Many drones and robots use separate hardware for real-time control (an "autopilot" microcontroller) and high-level computation (a "companion computer" running Linux). This separation exists partly for isolation—keeping the flight controller safe from the complexity of a full OS. PARTEE can eliminate this need. Time-sensitive control code can run on the same SoC as the Linux environment, isolated by TEE partitions rather than physical separation. The result: reduced weight, lower power consumption, simplified integration, and freedom from autopilot hardware lock-in.

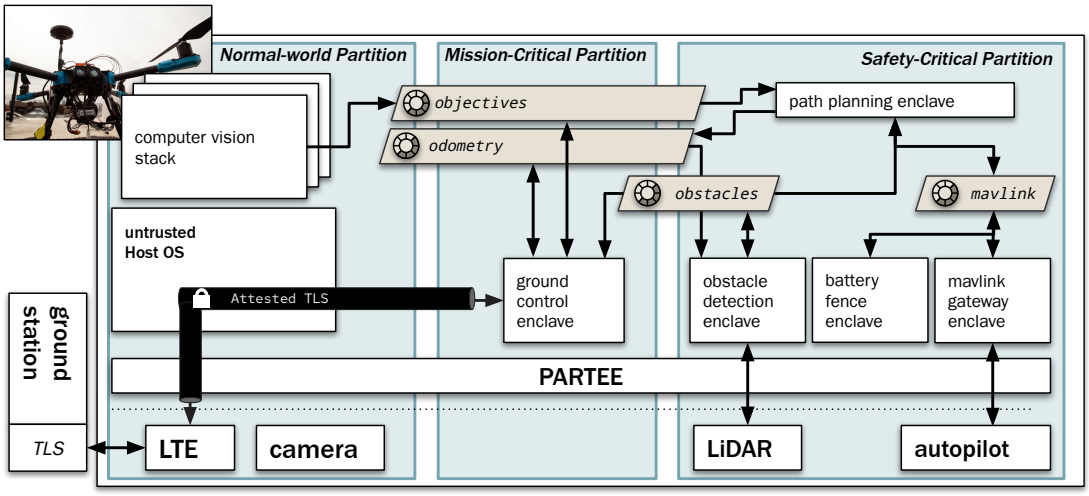

Figure 2: *Overview of our quadcopter demo software architecture, demonstrating one way the system can be partitioned with PARTEE to achieve strong security and reliability.*

4.2 PARTEE as NVIDIA Holoscan's real-time security layer

Consider a robotic surgery system where a momentary delay in video processing could mean the difference between a successful procedure and a critical error. NVIDIA's Holoscan platform targets such edge-deployed systems with real-time sensor and data-processing needs, from smart medical devices to surgical robotics.

The Holoscan C++ SDK provides a programming framework where applications are constructed from interconnected *fragments* of *operators*. The fragments can either be run as a single Ubuntu Linux process, or as distributed applications, with the Holoscan SDK automatically handling middleware transport. This architecture maps naturally onto PARTEE's partition model: developers can run fragments inside containerized TEE partitions, connected via PARTEE's real-time shared memory *IPC*.

The security and availability benefits of this are significant. Under normal operation, fragments execute with hardware-enforced isolation, protecting sensitive video streams and patient data from OS-level compromise. Under attack—even with a root-level adversary in the Normal World—PARTEE's secure timer guarantees that critical fragments receive their allocated CPU budget. A video decoding pipeline configured for 10ms frame delivery will meet that deadline regardless of what else is happening on the system.

This integration pattern extends beyond Holoscan. NVIDIA Isaac ROS and traditional ROS deployments can adopt the same model: isolating safety-critical nodes in TEE partitions while maintaining publish/subscribe communication with the broader system.

5 Competitive Analysis

PARTEE is distinguished by its combination of hardware-backed isolation *and* availability—with the usability that alternatives do not provide. Traditional TEE frameworks, like OP-TEE, provide strong isolation for small cryptographic services but lack the programming environment and real-time guarantees that robotics workloads require. Partitioning hypervisors like

Bao [11], PikeOS [8], and VxWorks Helix [16] offer availability guarantees but impose significant constraints: specialized RTOSes,

Capability	PARTEE	OP-TEE	Bao, PikeOS, VxWorks	No Isolation
Protection under root compromise	✓	✓	✓	
Unmodified Linux apps	✓			✓
Real-time availability guarantees	✓		✓	
No specialized RTOS required	✓			✓
ROS / middleware integration	✓			✓
Dynamic partition creation	✓			N/A
Hardware-rooted attestation	✓	✓	Partial	
Open source available	Planned	✓	Partial	N/A
Downstream engineering costs	Low	Medium	High	Low

Table 1: *Comparison of isolation approaches for physical AI systems.*

5.1 Certified Trust via Formal Verification

PARTEE's architecture is designed with future formal verification in mind. The partition scheduler and resource accounting logic are implemented in a minimal, layer-based design. We are working toward machine-checked proofs of key isolation properties, building on techniques from our research on CertiKOS and *Certified Concurrent Abstraction Layers (CCALs)* [4, 5, 10]. This positions PARTEE for safety-critical certifications (DO-178C, ISO 26262) that increasingly require evidence beyond testing.

6 Conclusion: Towards a Trustworthy Software Ecosystem

Trust will determine adoption of physical AI systems as they move from research and development environments into safety-critical, regulated, and human-facing deployments. Unlike cloud and traditional IT systems, failures in physical AI are immediate, visible, and often irreversible—making trust not a feature, but a prerequisite.

We argued that existing security approaches fall short for physical AI because they fail to deliver three properties simultaneously: strong isolation under compromise, real-time availability, and compatibility with modern software stacks. PARTEE addresses this gap by rethinking trusted execution environments as lightweight, container-like partitions enforced at the firmware boundary.

By enabling unmodified Linux applications, integrating directly with robotics middleware, and providing hardware-backed availability guarantees, PARTEE makes trust a first-class system property rather than an afterthought. As physical AI systems proliferate, architectures that retrofit trust will fail; it must be built into the foundation of the software stack for enduring success. PARTEE represents a step toward a more trustworthy software ecosystem—one where security, reliability, and performance are no longer in tension, but rather reinforced by architecture.

Acronyms

CCA ARM Confidential Compute Architecture 22
CCAL Certified Concurrent Abstraction Layer 13

DoS denial-of-service 6

GPU graphics processing unit 3

IPC inter-process communication 8, 11, 12

NPU neural processing unit 3

PKI public key infrastructure 7

ROS Robot Operating System 6, 11
RTOS real-time operating system 7

SDK software development kit 7
SoC system-on-a-chip 3

TA trusted application 22, *Glossary:* Trusted Application (Trustlet)
TEE trusted execution environment 4, *Glossary:* Trusted Execution Environment
TLS Transport Layer Security 10, 14
TPM trusted platform module 9

VM virtual machine 4, 7

Glossary

attestation A security process where a system or component provides cryptographic proof of its identity, configuration, or integrity to verify it hasn't been tampered with or compromised. This is commonly used in trusted computing to establish a chain of trust. 7, 9
Attested TLS A *Transport Layer Security (TLS)* connection where one or both endpoints provide cryptographic proof (attestation) that they are running specific, unmodified software inside a TEE. This allows a remote party to verify the integrity of the system before exchanging sensitive data. 9
availability The probability that a system is operational and ready to perform its function when needed, often expressed as a percentage of uptime. In real-time systems, this specifically concerns whether the system can reliably meet its timing constraints when required. 4

embodied AI AI that learns and operates through a physical or simulated body, emphasizing how intelligence emerges from sensorimotor interaction with an environment. This approach suggests cognition is fundamentally shaped by having a body that perceives and acts, rather than being purely computational. 3

partitioning hypervisor A type of hypervisor that provides strong spatial and temporal isolation between virtual machines, ensuring that faults or resource usage in one partition

cannot affect others. This is critical in safety-critical and real-time systems where different criticality levels must be kept separate. 7

partitions An isolated execution environment with dedicated resource budgets (CPU time, memory, I/O bandwidth) enforced by hardware or a minimal hypervisor [13]. Partitions provide strong guarantees against resource interference, making them suitable for safety-critical and mixed-criticality systems. 6

physical AI Inference networks that interact with and manipulate the physical world through robotics, sensors, and actuators, such as autonomous vehicles, robotic arms, or drones. 3

real-time A system characteristic where correctness depends not only on logical results but also on the time at which results are produced—responses must occur within strict, predictable time bounds. Real-time systems are classified as hard real-time (missing a deadline causes system failure) or soft real-time (missing deadlines degrades performance but doesn't cause failure). 4, 6

root-of-trust A foundational hardware or firmware component that is assumed to be secure and from which all other trust in the system is derived. Typically implemented in immutable boot ROM or a hardware security module, it anchors the chain of trust for secure boot, key storage, and attestation. 9

system call A request from a user-space application to the operating system kernel for privileged operations such as file I/O, network access, or memory allocation. System calls form the primary interface between applications and the OS. 10

Trusted Application (Trustlet) A small, security-sensitive program that runs inside a TEE, isolated from the main operating system. Trusted applications typically perform discrete operations like cryptographic signing, key management, or secure credential storage, and expose a limited interface to untrusted code. 14, 22

Trusted Execution Environment A secure "enclave" that isolates sensitive code and data processing from the rest of the system—it protects the confidentiality and integrity within the environment even if the OS is compromised [1]. Modern TEEs are created using CPU hardware features, such as Intel SGX, Intel TDX, AMD SEV-SMP, ARM CCA, or the ARM TrustZone. TEEs are the building block for cloud confidential computing, allowing "zero-trust" data processing on remote, shared machines. On mobile devices, TEEs typically perform cryptographic key management, biometrics, secure payments, and digital rights management. 4, 14

References

[1] A technical analysis of confidential computing. Technical report, The Confidential Computing Consortium, 2022.

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

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

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

[5] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. Certified concurrent abstraction layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '18, pages 646–661, Philadelphia, PA, June 2018. ACM New York, NY, USA.

[6] Richard Habeeb, Hao Chen, Man-Ki Yoon, and Zhong Shao. It's a non-stop PARTEE! practical multi-enclave availability through partitioning and asynchrony. In *Proceedings of the 41st Annual Computer Security Applications Conference*, ACSAC '25, Honolulu, HI, 2025.

[7] Richard Habeeb, Man-Ki Yoon, Hao Chen, and Zhong Shao. Ringmaster: How to juggle high-throughput host OS system calls from TrustZone TEEs. arXiv preprint 2601.16448.

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

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

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

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

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

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

[14] Richard Sutton. The bitter lesson, 2019.

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

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

A Appendix: Threat Modeling

PARTEE assumes a powerful software adversary who has gained root-level access to the Normal World OS. This attacker can read and write arbitrary Normal World memory, intercept or delay system calls, and attempt DoS attacks against TEE applications. The following tables categorize relevant threats, PARTEE's mitigations, and complementary defenses that provide defense-in-depth.

A.1 System-Level Threats

Threat	Details	PARTEE's Mitigation	Complementary Defenses
OS/kernel compromise	Attacker gains root access to the Normal World Linux kernel via exploit, misconfiguration, or malware.	TEE partitions remain isolated; attacker cannot read or modify TEE memory, code, or execution.	Kernel hardening, minimal kernel configs, automated patching, runtime integrity monitoring.
Privilege escalation	Attacker escalates from unprivileged Normal World process to "root" or attempts to escalate within TEE.	Cross-world escalation blocked by hardware. TEE privilege escalation mitigated by partition isolation.	Mandatory access control (SELinux, AppArmor), capability dropping, `seccomp` filters
Remote teleoperation exploit	Attacker (likely an insider) compromises remote control channel to send unauthorized commands to robot.	Partitioned containers protect most critical applications, limiting damage of insider threats.	Certificate pinning, Attested TLS, and anomaly detection
Software supply-chain attack	Malicious code is introduced via compromised dependencies, packages, or SDKs.	Attestation verifies TEE application and library integrity. Compromised Normal World dependencies cannot affect TEE code. Cannot prevent supply-chain attacks on TEE applications themselves.	SBOM tracking, dependency pinning, reproducible builds, signed packages, vendor security audits.
Malicious image update	Attacker pushes compromised firmware/OS image to device, or tries to run a previous vulnerable firmware or OS image bypassing security.	Secure boot chain verifies firmware signatures. Rollbacks are disallowed preventing downgrade attacks.	Update server authentication, staged rollouts, integrity monitoring post-update.

Table 2: *System-Level Threats*

A.2 AI and ML-specific Threats

Modern physical AI systems face novel attack vectors targeting their AI/ML components. Many of these attacks exploit fundamental properties of machine learning systems and cannot be fully prevented by any architectural defense—just as humans remain susceptible to deception, propaganda, and optical illusions. Thus, what PARTEE provides for AI-specific threats is *containment, not prevention*: ensuring that a compromised or manipulated AI component cannot escalate to full system compromise, cannot access resources beyond its designated partition, and cannot override hardware-enforced safety limits. This defense-in-depth

posture acknowledges that AI components may fail or be deceived, and designs the system so that such failures remain bounded.

Threat	**Details**	**PARTEE's Mitigation**	**Complementary Defenses**
Prompt injection	Adversary-controlled text (signs, documents, network data) is processed by an LLM, causing unintended commands or behavior changes.	PARTEE can isolate LLM processing from safety-critical control loops, limiting blast radius.	Input sanitization, prompt hardening, output filtering.
Agentic confused deputy	An AI agent with legitimate permissions is manipulated into performing actions on behalf of an attacker, *e.g.*, accessing protected files or actuators.	Partition-level access control limits what resources a compromised agent can reach. Least-privilege design reduces attack surface.	Capability-based permissions, action logging, human-in-the-loop for irreversible actions
Embodied AI jailbreak	Attacker bypasses safety constraints on an embodied AI agent via adversarial prompts, enabling dangerous physical actions.	Safety-critical motion limits can be enforced in a separate, isolated partition that the AI agent cannot modify.	TEE isolation of a: collision avoidance system, safety controller, anomaly detection, or AI kill-switch
Sensor spoofing / adversarial inputs	Adversary manipulates sensor data (camera, LIDAR, GPS) to cause incorrect perception or decisions, *e.g.*, adversarial patches on stop signs.	Sensor data integrity protected after capture; partitioning prevents compromised perception from directly controlling actuators.	Sensor fusion, redundancy, out-of-band validation, anomaly detection
Model poisoning / backdoors	Attacker compromises training data or model weights to embed backdoors that activate under specific conditions.	Attestation can verify model integrity at load time and partition impacts at runtime. PARTEE cannot detect backdoors embedded before deployment.	Training data provenance, model auditing, runtime anomaly detection.
Model extraction / theft	Attacker extracts proprietary model weights or architecture from a deployed system.	TEE isolation protects model weights in memory. Encrypted storage protects weights at rest.	Additional obfuscation, watermarking for detection, query pattern detection

Table 3: *AI and ML-specific Threats. PARTEE provides containment and integrity guarantees but cannot prevent all AI-level attacks alone.*

A.3 Threats Specific to TEE Applications

Threat	Details	PARTEE's Mitigation	Complementary Defenses
TEE software exploitation	Vulnerability in TEE application code allows attacker to corrupt TEE state or escape isolation.	Partition isolation contains damage to a single partition, in addition to normal application boundaries.	Memory-safe languages (Rust), fuzzing, code audits, minimal TEE application surface.
Malicious TEE loading	Attacker attempts to load unauthorized or modified TEE application.	Cryptographic authentication required for TEE loading. Only applications signed with partition keys are accepted.	Multi-party signing, key management policies, hardware security modules for signing keys
Attestation spoofing	Attacker forges attestation reports to convince remote verifier that compromised system is trustworthy.	Attestation rooted in hardware keys that cannot be extracted. Reports signed with ephemeral keys derived from root-of-trust.	Physically unclonable functions, certificate transparency logs
Private data exfiltration	Attacker extracts user secrets or sensitive sensor data (camera, microphone, LIDAR) from compromised system.	Sensors can be assigned to TEE partitions; data never exposed to Normal World.	Data minimization, on-device processing, encrypted export with user consent
Controlled-channel attacks	Attacker uses page fault patterns, cache timing, or interrupt timing to infer TEE secrets.	Page tables exclusively managed by PARTEE, preventing page-fault side channels. Interrupt handling minimizes timing leakage.	Constant-time implementations, cache partitioning, ORAM for sensitive access patterns.
Iago attacks (malicious syscall returns)	Malicious OS returns corrupted, fabricated, or malformed data in response to TEE system calls to exploit assumptions in TEE code.	PARTEE's library OS layer validates and sanitizes all values returned from the Normal World before use by TEE applications.	Defensive programming, bounds checking, treating all OS returns as untrusted input.

Table 4: Threats to TEE Applications

A.4 Availability and Real-Time Threats

Threat	Details	PARTEE's Mitigation	Complementary Defenses
Resource DoS (CPU)	Malicious Normal World code starves TEE of CPU cycles by refusing to yield or flooding interrupts.	Secure timer guarantees TEE partitions receive allocated CPU budget regardless of Normal World behavior.	Watchdog timers, anomaly detection on CPU usage patterns.
Resource DoS (memory)	Attacker exhausts memory to prevent TEE allocation or operation.	Partition memory quotas enforced at boot or creation time. TEE memory is hardware-isolated from Normal World.	Memory usage monitoring, graceful degradation under pressure.
Resource DoS (I/O)	Attacker saturates I/O channels to delay or block TEE data access.	Critical device I/O handled directly by PARTEE (*e.g.*, essential serial or low-level busses), and bandwidth is divided by partitions with access. For Linux controlled I/O, asynchronous system calls prevent blocking.	Traffic shaping, QoS policies, user-space drivers
System call blocking	Malicious OS delays or never returns from TEE-initiated system calls, blocking TEE execution.	Asynchronous I/O architecture ensures TEE never blocks on OS. Timeouts allow TEE to proceed or fail gracefully.	Redundant data sources, cached fallbacks, safe-state defaults.
IPC flooding	Attacker floods inter-partition or ROS communication channels to block legitimate messages.	PARTEE enforces per-partition rate limits and access control on IPC topics. Critical topics isolated from untrusted partitions.	Message prioritization, separate channels for safety-critical traffic.

Table 5: Availability and Real-Time Threats

A.5 Out-of-Scope Threats

The following threats are explicitly outside PARTEE's threat model. Addressing these requires complementary technologies and operational controls.

Threat	Details	Complementary Defenses
Physical attacks	Attacker with physical access performs JTAG debugging, chip decapping, cold boot attacks, glitching, or fault injection.	Tamper-evident/resistant enclosures, disabled debug interfaces, secure element chips, environmental sensors.
Microarchitectural side channels	Attacker exploits CPU cache timing, speculative execution (*e.g.*, Spectre or Meltdown), or power analysis to extract TEE secrets.	Constant-time implementations, cache partitioning, speculative execution mitigations, power analysis resistant hardware.
Hardware supply-chain attacks	Malicious hardware (implants, counterfeit chips, compromised firmware blobs) provides backdoor access.	Hardware provenance verification, trusted foundries, component authentication, runtime integrity checks.
RF/network denial of service	Attacker jams wireless communications or floods network to prevent remote operation or updates.	RF shielding, frequency hopping, wired fallbacks, local autonomous operation capability.
Malicious insider (authorized developer)	Developer with legitimate signing keys intentionally deploys malicious TEE application.	Multi-party signing requirements, code review mandates, audit logs, separation of duties, key escrow.

Table 6: Out-of-Scope Threats. PARTEE does not defend against these; complementary measures are required.

B ARM TrustZone and CCA Hardware

PARTEE builds on ARM's hardware isolation primitives, which are available on nearly all ARM processors used in robotics, IoT, and embedded systems.

ARM TrustZone. TrustZone, introduced in ARMv6 (2004), partitions a processor into two *worlds*: a *Normal world* running the main OS and applications, and a *Secure world* for trusted code and data [12]. The hardware enforces this separation, so the Normal World software cannot access Secure World memory, peripherals, or registers, even with kernel or hypervisor privileges. Unlike a hypervisor, it provides exactly two security domains with a fixed trust hierarchy: the Secure World can observe and control the Normal World, but not vice versa. This asymmetry makes TrustZone lightweight: world switches can take dozens or hundreds of cycles, instead of thousands compared to VM switching. Traditional TrustZone deployments run a small trusted OS (such as OP-TEE) in the Secure World, hosting isolated *trusted applications (TAs)*, or *trustlets* that perform security-sensitive operations like key storage, cryptographic signing, or DRM.

ARM Confidential Compute Architecture (CCA). ARMv9 (2021) introduced CCA as a successor to TrustZone's two-world model. CCA adds a new isolation primitive: *Realms*, which are hardware-isolated execution environments that are protected from both the Normal World OS *and* the Secure World. This enables confidential computing scenarios where even privileged system software cannot access Realm memory. CCA also introduces the *Granule Protection Table*, which provides fine-grained, per-page memory access control across four security states (Normal, Secure, Realm, and Root). For robotics and IoT, CCA offers a path toward stronger multi-stakeholder isolation—for example, separating OEM code from third-party vendor code, each in its own Realm.

PARTEE's Hardware Approach. PARTEE's initial implementation targets ARMv8 processors with TrustZone extensions, which represent the vast majority of deployed robotics and IoT hardware. PARTEE's partition abstraction is designed to map naturally onto CCA Realms as ARMv9 hardware becomes available, providing a migration path for customers who need stronger isolation guarantees in the future.