RT-TEE: Real-time System Availability for Cyber-physical Systems using ARM TrustZone

Jinwen Wang, Ao Li, Haoran Li, Chenyang Lu, Ning Zhang
Washington University in St. Louis, MO, USA

***Abstract*—Embedded devices are becoming increasingly pervasive in safety-critical systems of the emerging cyber-physical world. While trusted execution environments (TEEs), such as ARM TrustZone, have been widely deployed in mobile platforms, little attention has been given to deployment on real-time cyber-physical systems, which present a different set of challenges compared to mobile applications. For safety-critical cyber-physical systems, such as autonomous drones or automobiles, the current TEE deployment paradigm, which focuses only on confidentiality and integrity, is insufficient. Computation in these systems also needs to be completed in a timely manner (e.g., before the car hits a pedestrian), putting a much stronger emphasis on availability.**

To bridge this gap, we present RT-TEE, a real-time trusted execution environment. There are three key research challenges. First, RT-TEE bootstraps the ability to ensure availability using a minimal set of hardware primitives on commodity embedded platforms. Second, to balance real-time performance and scheduler complexity, we designed a policy-based event-driven hierarchical scheduler. Third, to mitigate the risks of having device drivers in the secure environment, we designed an I/O reference monitor that leverages software sandboxing and driver debloating to provide fine-grained access control on peripherals while minimizing the trusted computing base (TCB).

We implemented prototypes on both ARMv8-A and ARMv8-M platforms. The system is tested on both synthetic tasks and real-life CPS applications. We evaluated rover and plane in simulation and quadcopter both in simulation and with a real drone.

I. Introduction

The software of modern cyber-physical systems (CPSs) is often highly complex. For example, the code in a modern automobile such as the Chevy Volt is as complex as the total flight software of the Boeing 787 airplane [1]. Under the pressure to include more features and to save on production cost, weight, and testing, CPS system designers are consolidating more and more functionalities on a single system-on-chip (SoC) [2], [3]. Numerous software vulnerabilities have been discovered on modern cyber-physical systems such as drones [4], [5] and automobiles [6]. While some of these vulnerabilities are only nuisances [7], others allow attackers to escalate into system privilege [7], [8], [4], [9] and can have life or death implications [6].

Lack of availability protection in existing defenses: Recognizing the importance of embedded system security, there has been significant interest in hardening the software using security mechanisms, such as control-flow integrity, privilege minimization, specialized reference monitor, etc. [10], [11], [12], [13]. Common to all software approaches is the reliance on a trusted OS. However, many existing embedded systems, microcontrollers in particular, have a large amount of code in the privilege mode [14] for convenience of development or performance.

Trusted Execution Environment [15], [16], [17], [18] is a complementary approach that provides a powerful abstraction of a trusted machine even if the system software is compromised. TEE technologies, such as TrustZone, are now a de facto solution for mobile device security [19], [20], [21], [22], [23], [24], [25], [26]. However, similar to all existing software solutions, when the attackers can escalate their privilege into the OS, current TEE software stacks offer little assurance for system availability. Since the current TEE design only protects computation confidentiality [19] and integrity [20], management of resources, including process scheduling, is left to the non-secure OS. Recently, there has been increasing interest in enabling availability protection using new hardware designs [27], [28], [29], [30]. However, the application of such hardware primitives in real-time cyber-physical systems, such as autonomous drones, remains an open question.

Importance of availability in CPS: A defining characteristic of real-time CPSs is their continuous interaction with the physical world. Therefore, it is crucial that system resources are made available to safety-critical tasks in a timely manner. For example, the pedestrian detection algorithm on a self-driving car is a real-time task with a direct connection to the physical world process. A delay in the execution of this workload by the attacker can render the result utterly useless, since a catastrophic accident may have already happened, as recently demonstrated in [31], [32]. To further motivate the problem, we have also developed a new attack called time warping attack, which exploits access to Dynamic Voltage and Frequency Scaling (DVFS) to tamper with the timing characteristics of critical control components protected with TEE, leading to control destabilization crashing the robot.

Real-Time Trusted Execution Environment: In this paper, we assume a strong adversary that can exploit vulnerabilities in CPS firmware [5], [4], [6] to take control of the OS, and we address the research question of how to use security primitives on commodity embedded hardware to provide system availability assurance for real-world CPSs.

Our main contribution is the design, implementation, and evaluation of RT-TEE, a real-time trusted execution environment that protects system availability using hardware-assisted system resource partitioning on embedded platforms, such as ARM TrustZone. Availability entails the guarantee of timely access to system resources, including both computation (con-

trol) and I/O (sensing and actuation). However, moving the critical processes and resource management into the TEE not only significantly increases the system trusted computing base (TCB), but also degrades the performance. There are three key research challenges:

Challenge 1) Minimal hardware abstraction for availability guarantee in CPS: To provide availability guarantee, the TCB has to assert complete mediation over resources needed by the safety/security critical tasks for availability. However, resource management is commonly implemented by the untrusted OS in existing TEE designs. Building on the concept of control loops, we formulate the requirements on the minimal set of capabilities the hardware has to provide and show how they can be met using primitives from the TEE. This allows us to construct the rest of the design using a minimized hardware abstraction. From a high level, to ensure availability for CPS, RT-TEE relies on the secure timer to trap execution back to TCB to provide computational availability for the control. It also relies on a secure I/O reference monitor to enforce isolation and protection for sensing and actuation.

Challenge 2) Real-time computation availability: Due to the strong temporal affinities of CPS, computation resources not only have to be available, but also have to be in real-time. Contrary to the popular belief that real-time processes have to finish in a very short time, the key requirement is on meeting the deadlines from the perspective of real-time computing [33], [34]. This is typically accomplished using a trusted real-time scheduler. A naive approach is to directly implement such a scheduler inside the TCB for all secure and non-secure processes, but it significantly increases the TCB complexity. Another approach is to always complete the secure tasks first, also known as idle scheduling. However, this design can lead to unnecessary starvation of non-secure tasks, which hurts overall system performance since critical/secure tasks may not need to be executed immediately; they just need to be completed before the deadlines. For example, the battery checking task is secure safety-critical but doesn't have to be executed immediately, while the video streaming application on the drone is not safety-critical but should be accommodated to the extent that secure tasks do not miss their deadlines.

To minimize the penalty on performance without significantly increasing the complexity of the secure scheduler, we propose a policy-based event-driven hierarchical scheduler. Our hierarchical scheduler has two layers. Only the top-level scheduler has to be added to the TCB to guarantee secure processes have the resources to meet the deadlines. This is because the theoretical guarantee on the completion of secure tasks by compositional schedulability analysis makes no assumption on the behavior of the non-secure environment.

Challenge 3) Fine-grained peripheral availability: Naive use of TEE to protect I/O resources is neither sufficient nor effective for two reasons. It is insufficient because device level protection may not be universally available on all peripherals. Using SPI bus as an example, the access control is coarse grained, only specifying if a security domain has access to the bus or not. It is also not effective because migrating device drivers into the TCB will significantly increase its complexity.

To enable fine-grained access control on the peripherals, we designed and implemented an I/O reference monitor on top of TEE to remove the assumption on trusted drivers. To minimize the impact on the TCB, we leveraged the unique characteristics of cyber-physical systems, where each control loop performs the same set of I/O actions, to allow for significant driver debloating, where only a subset of the driver functionality is maintained for sensing and actuation. To enable feature-rich drivers without increasing the TCB, we proposed to sandbox the driver in conjunction with the I/O reference monitor to prevent compromised drivers from harming the system.

Prototype and Contribution: We have implemented a prototype on both ARMv8-A and ARMv8-M architecture, using Raspberry Pi and NXP LPC 55S69 development board, respectively. Using Raspberry Pi as the controller running ArduPilot, we assembled a quadcopter to test the impact of security protection on both real-time properties and control variation. To show how the environment can be used, we presented two concrete use cases on autonomous drones, protecting the entire flight controller or just the fail-safe controller for emergency recovery of the autonomous aerial vehicle. To evaluate the performance and understand the limitation of our proposed approach, we conducted a series of experiments on both synthetic workloads and real-life applications on both simulator and real-world systems. We found that our RT-TEE introduces a small overhead in task execution time on real-world drone applications.

In Summary, we have made the following contributions,

- We designed and implemented a real-time trusted execution environment, RT-TEE, capable of ensuring real-time availability on both CPU and I/O for commodity embedded processors in the presence of a compromised OS, addressing a key requirement for safety-critical CPS/IoT.
- To balance real-time responsiveness and TCB minimization, we designed and developed a policy-based event-driven hierarchical scheduler. To minimize the attack surface of device drivers in the TCB, we developed an I/O reference monitor on top of driver debloating and sandboxing to ensure the real-time I/O availability.
- We implemented a prototype on both ARMv8-A and ARMv8-M processors[1]. We tested our system on both synthetic tasks as well as real-world applications, covering three CPS platforms, quadcopter, plane, and rover, in simulation. We also deployed RT-TEE on a real-life quadcopter to validate the feasibility.

II. Background and Motivation

Lack of Availability in Existing TEE Deployment Model: ARM processor families, which power more than 60% of embedded devices, have a long history of building a trusted

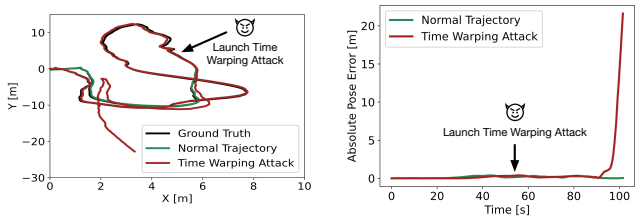
[1]The prototype source code and emulation environment is available at https://github.com/WUSTL-CSPL/RT-TEE

execution environment called ARM TrustZone into both low-end Cortex-M and high-end Cortex-A series. Similar to ARM, many commodity [16], [35], [15] and customized processor [18], [36], [17] offerings enable hardware-enforced resource isolation between the secure and non-secure environment, which are also referred to as the secure world and non-secure/normal world in ARM. Using such isolation, TEE offers a secure environment for secure processing even if the non-secure OS is compromised. However, based on the design principle of TCB minimization, most existing deployment models of powerful TEE hardware rely on the non-secure OS for resource orchestration. Using the current most widely deployed embedded TEE, TrustZone, as a case study, we surveyed all existing TEE software stacks, including nVidia TLK, Linaro OP-TEE, Trustonic TEE, Huawei iTrustee, Android Trusty, and Qualcomm TEE. All of them rely on the non-secure OS for resource management, including process scheduling. The detailed survey can be found in Appendix J.

Real-Time System Background: Contrary to the popular belief that real-time systems need to complete individual tasks quickly, the expectation in the real-time system community is that a task shall finish before its deadline [37]. A task is usually implemented as a thread in an OS. Real-time (RT) tasks are tasks with certain timing constraints. *Periodic tasks* are the most common model in real-time scheduling, because they map well to cyber-physical processes, where a task releases *jobs* periodically. The interval between two consecutive job releases is referred to as the *period*. Each job needs to be executed and completed before its *deadline*. A deadline can be explicit (specified) or implicit (at the end of a period). A hard real-time job must be completed before the deadline; completion past the deadline does not provide any utility and may lead to serious consequences. To facilitate scheduling, a *priority* is assigned to a task. The priority can be fixed (i.e., determined before run-time) or dynamic (i.e., changing based on the current tasks running in the system).

Security Implication of Real-time Property: The timing critical nature of CPS changes the landscape of attack vectors when the non-secure OS is compromised. Resources not only need to be made available, but also have to be available in a timely manner such that the computation can finish on time. To motivate the necessity of real-time scheduling for security, we developed a concrete attack called time warping attack that exploits DVFS and can destabilize the system even when the controller for the CPS is bug-free and protected by TEE.

Time Warping Attack: Dynamic Voltage and Frequency Scaling is a ubiquitous energy management technique that enables a trade-off between processor speed and energy consumption. During the schedulability test, the worst-case execution time is calculated based on the assumption of specific processor frequency. When it is changed, the original allocated budget for secure/critical tasks will no longer suffice. Since the frequency scaling attack can occur anytime during the execution of the secure environment by launching the attack on a different core occupied by the untrusted non-secure OS, the secure environment also faces the challenge of time-of-check vs time-of-use (ToCToU). This frequency reduction leads to a misconception of time elapsing in the secure environment, and results in control destabilization.



(a) Localization Trajectories (b) Absolute Pose Error

Fig. 1: Trajectory under Frequency Scaling Attack in Open-loop Testing.

To visualize the potential impact, we launched the attack against the control program of the drone, which is using VINS-Fusion [38] for localization. The testing environment is the EuroC drone dataset recorded in the ETH machine hall. Our attack lowered the frequency of the processor by half. As shown in Fig. 1a and Fig. 1b, the trajectory under Timing Warping Attack deviates significantly even in open-loop testing. Fig. 1b shows the deviations quantitatively with respect to time. At a certain range along the trajectory, the deviation is more than 3 meters, which would lead to the drone crashing into the machinery in the factory in the real world.

III. Threat Model and Security Goal

Threat Model: We aim to tackle a strong adversary who can exploit software vulnerabilities to execute arbitrary code in the non-secure OS. The attacker's goal is to launch a denial of service attack (DoS) to destabilize the cyber-physical system, causing adverse kinetic effect. Among all the possible attack vectors, an adversary may attempt to disable an I/O device to prevent sensing or actuation. He/she may also attempt to prevent the controller from running by denying access to processor/memory or by directly tampering with the controller memory. For example, an attacker may also attempt to delay pedestrian detection on self-driving vehicles. In a time warping attack, the processor frequency is maliciously modified to deny localization process access to computing, leading to the drone crashing. However, we leave the defense against side-channel attacks and hardware attacks for future work.

Assumptions: We assume the hardware platform supports a trusted execution environment that provides strong isolation on processor, memory, and peripherals. Isolation is a key design principle of TEE among both the commodity TEEs [35], [15], [39], [16] and customized TEEs [36], [30], [28]. We also assume that there is a time source capable of accounting for physical world time. We also assume all the hardware components can be trusted, including processor, sensor/actuator peripherals, and bus masters.

Security Goal: The goal of RT-TEE is to provide a real-time trusted execution environment on embedded CPSs. In the presence of an untrusted OS, RT-TEE has the following goals besides existing conventional TEE protection:

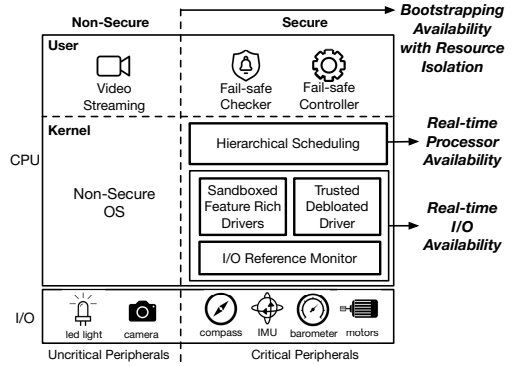


Fig. 2: System Overview for Trusted CPS Framework

R1. Access to Computation and I/O Resources. Cyber-physical control loops often involve sensing, computation, and actuation. To provide a minimalist execution environment for the cyber-physical system controller, the secure safety-critical process shall have access to both computation resources (for control) and I/O resources (for interaction with physical world with sensing and actuation).

R2. Access to Physical Passage of Time. A trustworthy source of the physical passage of time is crucial because the computing system is trying to control a physical world process. Inducing an actuation at the wrong time can easily destabilize the control system.

R3. Real-time Availability. Many safety-critical CPS controllers are real-time in nature. Results from processes such as pedestrian detection or collision avoidance have little to no value if they exceed the deadlines. Therefore, the resources shall be available to the protected workload in real-time.

R4. Prevention of DoS by Shutdown or Peripheral Damage. While DoS attacks often pose little risk to safety in IT computing environments, they can be catastrophic in CPSs. As a result, RT-TEE shall be able to prevent a non-secure OS from denying access to resources (processor and peripherals) by shutting down the computing system or putting sensors/actuators in non-recoverable states.

R5. Minimal Impact on TCB. Complete mediation of access to system resources is necessary for availability assurance. Even when only critical resources are mediated, there is a significant increase in the size of TCB, e.g., from the inclusion of device drivers. RT-TEE shall maintain a small TCB.

IV. RT-TEE Design

RT-TEE aims to provide real-time system availability in the presence of a non-secure operating system. As shown in Fig. 2, RT-TEE leverages hardware primitives to isolate resources between the secure environment and the non-secure environment. There are three key design elements in RT-TEE, bootstrapping availability, real-time processor availability, and real-time I/O availability. For the rest of the discussion, we will use ARM as a reference architecture. However, we believe the design philosophy can also benefit other platforms.

A. Minimal Abstraction for Resource Availability

The first challenge is to identify the minimal set of requirements to bootstrap resource availability such that the rest of the design can build on top of a hardware abstraction layer. While TEEs such as TrustZone are designed to provide resource isolation, separation alone does not provide availability, especially when the software component (OS) that is responsible for resource management is not in the TCB. Recognizing that trustworthy resource management is not possible, we then ask what the essential components of CPS are and what type of availability guarantee should be provided. A foundational design for many CPSs is the control loop, which consists of sensing, control, and actuation. Deriving from this requirement, RT-TEE has to ensure the integrity and availability of computation for control computation and I/O for sensing and actuation.

Primitive for Bootstrapping Computation Availability: There are two requirements. First, the TCB has to be able to regain control of the processor from the non-secure OS to perform computation. This requires a secure timer which is not modifiable by the non-secure OS and traps the processor directly into the secure environment. Secure timer is also instrumental for real-time responsiveness, since it allows the secure environment to obtain processor resources in a timely and deterministic manner. Due to the importance of such features, secure timer is widely available in commodity embedded TEE platforms. According to our survey, secure timer is available in 19 out of 21 processors that support ARM TrustZone. More details can be found in Appendix A. Furthermore, research prototypes such as AION [28] and others [30], [40], [18], [17], [41] also provide secure timer as a primitive. Second, the secure environment also needs to maintain access to the processor to finish the necessary computation. Therefore it is important for the TCB to have the ability to prevent interruptions from the non-secure OS. Using the ability to regain control and prevent further interruptions, the TCB can bootstrap the computational availability.

Primitive for Boostrapping I/O Availability: Recognizing the importance of peripheral access in embedded systems, many existing embedded TEEs provide primitives for the TCB to obtain exclusive access to individual I/O devices [35], [15], [42], [28]. For example, different from SGX, which is widely deployed in server platforms, both ARM TrustZone and the customized TEE SoC proposed in [30] have primitives to enable exclusive I/O access. Once the secure environment has exclusive access, it can leverage software to mediate all the requests to the peripheral devices to ensure prioritization, bootstrapping the I/O availability.

Primitive for Obtaining Physical Passage of Time: CPS has to sense and actuate at the right time to ensure safety during its interaction with the physical world [31], [33]. As a result, accurate accounting of time passage in the physical world is essential. The secure physical world clock primitive can be realized using a SoC-provided non-mutable clock or software-based time keeping on top of a secure physical timer.

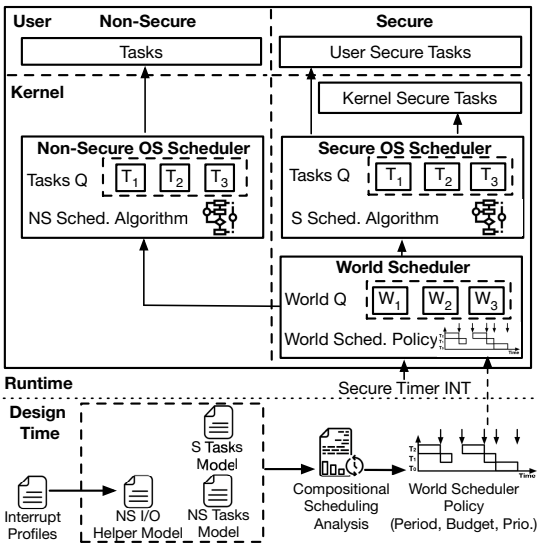


Fig. 3: RT-TEE Scheduling Architecture

B. Real-time Availability for Computation

Building on top of the bootstrapped computation availability, a key research question is how to ensure the real-time property of computation availability. In safety-critical real-time cyber-physical systems, security, criticality, and timeliness are often the most important attributes of individual real-time tasks (processes). Security often describes the ability to maintain confidentiality and integrity against adversarial attacks. Existing TEEs are designed to protect these two properties. Criticality reflects the importance of the task for the correct functioning of the cyber-physical system. Security and criticality are often aligned; critical tasks also need to be secure. Timeliness captures the need to complete a particular task in a timely manner, i.e., completing before the deadline. To ensure timeliness, system designers have to perform schedulability analysis [43], [33] at design time. The analysis takes in a set of real-time task characteristics (periods, deadlines, budgets) and produces a set of scheduling parameters that can guarantee that all tasks will have the computation resources to finish before the deadlines, regardless of how the tasks are using the allocated resources. However, direct adaption of secure scheduling cannot provide an effective trade-off between security (TCB complexity) and performance (meeting deadlines).

Challenges in Secure Scheduler Design that Balances Real-Time Performance and TCB Minimization: One naive scheduler design is static allocation without coordination, with the non-secure OS and the secure environment occupying completely different cores; however, this hurts overall system performance. An alternative design is to maximize performance by having a single scheduler for both secure and non-secure environments. Most of the current TEEs adopt this paradigm and have a global scheduler in the non-secure OS. However, if we move the global scheduler into the secure environment to offer availability, the secure scheduler has to understand the semantic and maintain run-time information for all the non-secure tasks running in the general purpose OS as well. This will significantly increase the complexity of the scheduler, leading to a large expansion of the TCB. To make such coordination more challenging, the secure TCB also has to assume all schedule-related inputs from the non-secure environment are malicious. As a result, interaction between the two environments on scheduling has to be minimized.

One scheduling method that minimizes interactions is to always prioritize secure critical tasks. This approach is called idle scheduling and has been adopted in previous works [44], [45]. However, while security and criticality are often aligned, secure critical tasks may not necessarily require the shortest response time. For example, it is better for the system to process the video streaming workload first, as long as the control tasks meet their deadline and are capable of maintaining control of the vehicle frame. As a result, idle scheduling can lead to degraded overall system performance while gaining little advantage in control performance. These design trade-offs are often observed in mixed-criticality systems (MCSs), a concept that originally came from avionics. Due to increasing pressure to reduce space, weight, and power (SWaP), most modern CPSs consolidate functionality on single physical processors, often making them MCSs [34], [33].

Our Solution – Policy-based Event-driven Hierarchical Scheduling for Real-time Computation Availability: To decouple task security/criticality from timeliness/task priority, and to enable better overall system performance without increasing the complexity of the secure scheduler, we make use of hierarchical scheduling [46] to harmonize security/-criticality and timeliness. Hierarchical scheduling is a layered approach where the scheduler at each layer is only responsible for subsystems in that layer. Using composition scheduling analysis, it is possible to produce a system specification that will satisfy all the real-time constraints of all the tasks. In other words, as long as the scheduler strictly follows the specification/policy produced by the analysis, all tasks (secure and non-secure) are guaranteed to have the computation resources to finish before the deadlines. RT-TEE employs a two-layer hierarchical scheduling design to decouple real-time properties (priority) with security properties (security/criticality), enabling more effective resource coordination between secure and non-secure environments with minimized interaction.

However, direct application of conventional hierarchical scheduling in RT-TEE has several significant drawbacks. First, a conventional scheduling system often makes use of timer to interrupt at fixed intervals, usually single jiffy, to provide time tracking. While this overhead is tolerable without the security architecture, frequent timer interrupts are prohibitively expensive in RT-TEE due to the expensive context switching. To tackle this challenge, our scheduler leverages the property that tasks are predictable in cyber-physical systems to only interrupt based on scheduling events, such as budget replenishment and task completions. The event-driven scheduling system significantly reduces the interrupt overhead in the scheduling system. Furthermore, different from existing scheduling systems where I/O events can interrupt the running

process, RT-TEE needs to take into consideration the security properties of the I/O devices and the nature of the interrupt since a compromised non-secure OS can program a peripheral to continuously raise interrupts to launch a DoS attack. While it is possible to completely disable interrupts from non-secure peripherals or assign all the peripherals to the secure environments, neither of these approaches is ideal. To provide better I/O response time to the non-secure environment, it might be possible to enable non-secure interrupt for a limited number of times over a short duration, based on the security policy. This, however, changes the schedulability analysis due to lack of time accounting in interrupt context, which the non-secure OS can exploit to perform denial of service. To enable schedulability analysis on this mechanism, we propose to create a synthetic non-secure I/O serving task to capture its impact on real-time scheduling.

The resulting design is a policy-based event-driven two-layer hierarchical scheduler as shown in Fig. 3. The first layer is the world scheduler, responsible for scheduling the individual execution environment. The second layer is the OS scheduler inside the individual execution environment.

At design time, the system designer has to first determine the real-time task models for both the secure and non-secure environment. She will also need to determine the non-secure interrupt profiles based on the security policies, which will also be translated into a real-time non-secure task for scheduling. The synthetic interrupt task and the original task models are then passed to the compositional scheduling analysis engine, which will produce a solution on the concrete scheduling parameters used by the world scheduler.

At runtime, when a secure timer interrupt is triggered, the execution will be redirected to the world scheduler, which will resume either the secure environment or the non-secure environment based on the scheduling policy. When an individual environment is resumed, the OS will execute under the abstraction that it owns the entire system. Since the scheduler is event-driven, before resuming a world, it has to anticipate the next scheduling event, such as job release from higher priority jobs or budget replenishment, to set the timer appropriately to minimize overhead. From the real-time availability perspective, in real-time systems, the budget replenishment server algorithm is the mechanism to ensure timing resource isolation since it assigns a portion of the bandwidth (cycles) of the CPU to different tasks. Together with the scheduling system that ensures priority, the replenishment process enforces the timing policy of the system. RT-TEE is designed to support different types of scheduling algorithms.

C. Fine-grained I/O Access Control for Peripheral Availability

Limitations of Direct Application of TEE: While TEEs, such as TrustZone, are designed to support resource isolation, there are several drawbacks with naive adaption of the primitive. 1) Existing implementation of TEE often provides exclusive access to a peripheral unit on the SoC. However, when the peripheral device is a bus controller (especially the simpler serial buses such as SPI or I2C), the access control provided by hardware is coarse-grained in that they can only be granted at the bus level. Even if only one connected sensor is security-critical, every device access via that bus has to go through the secure TCB. In the context of CPS, most sensors are connected to the processor via either SPI or I2C [47], [48], presenting realistic challenges. 2) When I/O devices are assigned to the secure environment, the device drivers will need to be included in the secure kernel, resulting in a significantly larger TCB. Furthermore, a malicious command from either a compromised driver or a confused deputy can put sensors or actuators into a non-reversible state, affecting availability. For example, overcharging the servo motor may damage the mechanical components to the extent that it won't restart anymore. Given the number of driver vulnerabilities disclosed daily, it is very difficult to ensure all drivers are bug-free.

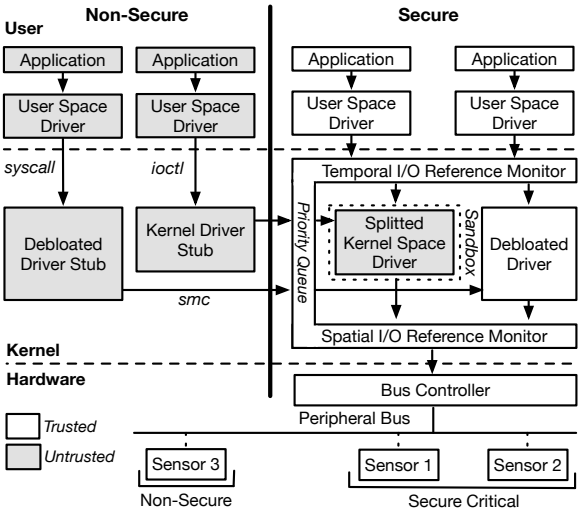


Fig. 4: RT-TEE Secure I/O Architecture

I/O Protection Goals: To address the aforementioned limitations, we propose to remove the assumption of trusted drivers. Therefore, in the presence of compromised drivers executing in secure privilege mode, RT-TEE has to provide the following properties. 1) *IO_R1:* TCB for secure I/O shall be minimized. 2) *IO_R2:* Policy-driven fine-grained access control on I/O devices – The interactions between processes (mainly the non-secure OS) and devices shall follow the security policy. 3) *IO_R3:* I/O availability – Access to secure devices shall not be impacted by the non-secure environment. 4) *IO_R4:* Real-time I/O availability – Access to peripherals has to be timely.

I/O Solution Overview: An overview of our solution is shown in Fig. 4, there are two key components.

TCB minimization – We adapt two instances of drivers for each peripheral. One instance offers rich functionality, but cannot be trusted and is sandboxed. The other one provides minimal functionality but is instead trusted. For the trusted minimal instance, we leverage the predictable nature of CPS to debloat the drivers via templatized transformation, which essentially replays known hardware commands and therefore only incurs a small overhead on the TCB. I/O operations are transactionized to ensure that the peripheral hardware can be shared. Using the

bus controller as an example, one solution is to have the secure and safety-critical devices on the bus driven by the debloated driver, while the non-secure ones are driven by the sandboxed feature-rich driver.

I/O reference monitor – It is responsible for peripheral access control and therefore is responsible for availability. There are two key dimensions of access control, spatial, and temporal. The spatial aspect of the I/O reference monitor mediates what peripherals a process can access and how it can interact with the device. The temporal aspect of the I/O reference monitor mediates over when a process can interact with a peripheral and for how long. Together they ensure the real-time availability of the system.

TCB Minimization by Debloating Trusted I/O Drivers: In order for a secure process to make use of the secure peripherals, there has to be a driver. The key research challenge is how to provide such functionality without including the complex and buggy drivers in the TCB. The debloating technique we proposed is based on the observation that CPSs are highly predictable by design. Such predictability also translates to its interaction with sensors and actuators, where most of the device interactions are exactly the same. Therefore, there is an opportunity to convert complex hardware interactions to simple replay of the fixed I/O interactions, trading expressiveness for reduced attack surface, minimizing the code size.

There are two opportunities to debloat the driver, pruning and transformation. Pruning refers to elimination of functionality. For most of the peripheral devices in CPSs, there are primarily two stages, initializing and utilization of the device. Based on the assumption that the system has secure boot, device initializations are executed in the non-secure environment as part of the secure boot. Immediately after the initialization, access to the devices from the non-secure environment is blocked. This insight allows for significant reduction in the code base. For example, in the Navio2 platform, there are more than 30 I/O request types during initialization for bus probing and device initialization, but only half a dozen message types for sensor and actuation interactions.

The second opportunity is transformation. An overview of our approach is shown in Fig. 5 where code that drives hardware interactions (such as sensor reads) is converted to data recording templates capturing the interaction patterns (what content was written to which MMIO register in what order). A driver replayer then repeats the template to drive the peripheral. More specifically, we record all the MMIO register reads and writes, and repeat these data recordings to drive the peripheral. However, though predictable, the interactions may not always stay the same. One of challenges is understanding the semantic of the message and recognizing what is always fixed (such as TA bit in the control register) and what is constantly changing (such as motor speed). For peripherals on the SoC, the driver often resides within the kernel. However, for sensors and actuators connected via a bus such as SPI and I2C, there are both the kernel space driver and the user space driver. As a result, the recording captures

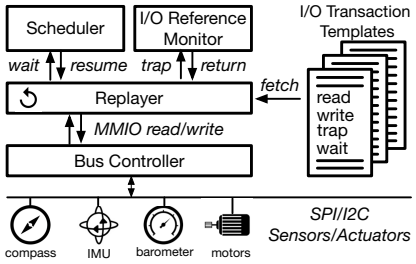


Fig. 5: Debloated Driver

the logic of both the user space driver and the kernel space driver for the peripheral. To facilitate template generation, we apply taint analysis to understand how the control program is driving device interactions. In the user space, we tracked what information is control-supplied in the buffer passed to the kernel, often via IOCTL. In the kernel space, we tracked how user space inputs are used in the MMIO interactions, and therefore were able to mark the variable offsets in the data recording to create an interaction template. At runtime, all the I/O addresses will be statically mapped, and all the DMA memory is statically allocated. The replayer will fill in the template with the appropriate values from the request after verification by the I/O reference monitor. At the minimum, the replayer has to be able to repeat MMIO reads and writes. However, to improve performance, support IRQ and DMA, and support real-time availability, we also added support for schedule timeout. To provide the I/O reference monitor with the ability to mediate access, we also added an operation type in the template to trap the replayer in the reference monitor.

TCB Minimization by Sandboxing Untrusted Drivers: While the debloated drivers are sufficient to support interactions with safety-critical sensors and actuators, it requires predictability and a well-formatted template. For non-critical functions in the peripheral or non-secure devices on the bus, it may not require the same level of security protection or may not even have as predictable patterns as those used for CPS controls. To enable these feature-rich drivers for non-secure devices, we propose to sandbox them using software instrumentation. However, software-fault-isolation (SFI) techniques only prevent the untrusted driver from accessing the rest of the secure environment; there are several additional challenges.

First, drivers have to access the MMIO registers to interact with the hardware. For bus controllers, these same sets of MMIO registers also allow for interaction with the secure safety-critical peripherals. For example, the sandboxed driver could change the chip select bits to send messages to the secure actuator instead. To prevent the driver from tampering with secure devices, all MMIO register reads/writes are instrumented to trap into the I/O reference monitor to enforce the access control policy. Furthermore, besides adding CFI, all the other memory reads/writes will be masked to confine the driver, preventing data-only attacks on MMIO registers.

Second, device drivers often use many kernel helper functions, migrating a full-feature driver into the secure environment where the kernel is intentionally minimized is likely not

possible without increasing the size of TCB. To address this, we adopted a split driver design where most kernel/application facing functionalities are implemented in the non-secure environment, while the hardware interacting code resides in the sandboxed secure environment. On one extreme, almost the entire device driver can reside in the non-secure OS, while each register's reads/writes are redirected to the secure environment split. However, for performance consideration, the balance often lies in putting the functions with large numbers of register operations into the secure split.

Availability via Spatial I/O Reference Monitor: The I/O reference monitor that mediates all the secure I/O devices' access is the component that ensures availability. There are two main attack vectors an adversary can use to deny secure task access to a peripheral, corrupting peripheral state (e.g., sending random bytes to secure devices on the bus or to the tx-fifo of bus controller) or physically damaging the peripheral (e.g., overcharging the motor). For peripherals directly connected to the SoC without a bus, it is either a secure device or a non-secure one. To ensure its availability, the I/O reference monitor can simply make sure that only secure processes have access to the device. However, most sensors and actuators are connected via a serial bus. For devices on the bus, availability assurance presents additional challenges.

To prevent the untrusted driver in the sandbox from tampering with secure devices on the same bus, each MMIO access will be checked to ensure that the driver has access rights to the device, particularly by monitoring the chip select line or chip select bits in the control register. An adversary may also attempt to corrupt the bus controller by leaving contents in the tx-fifo before handing the access to the secure debloated driver such that the driver may send existing contents to the secure devices. To prevent this, the I/O reference monitor will ensure that the controller state is reset upon switching between the secure debloated and non-secure sandboxed instances of the drivers. To prevent physical damage to the sensor, the I/O reference monitor can verify the values in commands sent to sensors and actuators are within validated range. Since our reference monitor has the ability to introspect every single MMIO access, the policy can be as rich as is demanded by the mission. For example, sensor access control can even be done on the sensor command level instead of the device level.

The I/O reference monitor also protects the system from full system shutdown and time warping attacks. The DVFS-based attack presented in TimeWarp earlier and system shutdown relies on interactions with the power management (PM) and clock management (CM) on the SoC. The I/O reference monitor can prevent malicious use of these peripheral features. While it may still be possible to offer limited access to the power/clock management features, such as shutdown, the impact on availability requires careful considerations. For example, allowing access to DVFS would require the secure scheduler to switch to a task profile to ensure availability.

Real-time I/O Availability via Temporal I/O Reference Monitor: Once I/O device availability is established, achieving real-time availability means that ensuring I/O requests are processed according to real-time scheduling policy such that secure tasks can access the necessary resource in a timely manner. A key challenge in this is priority inversion, in which higher priority tasks are waiting for a lower priority task on a shared resource. However, priority inversion is often inevitable since marginal gain from interrupting the lowest level I/O operation, such as transmitting a single message, is often very small. As a result, the main approach to achieve real-time I/O operation is reducing priority inversion time [49], [50]. In RT-TEE, individual I/O request will be placed in different priority queues based on the inherited priority from the requesting process. However, processing of individual I/O requests on the same peripheral device is not preemptible, thereby creating a priority inversion. To ensure real-time responsiveness, the I/O reference monitor will bound the I/O request processing time by enforcing a time limit on an I/O job, therefore bounding the priority inversion time and the worst case execution time (WCET) of the job.

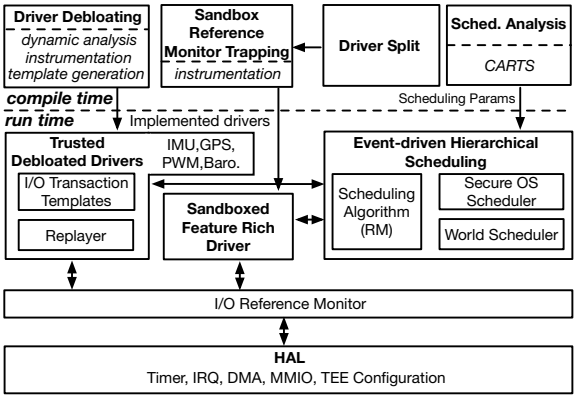


Fig. 6: RT-TEE Implementation

V. Implementation

We implemented a prototype of RT-TEE on both ARMv8-A and ARMv8-M architectures. For ARMv8-A, we built on top of OP-TEE v3.4 for the secure kernel and Raspbian Linux 4.14.95-emlid-v7+ for the normal world. For ARMv8-M, we build on top of the evaluation firmware OS from NXP. As shown in Figure 6, there are three main implementation efforts, HAL, hierarchical scheduling, and secure I/O. For both the scheduling and secure I/O, there are both design/compile time components and run-time components. Due to space limitations, the implementation of HAL can be found in Appendix B.

Hierarchical Scheduler: To support real-time scheduling, we implemented a customizable hierarchical scheduler on top of the RT-TEE HAL. The basic components in a scheduling system include management of tasks and prioritization. On the task management side, since the current OP-TEE does not support secure scheduling natively, we implemented our own context switching and task structure on top of the current OP-TEE thread pool architecture using the Linux scheduling subsystem as a reference architecture. From the prioritization side, our

scheduler framework is fully modularized. It provides the basic API to developers for building different real-time algorithms, such as world_budget_update and world_prio_recalc. Besides the infrastructure, we also implemented several concrete prioritization and budget replenishment algorithms. Our prototype uses partitioned Rate-Monotonic (RM) with deferrable server for the world scheduler and RM scheduler for the secure OS.

Debloated Driver: To record all the sensor operations, we instrumented the lowest level kernel functions, writel and readl, then exercised the sensor operations from the user space device driver. The recording contains a list of tuples [r/w, MMIO address, content]. This list forms the basis of interaction template. For many sensor reads, this list is fixed because the driver is repeatedly asking the sensor for the last readings. For other interactions, such as motor speed, the value is constantly changing; however, the format remains the same. For polling-based drivers, this list of MMIO reads/writes is enough. The replay simply reads the tuple from memory, and writes to the MMIO addresses using the hardcoded data or reads from the registers. To support DMA on the template driver, we also statically allocated memory and harcoded both the structures and addresses into the template. Besides read/write, our recording also records use of IRQ by instrumenting the wait_for_completion function. For these types of interactions, the template needs to be paused until IRQ is received, upon which the replay can continue. We implemented the template driver for all the sensors (5) and actuators (1) on Navio2.

Sandboxed Driver Implementation: There are two main efforts, the process of splitting the driver into secure and non-secure halves, and the process of instrumenting the secure split to enable CFI, memory read/write sandbox, and MMIO read/write trapping. We implemented one split driver for the SPI bus controller since it is one of the most complex drivers that is used by a majority of sensors and actuators on our drone prototype. The ArduPilot program, containing millions of SLOCs, also provides user space sensor drivers, such as the AP_InertialSensor_Invensense.cpp, that interact with the kernel using the IOCTL interface. Decoupling the existing drivers from the non-secure OS and completely moving it to secure environment is impractical and does not benefit security, but having only hardware interaction (i.e., MMIO reads/writes) in the secure world is also not a good solution since it introduces a significant amount of overhead. To minimize the performance penalty, we started with the main transfer data function bcm2835_spi_transfer_one and moved all its dependencies to the secure world but manually kept all the linux kernel structures away. In the future, this can be an automatic process. Once the driver was splitted, we also replaced all the calls to readl/writel with traps to the I/O reference monitor. To provide forward edge CFI, we also unrolled all the indirect targets. The backward edge CFI is implemented using a shadow stack by instrumenting function entry and exit. Lastly, we also instrumented all the memory reads/writes to sandbox the secure driver by masking.

I/O Reference Monitor Implementation: The RT-TEE I/O reference monitor has two main components, the spatial reference monitor and the temporal reference monitor. The spatial I/O reference is highly policy dependent. In our implementation, we implemented chip select checking for bus communication and range checking for motors. To ensure I/O requests are processed in real-time, all requests are processed based on the priority of the process. There are multiple queues for different ranges of priority levels and the corresponding I/O tasks that handle requests in this queue. Therefore, it is configurable from one queue per priority to one queue in the entire system. In our prototype, we use a single queue and assign all secure requests to the top half of the priority and non-secure requests to the other half. Besides prioritization, another key aspect is in bounding the priority inversion time. To do so, we measure the runtime statistics of the peripheral interactions and use the scheduler to enforce a time limit on each I/O operation.

VI. Evaluation

Our evaluation of the system focused on understanding the added overhead of security on real-time systems. The microbenchmarks measure the overhead of individual components of RT-TEE, while the macrobenchmarks examine the overall system performance under both synthetic workload and real-world CPS applications. We have two evaluation platforms, one for each ARMv8 architecture. For ARMv8-A, we evaluated on a self-built quadcopter with Raspberry Pi 3 Model B powered by ARM Cortex-A53 and with Navio2 board as the controller. For ARMv8-M, we performed the evaluation on the LPC55S69 micro-controller. For real-world CPS applications, we chose ArduPilot [51], one of the most widely used open-source autonomous vehicle controllers. Since hardware-in-the-loop (HITL) is not supported in ArduPilot, we had to resort to software-in-the-loop (SITL) for evaluating platforms that we don't have hardware for. To minimize the impact of simulation on control performance, we ensured the platform resource utilization was below capacity and that no tasks were missing deadlines. We also added delays to the SITL to simulate actual sensor/actuator response. Lastly, we built a drone using RPI3 as the flight controller to demonstrate feasibility.

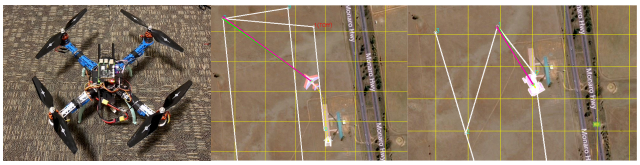


Fig. 7: Experiment Platform

A. Microbenchmark on Scheduling

The microbenchmark, shown in Table I, measures the overheads introduced by the system. All the measurements are an average of 100 runs. Task invocation delay measures the delay between scheduler deciding which tasks to run and the actual start of the task execution. The conventional non-secure-based scheduling mechanism in OP-TEE takes around 53 μs and 111 μs to invoke a secure kernel space function and a secure

TABLE I: RT-TEE Scheduling Overhead

Runtime Overhead	RT-TEE	OP-TEE
S EL0 Task Invoc. Delay	89.35 us	111.00 us
S EL1 Task Invoc. Delay	16.51 us	53.00 us
World Scheduling Delay	0.94 us	N/A
Workload Dis: 50%(NS) 50%(S) Total Workload: 70% (Run Time16.91s)		
Total Scheduling Event	2713(W)/2155(S)/4156(NS)	16751(N)
Total Scheduling Overhead	158.24 ms	617.95 ms

user space function, respectively. It takes around 16.51 μs and 89.35 μs under RT-TEE. The performance gain is mainly due to significant reduction in context switch overhead for invoking secure world functions. The difference between user task and kernel task is primarily due to the user process setup. We also measured the task invocation time on the LPC55S69 micro-controller. The invocation delay is 0.35 μs with RT-TEE and 0.31 μs without RT-TEE. The invocation delay is shorter due to a much simpler context switch. To measure the efficiency of the event-driven scheduler adopted by RT-TEE, we instrumented RT-TEE hierarchical scheduler and non-secure OS scheduler in Linux to record the scheduling event count and the total overhead over the execution of a set of real-time tasks executing in both S and NS environment over 16.91s. As shown in Table.I, event-driven scheduler has 46.13% fewer scheduling events and 74.40% less scheduling runtime overhead than conventional time-based scheduler.

B. Micro-benchmark on I/O

I/O – Debloated Driver: To find the number of sensor operations, we instrumented all the IOCTL in ArduPilot. We found that during drone operation, there are six sensor/actuator operations. To demonstrate driver flexibility, we implemented and evaluated the debloated/template-based drivers for all the sensor and actuator operations needed to operate the drone. Each operation has its own individual template, as shown in Table. II. These drivers encompass 2 buses (SPI and I2C) and three mechanisms of message delivery (DMA, IRQ, and Poll). However, driver interaction with peripherals always follows a similar protocol, which allows us to turn them into templates. During the experiments, we attempted to speed up the I/O operations in the template but failed due to the bus speed. As a result, we decided to adhere to the existing delays in the template. To compare with the native driver, we use the same user space driver, but instead invoked the template version. The runtime overheads of our debloated drivers are shown in Table II. The latency is comparable to the original drivers. We believe this is due to the fact that the debloated drivers save complex configuration and simply replay data; however, they suffer from world switches as well as I/O queue processing.

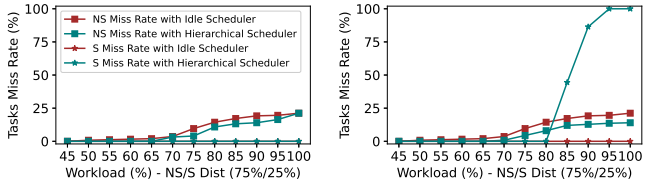
TABLE II: Debloated Driver Statistics

Sensors/Bus	Operation Semantics	Tpl. Size	Transfer Mech.	Native Avg(us)	Debloated Avg(us)
MPU9250/SPI	R Inertial	117	DMA	157	165
AK8963/SPI	R Compass	126	Poll	41	46
LSM9DS1/SPI	R Compass	126	Poll	37	43
UbloxM8N/SPI	R GPS	124	DMA	227	235
Motors/SPI	W Motors	4374	IRQ	782	792
MS5611/I2C	R Temp/Pres	225	IRQ	77	84

I/O – Sandboxed Driver Overhead: We split the SPI controller driver for RPI3 as a feasibility study. To identify the best split between the secure and non-secure parts, we dynamically analyzed the driver to identify the function in the execution trace that will have the least amount of Linux structure dependency and the largest group of I/O operations. Currently, this is a manual process. Once the function was identified, we proceeded to extract all the dependencies of the function along with the structures they refer to and flatten them. Using the self-contained driver, we then sandboxed it using the GCC pass. The non-secure split was also modified to call secure world split upon entry of the function. The average overhead was 6.86% for runtime and 13.75% for code size. The overhead was within our expectation since average masking overhead is around 5%, and there is only a single invocation to the secure world.

C. Macro-benchmark with Synthetic Tasks

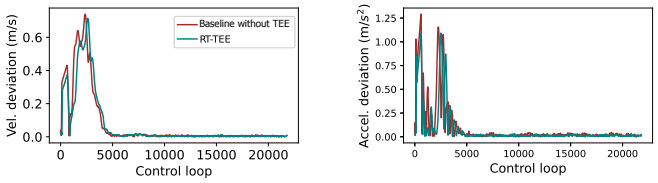
Using the synthetic tasks, we answered the question of whether hierarchical scheduling has better real-time performance than idle scheduling at different system loads. The performance is measured by how few tasks miss their deadlines. Particularly, we wanted to see heirachical scheduler deliver on the promise of not missing secure task deadlines even if non-secure environment is prioritized. All experiments were conducted using ten synthetic tasks. The actual execution was randomly generated from 1ms to 20ms, with five as secure tasks and five as nonsecure tasks. Given the real-time task specification, we used CARTS [43] to calculate budgets for each world, which ensures the schedulability of tasks.



(a) Prioritizing Secure (b) Prioritizing Non-Secure

Fig. 8: Tasks Miss Rate Compared with Idle Scheduling on ARMv8-A

Idle Scheduling Performance Comparison: Fig. 8 shows real-time performance comparison between idle scheduling and hierarchical scheduling. We can make the following observations. 1) Hierarchical scheduling has better real-time performance when secure world is prioritized in hierarchical scheduling since task miss rate is always lower with hierarchical scheduling, as shown in Fig. 8a. 2) Hierarchical scheduling always outperforms idle scheduling when system load is below 80%, as shown in Fig. 8b. In other words, our scheduling structure allows better responsiveness of non-secure tasks without impacting secure workloads as long as the system is designed to be below 80% utilization, which is a reasonable assumption since many CPS systems are not schedulable when the system utilization is high (69% for rate monotonic systems [37]). In other words, when the real-time task set is schedulable, it is



(a) Velocity deviation (b) Acceleration deviation

Fig. 9: Control Performance with RT-TEE and Baseline w/o TEE on Copter

always better to leverage hierarchical scheduling to provide better non-secure task responsiveness.

D. Macro-benchmark with Real-world CPS in Simulation

While synthetic tasks are informative and allow us to explore the various conditions the system can face from the computation perspective, we are also interested in seeing how the system can be used to protect real-world CPS applications. Therefore, we used RT-TEE to protect three real-world applications to show the feasibility and potential limitations. Particularly, we used RT-TEE to protect fail-safe controller and attitude controller on the quadcopter and fail-safe controller on the plane and rover platforms. Fail-safe controller was selected because it can be used to detect unsafe physical condition of the autonomous system, while attitude controller was selected for its ability to act as an enforcer for mission trajectory. We used CMAC-copter-circuit, CMAC-circuit, and CMAC-bigloop of ArduPilot autotest as missions for quadcopter, plane, and rover, respectively.

RT-TEE Impact on Control Performance: To understand the impact of RT-TEE on the control, we recorded the actual position, velocity, and acceleration of the autonomous vehicle, and compared that to the reference state. The deviation between observed state and reference state is quantified using integral absolute error (IAE), a widely used stability metric in control systems. Shown in Fig. 9 and Table III are the measurements during the test flight. The control deviation introduced by RT-TEE is within 13%, and can be further reduced by tuning the scheduling of the attitude controller. Shown in Fig. 9a and Fig. 9b, the performance under stabilized system after the short period of convergence is comparable to the baseline. The control performance for rover and plane can be found in Appendix. E.

Macro-benchmarks on Real-life Copter: We measured the average execution time of 27 ArduCopter task executions over 100 times when flying an actual quadcopter with RT-TEE. The average execution time of each real-time task in ArduCopter reflects the overall runtime overhead generated by RT-TEE components, such as secure timer interrupt handler, world scheduling, secure task scheduling, I/O reference monitor, and debloated driver. To measure task execution time in ArduCopter, we instrumented each real-time task to record task start time and end time to calculate the runtime. To minimize time recording overhead, we stored all the data into pre-allocated memory and exported them when a copter mission finished. As shown in Fig. 10, none of real-time tasks exceeding their max execution time, demonstrating means the runtime overhead generated by RT-TEE components is acceptable in ArduCopter.

TABLE III: Control Deviation of ArduCopter with RT-TEE

	Baseline without TEE		RT-TEE		
Control Var	Average	Max	Average	Max	Avg Overhead %
Δ X-axis pos.	2.029	2.627	2.138	2.944	5.37
Δ Y-axis pos.	5.917	8.043	6.643	9.047	12.27
Δ X-axis vel.	0.009	0.277	0.010	0.300	11.11
Δ Y-axis vel.	0.020	0.766	0.023	0.790	12.75
Δ X-axis accel.	0.008	0.091	0.009	0.084	12.50
Δ Y-axis accel.	0.012	0.230	0.013	0.250	8.33

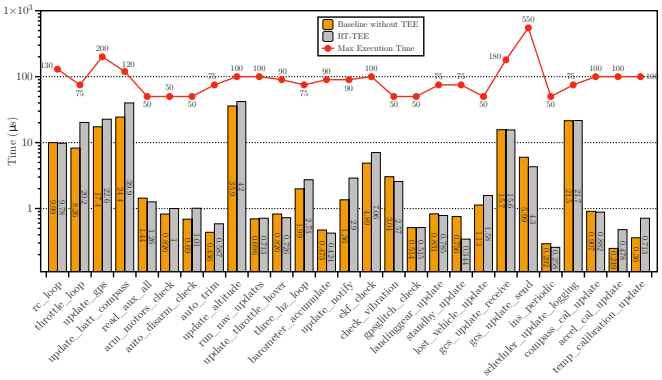
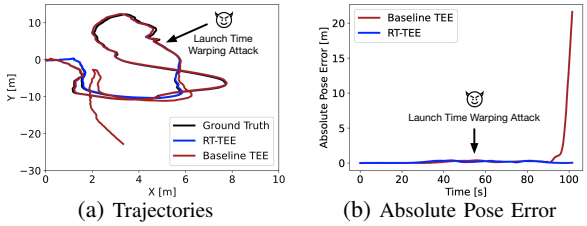


Fig. 10: Runtime Overhead with RT-TEE and Baseline without TEE

Defense Case Study: Fig. 11a and Fig. 11b shows the defense effectiveness of RT-TEE quantitatively under the same setting as the time warping attack introduced earlier in the motivation, between baseline TEE (such as OP-TEE) and RT-TEE. It can be observed that under the protection of RT-TEE, the absolute pose error remains very small even after the attack is launched.



(a) Trajectories (b) Absolute Pose Error

Fig. 11: Time Warping Attack on Baseline and RT-TEE

VII. Security Analysis

A. Real-time Computational Availability – R1, R2, R3

Computation Availability – R1: The availability guarantee on a processor is through the use of non-bypassable timers. On ARM platforms, this is the secure physical timer, which is not modifiable by the non-secure OS. When the timer triggers, the secure world regains the non-interruptable control on the processor. Similar primitives are also available on other hardware platforms [30], [28], [35].

Physical Passage of Time – R2: Physical world interaction requires accurate timing on sensing and actuation. Malicious modification to time can lead to inaccurate perceptions. Physical passage of time can be obtained from the generic timer on ARM or the trusted timer on RISC-V. Neither of these timers is not modifiable by software.

Real-time Computational Availability – R3: From the computation perspective, real-time implies that secure safety-critical tasks have appropriate allocation of processor time to meet their corresponding deadlines. In RT-TEE, this is accomplished using a real-time hierarchical scheduler which enforces the real-time scheduling policy. For all types of real-time schedulers, including hierarchical scheduler, when schedulability analysis [43], [33] yields a solution that satisfies the scheduling constraints, all the tasks are guaranteed to meet their deadlines as long as the scheduling policy is strictly enforced. In other words, individual processes should get their slices of CPU at the right time for the right duration. Using the secure timer, the scheduler can ensure that tasks are given the opportunity to run at the right time. Using an accurate accounting of physical time, RT-TEE ensures that processes (malicious or benign) cannot occupy the processor beyond its allocated budget, thereby enforcing the scheduling policy to provide real-time availability.

Robustness against Adaptive Attacker: Since the task schedule is often public, an adversary may attempt to occupy the processor to prevent secure task from running. However, both the runtime budget and the priority are strictly enforced by the scheduler. An adversary may also attempt to block the I/O to cause priority inversion, but that is prevented by bounded priority inversion in the temporal I/O reference monitor.

B. Real-time I/O Availability Protection – R3

I/O Availability – R1: I/O availability in CPS implies access to functional sensors and actuators. The assurance of I/O availability in RT-TEE is based on the use of an I/O reference monitor which mediates over all accesses to the secure peripherals. Based on the I/O access control policy, adversarial interactions with the peripherals are prevented to maintain functional software and hardware states of the peripherals.

RT-TEE supports different types of I/O access policies. The simplest policy we used in the prototype states that only secure tasks can access secure peripherals. For the rest of the discussion, the focus will be on ensuring the complete mediation such that the policy can be enforced. There are two ways that a non-secure OS can access the peripherals, directly manipulating the MMIO registers and compromising the driver software to manipulate the device. Direct access to the MMIO registers is prevented using the hardware primitive for resource isolation [28], [30], [15], e.g., TZASC in ARM [15]. Drivers remain a key challenge, especially when our design excludes the feature-rich instance from the TCB. Debloated drivers replay known interactions. Therefore, an adversary can only influence what template to replay after the I/O reference monitor approves the I/O transaction request. The fixed-length inputs for the templates are also placed on the predetermined position of the data, minimizing the attack surface. For untrusted sandboxed drivers, the main challenge is the mechanism to confine its behavior with the peripheral. This is accomplished by trapping all the hardware interaction instructions to the I/O reference monitor. Based on the I/O security policy, the reference monitor can either allow or deny access. In the case when the peripheral is a bus controller, the I/O reference monitor can also be used to further restrict which device on the bus a process can interact with.

Real-time I/O Availability – R3: For I/O operations to be real-time, each request has to be prioritized according to the process priority. However, some of the low level I/O operations cannot be easily preempted and can lead to priority inversion, blocking the secure tasks. In RT-TEE, priority inversion is minimized and bounded. It is minimized by keeping multiple priority queues of all the I/O requests such that they are processed in the correct order. To bound the priority inversion, the temporal I/O reference monitor inspects each I/O request to determine if it exceeds the volume or size of the legitimate requests. The reference monitor will also use the scheduler to specify a timeout for each I/O operation, thereby bounding the priority inversion time.

Preventing DoS by Shutdown – R4: There are two mechanisms to shutdown or restart the system. One way is to manipulate power management registers. This is prevented using the I/O reference monitor. Another way is suspending the CPU by executing WFI or WFE instructions. However, the non-bypassable secure timer will always wake up the processor at the time scheduling events occur, thwarting the attack.

Preventing Physical Component Damage – R4: An adversary can send unsafe actuation commands to permanently damage physical parts. This can be prevented using the I/O reference monitor via value range specification.

C. TCB Minimization and Platform Security

TCB Analysis – R5: RT-TEE adds several components in the TCB, as listed in Table IV. The hierarchical scheduler includes both the world scheduler, the secure scheduler, and task structure as well context switching code. However, only rate monotonic scheduling algorithm is included in the TCB. The debloated driver pushes almost all the logic to data. Therefore, the replayer requires around 13 lines of well crafted C code to replay from memory, with only four types of recordings. The temporal reference monitor includes basic request data parsing about the I/O request, priority queue, and dispatcher to invoke the corresponding driver entry point. We leverage the existing queue structure and sorting algorithm to minimize TCB. For the spatial reference monitor, our current policy only involves range checking. We push all the policy logic to data and have a few C LOC to check data ranges instead. To support DMA and IRQ on the drivers, we also need to add 88 lines of assembly and C code on top of existing IRQ handlers. Lastly, since our sandbox does not emulate any functionality, it does not permit control flow outside the sandbox except returning to the dispatcher in I/O reference monitor. The only TCB are a

TABLE IV: TCB Sizes of Components in RT-TEE

Component	Sched.	Replayer	Refer. Monitor	S Interrupt Handler	Sandb.
LoC C/Asm	820/100	13/2	37/0	12/76	32/0

few lines of C code to reuse RT-TEE thread management and OP-TEE memory management to setup the execution context.

Defense against Time Warping Attack: Time warp attack exploits the DVFS to present a false sense of execution on secure tasks. It represents a new class of attacks that exploit performance interference. Even though access to DVFS can be prevented using the I/O reference monitor, complete performance isolation on all shared resources remains an open research question. To verify the defense, we launched the attack and were prevented by the I/O reference monitor.

Additional Security Analysis: Analysis on process isolation, side-channel, and sandbox isolation is available in Appendix F.

VIII. Related Work

TEE on Embedded Processors: Existing work on embedded TEE has three categories, novel TEE hardware architectures, hardening TEE environments, and the applications of TEE. 1) Recognizing the limitations on existing commodity TEEs, there are works focusing on novel TEE hardware architectures [17], [28], [30], [18], [52], [53], [35], [54], [55], [27], [40], [36], [41]. Particularly, [28], [30], [27] are designed to provide hardware primitives to bootstrap availability. RT-TEE is complementary in that our work aims to build on top of the HAL provided by these novel hardware designs. RT-TEE addresses key challenges regarding effective use of the primitives in scheduler and driver software system designs. 2) To improve the security of existing TEEs, there are also works hardening the TEE environments [56], [57], [58], [59], [60], [61], [62]. [59] is a closely related concurrent work in which Guo et al. proposed to drive peripherals using recordings. However, our work addresses additional challenges in real-time, availability, bus, and rich features. 3) Recognizing the benefits of TEE, there are works that leverage TEE to harden non-secure environments [20], [63], [64], [65], [66], [67]. RT-TEE is complementary in that it can provide additional protections, such as availability and protected I/O. While there is rich literature on TEE, their focus is mostly on confidentiality and integrity. Recently, there are new novel hardware designs to bootstrap availability; however, the TCB complexity on both the scheduling and I/O subsystems have not been explored in the context of availability. We believe RT-TEE is complementary to existing works, addressing important challenges for the wider adaption of TEE in CPS.

Real-time Scheduling and Security: Real-time scheduling is one of the most active areas in cyber-physical systems. Hierarchical scheduling has been used in hypervisors to enable more optimal utilization between isolated containers [68]. There has also been some work in repurposing TrustZone as a hypervisor [45], [69], [44] and putting the entire safety-critical operating system into a secure world to achieve performance isolation. There are also recent works in applying microkernel [34] and microhypervisor [70] to provide timing control. RT-TEE is complementary to the existing works in that it focuses on solving TCB minimization and secure I/O in addition to real-time scheduling.

CPS and Embedded System Hardening: Existing work on improving CPS resiliency generally falls into two categories, the control approach [71] and the system approach [72], [66], [73], [12], [11], [74], [64], [75], [13], [76]. The I/O reference monitor in RT-TEE is inspired by [13], [76]; however, RT-TEE is a prevention system rather than a detection system and additionally ensures availability. RT-TEE is complementary to these system hardening techniques, providing a previously unexplored security attribute: availability.

IX. Limitations and Discussions

Hardware Abstraction for Resource Isolation: Many modern embedded TEE architectures [17], [28], [30], [18], [35], [54], including the upcoming ARM CCA [54], support full system resource isolation, including both processor and I/O. Though the concrete mechanism and configuration are often SoC-dependent, RT-TEE builds on top of a common abstraction of availability primitives and is capable of supporting these new platforms through individual HAL implementations.

Driver Transformation and Exception Handling: One of the key trade-offs in driver debloating is simplicity for security. In predictable CPS, this is effective, as demonstrated in our drone prototype. However, there are also limitations. First, error handling is removed. In CPS platforms, this is acceptable since most of the error handling functions just probe the sensor again. If necessary, a sensor initialization template can also be added to support recovery. Second, there might be drivers with many features leading to an explosion of templates. A sandboxed driver can be used along with the debloated driver to mitigate the tension in RT-TEE.

X. Conclusion

In this paper we present RT-TEE, a real-time trusted execution environment that aims to address one of the most fundamental needs of safety-critical cyber-physical systems: system availability. RT-TEE builds on top of a minimized abstraction of hardware primitives that enables bootstrapping of availability. It leverages a newly designed two-layer policy-based event-driven hierarchical scheduler to provide real-time scheduling while minimizing the impact on TCB. To enable I/O availability, we proposed and designed an I/O reference monitor that enforces both temporal (real-time) and spatial (security) policies on the hardware/software interactions. To minimize the TCB impact due to I/O drivers, we proposed a combination of sandboxed feature-rich untrusted drivers and minimized functionality trusted debloated drivers. We built prototypes for both ARMv8-A and ARMv8-M platforms and validated the system on a real drone.

Acknowledgment

We thank the anonymous reviewers for their constructive feedback, Yujie Wang and Ruide Zhang for their help in the implementation. This work is supported in part by US National Science Foundation under grants ECCS-1646579, CNS-1837519, CNS-1916926 and CNS-2038995, and by the Fullgraf Foundation.

REFERENCES

[1] "How many millions of lines of code does it take?." https://xenproject.org/developers/teams/xen-hypervisor/. Accessed: 2019-09-01.

[2] G. Doll, "A smart way to drive ecu consolidation," *WindRiver, Tech. Rep.*, 2015.

[3] Intel, "Ecu consolidation reduces vehicle costs, weight, and testing."

[4] I. Astaburuaga *et al.*, "Vulnerability analysis of ar. drone 2.0, an embedded linux system," in *CCWC*, IEEE, 2019.

[5] "Mavlink vulnerability." https://diydrones.com/profiles/blogs/hijacking-quadcopters-with-a-mavlink-exploit. Accessed: 2021-08-07.

[6] "Hackers remotely kill a jeep on the highwaywith me in it." https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/.

[7] E. Deligne, "Ardrone corruption," *Journal in Computer Virology*, 2012.

[8] "Ftp system file access fulnerability on drone." https://nvd.nist.gov/vuln/detail/CVE-2017-3209/. Accessed: 2021-08-07.

[9] "Busybox cve-2017-16544." https://nvd.nist.gov/vuln/detail/CVE-2017-16544. Accessed: 2021-08-07.

[10] Y. D. *et al.*, "Holistic Control-Flow protection on Real-Time embedded systems with kage," in *Security*, USENIX, 2022.

[11] A. A. Clements *et al.*, "ACES: Automatic compartments for embedded systems," in *Security*, USENIX, 2018.

[12] C. H. Kim *et al.*, "Securing real-time microcontroller systems through customized memory view switching.," in *NDSS*, ISOC, 2018.

[13] A. Khan *et al.*, "M2MON: Building an mmio-based security reference monitor for unmanned vehicles," in *Security*, USENIX, 2021.

[14] A. Abbasi *et al.*, "Challenges in designing exploit mitigations for deeply embedded systems," in *EuroS&P*, IEEE, 2019.

[15] "ARM Security Technology, Building a Secure System using TrustZone Technology," apr 2009.

[16] F. McKeen *et al.*, "Innovative instructions and software model for isolated execution.," in *ISCA*, ACM/IEEE, 2013.

[17] V. Costan *et al.*, "Sanctum: Minimal hardware extensions for strong software isolation," in *Security*, USENIX, 2016.

[18] F. Brasser *et al.*, "Tytan: tiny trust anchor for tiny devices," in *DAC*, ACM, 2015.

[19] A. Baumann *et al.*, "Shielding applications from an untrusted cloud with haven," in *OSDI*, USENIX, 2014.

[20] A. M. Azab *et al.*, "Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world," in *CCS*, ACM, 2014.

[21] "Open portable trusted execution environment." https://www.op-tee.org/.

[22] "Trustonic tee." https://www.trustonic.com/technical-articles/what-is-a-trusted-execution-environment-tee/.

[23] "Huawei tee." https://www.commoncriteriaportal.org/files/epfiles/anssi-cible-cc-2020_67en.pdf.

[24] "Nvidia TEE." https://nv-tegra.nvidia.com/gitweb/?p=3rdparty/ote_partner/tlk.git;a=blob_plain;f=documentation/Tegra_BSP_for_Android_TLK_FOSS_Reference.pdf;hb=HEAD.

[25] "Android trusty." https://source.android.com/security/trusty.

[26] "Qualcomm tee." https://www.qualcomm.com/media/documents/files/guard-your-data-with-the-qualcomm-snapdragon-mobile-platform.pdf.

[27] M. Xu *et al.*, "Dominance as a new trusted computing primitive for the internet of things," in *S&P*, IEEE, 2019.

[28] F. Alder *et al.*, "Aion: Enabling open systems through strong availability guarantees for enclaves," in *CCS*, ACM, 2021.

[29] A. Thomas, S. Kaminsky, D. Lee, D. Song, and K. Asanovic, "Ertos: Enclaves in real-time operating systems," *Woodstock*, 2018.

[30] R. J. Masti *et al.*, "Enabling trusted scheduling in embedded systems," in *ACSAC*, IEEE, 2012.

[31] R. Mahfouzi *et al.*, "Butterfly attack: Adversarial manipulation of temporal properties of cyber-physical systems," in *RTSS*, IEEE, 2019.

[32] A. Li *et al.*, "Chronos: Timing interference as a new attack vector on autonomous cyber-physical systems," in *CCS*, ACM, 2021.

[33] S. Baruah *et al.*, "Towards the design of certifiable mixed-criticality systems," in *RTAS*, IEEE, 2010.

[34] A. Lyons *et al.*, "Scheduling-context capabilities: A principled, light-weight operating-system mechanism for managing time," in *EuroSys*, ACM, 2018.

[35] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *EuroSys*, ACM, 2020.

[36] P. Koeberl *et al.*, "Trustlite: A security architecture for tiny embedded devices," in *EuroSys*, ACM, 2014.

[37] C. L. Liu *et al.*, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, 1973.

[38] T. Qin, P. Li, and S. Shen, "Vins-mono: A robust and versatile monocular visual-inertial state estimator," *IEEE Transactions on Robotics*.

[39] D. Kaplan *et al.*, "Amd memory encryption," *White paper*, 2016.

[40] E. Owusu *et al.*, "OASIS: On achieving a sanctuary for integrity and secrecy on untrusted platforms," in *CCS*, ACM, 2013.

[41] G. Dessouky *et al.*, "CHASE: A configurable hardware-assisted security extension for real-time systems," in *ICCAD*, IEEE/ACM, 2019.

[42] J. Noorman *et al.*, "Sancus 2.0: A low-cost security architecture for iot devices," in *TOPS*, ACM, 2017.

[43] L. T. Phan *et al.*, "CARTS: a tool for compositional analysis of real-time systems," in *SIGBED Review*, ACM, 2011.

[44] R. Pan *et al.*, "Predictable virtualization on memory protection unit-based microcontrollers," in *RTAS*, IEEE, 2018.

[45] S. Pinto *et al.*, "LTZVisor: Trustzone is the key," in *ECRTS*, LIPIcs, 2017.

[46] R. I. Davis *et al.*, "Hierarchical fixed priority pre-emptive scheduling," in *RTSS*, IEEE, 2005.

[47] "Trustzone implmentation in parrot bebop drone.." https://wiki.paparazziuav.org/wiki/Bebop.

[48] "Navio2." https://navio2.emlid.com/.

[49] A. Golchin *et al.*, "Boomerang: Real-time i/o meets legacy systems," in *RTAS*, IEEE, 2020.

[50] C. Li *et al.*, "Prioritizing soft real-time network traffic in virtualized hosts based on xen," in *RTAS*, IEEE, 2015.

[51] "ardupilot." https://ardupilot.org.

[52] N. Zhang *et al.*, "Case: Cache-assisted secure execution on arm processors," in *SP*, IEEE, 2016.

[53] F. Brasser *et al.*, "Sanctuary: Arming trustzone with user-space enclaves.," in *NDSS*, 2019.

[54] "Arm cca." https://developer.arm.com/architectures/architecture-security-features/confidential-computing.

[55] J. Noorman *et al.*, "Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base," in *Security*, USENIX, 2013.

[56] S. Wan *et al.*, "RusTEE: Developing memory-safe arm trustzone applications," in *ACSAC*, IEEE, 2020.

[57] Z. Hua *et al.*, "vtz: Virtualizing ARM trustzone," in *Security*, USENIX, 2017.

[58] R. Bahmani *et al.*, "CURE: A security architecture with customizable and resilient enclaves," in *Security*, USENIX, 2021.

[59] L. Guo and F. X. Lin, "Minimal viable io drivers for trustzone," in *EuroSys*, ACM, 2022.

[60] A. Dhar *et al.*, "Protection: Root-of-trust for io in compromised platforms," in *NDSS*, ISOC, 2020.

[61] Z. Zhou *et al.*, "Building verifiable trusted path on commodity x86 computers," in *S&P*, IEEE, 2012.

[62] M. Lentz *et al.*, "Secloak: Arm trustzone-based mobile peripheral control," in *Mobisys, Applications, and Services*, ACM, 2018.

[63] A. M. Azab *et al.*, "SKEE: A lightweight secure kernel-level execution environment for arm.," in *NDSS*, ISOC, 2016.

[64] J. Zhou *et al.*, "Silhouette: Efficient protected shadow stacks for embedded systems," in *Security*, USENIX, 2020.

[65] D. Kwon *et al.*, "uXOM: Efficient execute-only memory on ARM cortex-m," in *Security*, USENIX, 2019.

[66] Z. Sun *et al.*, "OAT: Attesting operation integrity of embedded devices," in *S&P*, IEEE, 2020.

[67] S. Zhao *et al.*, "Sectee: A software-based approach to secure enclave architecture using tee," in *CCS*, ACM, 2019.

[68] S. Xi *et al.*, "RT-Xen: Towards real-time hypervisor scheduling in xen," in *EMSOFT*, ACM, 2011.

[69] S. Pinto *et al.*, "Virtualization on trustzone-enabled microcontrollers? voilà!," in *RTAS*, IEEE, 2019.

[70] D. de Niz *et al.*, "Mixed-trust computing for real-time systems," in *RTCSA*, IEEE, 2019.

[71] R. Quinonez *et al.*, "SAVIOR: Securing autonomous vehicles with robust physical invariants," in *Security*, USENIX, 2020.

[72] T. Abera *et al.*, "C-FLAT: control-flow attestation for embedded systems software," in *CCS*, ACM, 2016.

[73] N. S. Almakhdhub *et al.*, "μRAI: Securing embedded systems with return address integrity," in *NDSS*, ISOC, 2020.

[74] A. A. Clements *et al.*, "Protecting bare-metal embedded systems with privilege overlays," in *S&P*, IEEE, 2017.

[75] R. J. Walls *et al.*, "Control-flow integrity for real-time embedded systems," in *ECRTS*, LIPIcs, 2019.
[76] T. Kim *et al.*, "PASAN: Detecting peripheral access concurrency bugs within bare-metal embedded applications," in *Security*, USENIX, 2021.
[77] "Arm cpu." https://www.arm.com/products/silicon-ip-cpu.
[78] "Secureshield." https://www.synopsys.com/designware-ip/processor-solutions/arc-secureshield-tech.html.
[79] "Arc-sem." https://www.synopsys.com/dw/doc.php/ds/cc/ARC_SEM_110_120D.pdf.
[80] C.-Y. Chen *et al.*, "Indistinguishability prevents scheduler side channels in real-time systems," in *CCS*, ACM, 2021.
[81] "Bcm2835-arm-peripherals." https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf.
[82] A. Vasudevan *et al.*, "Design, implementation and verification of an extensible and modular hypervisor framework," in *S&P*, IEEE, 2013.
[83] A. Tang *et al.*, "CLKSCREW: exposing the perils of security-oblivious energy management," in *Security*, USENIX, 2017.

APPENDIX A
SECURE TIMER POPULARITY

To understand the popularity of hardware secure timer, we surveyed the support of secure timer on two of the most popular embedded processor manufacturers, ARM and Synopsys, who occupy the majority of the embedded processor market shares [77], [78].

TrustZone is the TEE technology by ARM. ARM Cortex-A supports physical secure timer inside the CPU. We surveyed all ARM Cortex-A processes on the ARM product list [77], including A5, A7, A32, A34, A35, A53, A55, A65, A65AE, A72, A73, A75, A76, A76AE, A77, A78, A78AE, A78C, A710, and A510 by crosschecking with CPU reference manuals. All the ARM Cortex-A processors clearly state secure physical timer support in their manuals with a few exceptions. The A710 and A510 are the ARMv9 CPUs with incomplete documentation, and we were not able to confirm support. The earlier generation A32 and A5 CPUs also do not provide any concrete statement about support. We also surveyed the Cortex-M series, M23, M33, M55, listed on the ARM official website[77]. They all have support for secure timer. Furthermore, out of the nine SoCs recommended by ARM, six of them additionally support secure timer peripherals.

For Synopsys, there are two processor series that provide the SecureShield[78] TEE technology, i.e., EM and SEM. All EM processors, including EM4, EM6, EM5D, EM7D, EM9D, EM11D, and EM22FS, have secure timer according to their manuals. Meanwhile, the SEM processors have watch dog timer which can be used to reset the processor or for other functionalities if desired when it expires [79].

APPENDIX B
REALIZING THE AVAILABILITY HARDWARE ABSTRACT LAYER ON ARM PLATFORMS

To take control of the platform, two key ARM features are used, secure timer and fast interrupt request (FIQ). To regain control at a specific time, we rely on secure timer. Within the set of core-specific timers, the majority of the TrustZone-aware SoCs offer a secure timer that can only be set and configured by the secure world. For RPI3 of ARMv8-A, we make use of the *EL1_Secure_Physical_Timer*. For LPC55S69 of ARMv8-M, we make use of the CTimer, which offers different configuration registers for normal world and secure world. To prevent the normal world from interrupting, the secure timer interrupts are routed to FIQ, which has higher priority than regular IRQ. A TrustZone-enabled GIC permits all implemented interrupts to be individually defined as Secure or Non-secure, through the Interrupt Security Registers set (ICDISRn)[45]. On the I/O front, peripherals are assigned to the secure world. On ARMv8-M platforms, the assignment of peripheral devices can be configured using SoC-specific controllers such as the Central Security Unit (CSU) in i.MX6 or TrustZone Protection Controller (TZPC) in Xilinx Zynq-7100 among others [62]. While we were able to use the TrustZone features on NXP development board to protect the peripherals, Raspberry Pi 3 is an evaluation hardware platform that doesn't include secure boot or TZASC. Therefore, the I/O address configuration code is not active in our prototype for this platform. For physical passage of time, both the banked systick timer and the system counter in the generic timer system provide a non-mutable clock. Besides ARM TrustZone, many recently proposed TEE platforms [17], [28], [30], [18], [35] also provide the necessary hardware security primitives for RT-TEE.

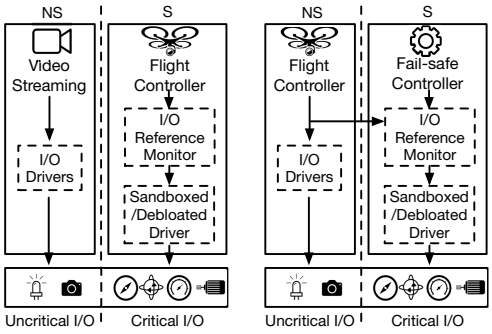


Fig. 12: Flight Controller Protection (left) and Fail Safe Protection (right)

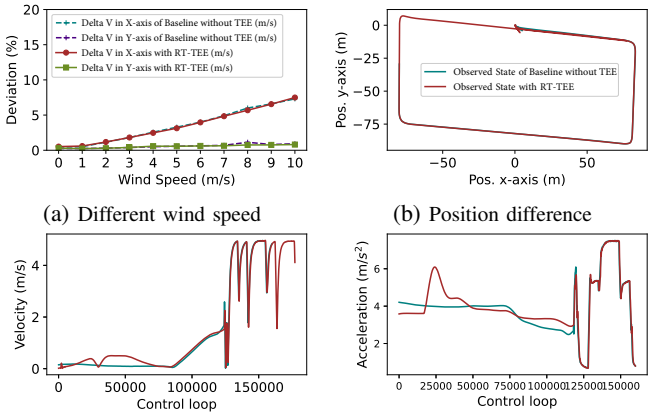
APPENDIX C
CASE STUDY ON AUTONOMOUS DRONE

RT-TEE is designed to support different types of safety-critical real-time task deployments based on security and real-time policies. In the following, we will show how it can be used to protect ArduPilot, an autonomous drone controller.

Case 1 – Flight Controller Protection: As shown in Figure 12, RT-TEE is used to protect the flight controller. Current commercial drones often install various kinds of third-party libraries to support different functionalities, such as video streaming and cartography, which may contain different software vulnerabilities. In systems that can tolerate a large TCB, the whole flight controller can be put into the secure world, protecting the safety-critical components from the rest of the system. From the I/O perspective, the peripherals used by a flight controller have to be assigned to the secure world. Furthermore, each driver needs to be analyzed and transformed before deploying in the secure world.

Case 2 – Fail Safe Protection: As shown in Figure 12, in the second case, the objective is to ensure fail-safe operations, therefore only the fail-safe controller is protected. Fail-safe checkers periodically check system status to make sure the system is in healthy condition. When the system state falls into a danger zone, a simple fail-safe default will be triggered to bring the system into safety. In this case study, we implemented the fail-safe checker along with a default fail-safe recovery landing procedure. All of the sensors and corresponding drivers used by fail-safe controller are assigned and migrated to the secure world. Upon detection of a failure, the self-contained lander will take over the drone and land it.

APPENDIX D
EVALUATION ON DIFFERENT PHYSICAL CONDITIONS

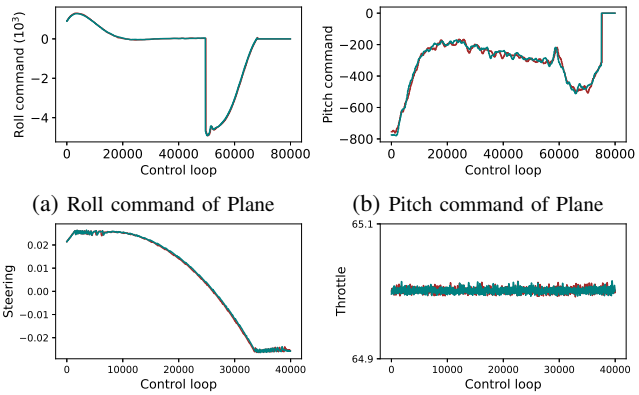


(a) Different wind speed (b) Position difference

(c) Velocity difference (d) Acceleration difference

Fig. 13: Phys. Cond. Impacts on Control with RT-TEE and Baseline w/o TEE

In real-world missions, the UAV often faces different physical environments, such as different wind conditions. In this set of experiments, we vary the wind conditions in simulation from no wind to gentle wind (with a horizontal wind component of 5 m/s or a vertical wind component of 1 m/s), and to strong wind (with a horizontal wind component of 10 m/s or a vertical wind component of 2 m/s) to observe how our system responds. Fig. 13a shows the control state deviations under different wind conditions on the velocity of the copter. From this set of experiments, we can observe that systems with RT-TEE and baseline without TEE share the same control characteristics in different wind conditions. Specifically, both fail to track the reference state when the wind is stronger than 10m/s, this is an inherent limitation of the physical construction of the machinery, such as max throttle. When the wind is weaker, both systems have maintained control well. Under the strong wind, Fig. 13c and Fig. 13d show systems with RT-TEE look less capable of stabilizing attitude than original systems as it deviates a bit more in the first half. Despite greater deviation in acceleration and velocity, the drone remains capable of following its mission trajectory, as shown in Fig. 13b.



(a) Roll command of Plane (b) Pitch command of Plane

(c) Steering command of Rover (d) Throttle command of Rover

Fig. 14: Additional Control Performance with RT-TEE and Baseline w/o TEE

APPENDIX E
ADDITIONAL EVALUATION ON CONTROL PERFORMANCE

To understand the potential impact of RT-TEE on different system controls across different CPS platforms. For plane, the roll command and pitch command are used to measure the control deviation. From Fig. 14a and Fig. 14b, we can observe that the performance with RT-TEE and baseline without TEE is almost identical. The similarity in performance with RT-TEE and baseline without TEE can also be observed in the rover experiment shown in Fig. 14c and Fig. 14d, where steering and throttle are used as the metric.

APPENDIX F
ADDITIONAL SECURITY ANALYSIS

Sandboxed Driver Isolation: A compromised sandboxed driver can attack the system by attempting to read or write memory in the secure environment; however, all reads and writes are instrumented to confine the memory addresses it can visit. An attacker may also write arbitrary content into the I/O registers; this is prevented using the I/O reference monitor. The driver may also attempt to branch outside the sandbox. However, there are no indirect calls in the code the attacker can exploit due to target unrolling. Moreover, the backward edge is protected by the shadow stack.

Protection against Malicious DMA Access: A compromised driver may also attempt to program the peripheral use DMA to tamper with secure environment memory. This can be mitigated either with the I/O reference monitor or hardware (such as system memory management unit (SMMU)).

Process Isolation: RT-TEE relies on the OS in each world to isolate and manage task scheduling individually. However, if needed, the world scheduler can be used to directly schedule tasks in both worlds, treating each task as an individual world at the cost of increased TCB.

Side-Channel and Covert Channel: From the perspective of side-channel and covert channel, the bandwidth on various shared resources (such as cache) is likely similar to or lower than existing TEEs. This is because the untrusted OS can no

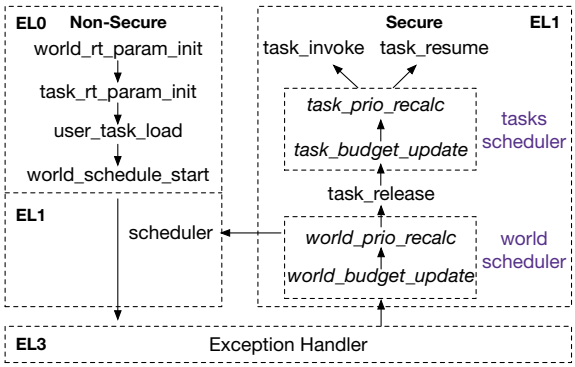


Fig. 15: Scheduler API Execution Flow

longer freely control secure function invocation due to the new scheduling infrastructure. However, real-time scheduling remains predictable [80].

APPENDIX G
ADDITIONAL DETAILS ON SCHEDULING SUBSYSTEM

World Scheduler: The real-time properties, including period, budget, and priority of each world are configured through invoking the RT-TEE API *world_rt_param_init* at initialization, subsequent updates can only be done via secure world, since non-secure world is no longer trustworthy once the system starts execution. Each world has four states, i.e. running, runnable, out of budget, and idle. Running state is set when a world is running. Runnable state is set when a world has budget but isn't running. Out of budget means a world runs out of budget and idle means a world owns budget but has nothing to run. When scheduler runs, it first burns the budget for the running world. World scheduler replenishes budgets and updates deadlines at the start of each period. The states of each world will become runnable after replenishing budgets. According to the priorities, the world scheduler will then choose a world to resume.

Secure OS Scheduler: Since there is no existing scheduling infrastructure in the secure OS of OP-TEE, we have to implement a minimal scheduling infrastructure that supports time accounting, context switches as well as APIs to support different scheduling heuristics. In addition to the infrastructure, we also implemented a RM scheduler for our prototype.

Multicore Support: RT-TEE supports multi-core deployment and makes use of mutex to protect the shared resources. Under current OP-TEE implementation, whenever the mutex blocks, the process will be suspended and control flow will be redirected back to normal world. This is a security problem, we took two steps to mitigate this issue. First, we minimize the amount of shared resources. Second, for the very few status related variables, we make use of spinlock instead.

RT-TEE Scheduling API: RT-TEE is designed to offer an open framework to implement different types of real-time scheduling on a real-world CPS platform. The APIs flow is shown in Fig 15. RT-TEE provided three kinds of API calls, SA (Scheduling Algorithm) functions, SI (Scheduling Infrastructure) functions, and USER functions, to perform world-level scheduling, secure tasks scheduling and RT-TEE initialization. SA APIs are developer customizable, SI APIs are responsible for assisting SA implementation. USER APIs is exposed to users to use RT-TEE.

During the system boot up phase, it is generally assumed that normal world is started with secure boot process and it is trustworthy until there are external communications. At this time, *world_rt_param_init* and *task_rt_param_init* should be invoked to initialize the real-time parameters of the two worlds, and the secure tasks. Alternatively, this can be hard-coded in the firmware and starts automatically with the secure world. Then *user_task_load* will be invoked to load user tasks, and *world_scheduler_start* is invoked to start world scheduler.

At the world scheduling level, *world_budget_update*, and *world_prio_recalc* are the two SA functions responsible for world budgets updating and resuming the next running world. Developers can implement these two APIs to provide different resource reservation algorithms (servers) and scheduling algorithms. *return_to_nw* and *return_to_sw* are two SI APIs invoked by the world scheduler who decides which world to run next. *task_release* will insert periodic released (ready to run) tasks into run queue.

At the secure tasks scheduling level, *task_budget_update*, and *task_prio_recalc* are two SA functions responding for secure tasks budgets updating and next running tasks picking up. Developer can implement these two APIs to provide different tasks-level resource reservation algorithms (servers) and tasks scheduling algorithms. *task_invoke*, and *task_resume* are two SIs assisting the implementation of *task_budget_update* and *task_prio_recalc*.

APPENDIX H
ADDITIONAL DETAILS ON I/O SUBSYSTEM

Sample Template for I2C Bus: Table V shows the template driver for reading pressure from MS5611 barometer. Every command comprises nine bytes. The first byte encodes the command i.e., write(0x0), read(0x1), wait(0x2), and trapping(0x3). The next four bytes encode the value of the MMIO register to be written or the virtual address of memory to store the read result or the worst execution time for the

TABLE V: Part of MS5611 Barometer Driver Template (Get Pressure)

Type	Val/WCET	Reg. Address	Semantic
0x0	0x000000fa	0x14404014	set clock speed of I2C controller
0x0	0x000f003e	0x14404018	set delay of sampling/launch data
0x0	0x00000077	0x1440400c	set barometer device bus address
0x0	0x00000001	0x14404008	set data length to be sent
0x0	0x00008280	0x14404000	start write transfer, set interrupt
0x2	0x00000032	0x14404000	wait for interrupt, WCET is 50us
0x0	0x00000044	0x14404010	write cmd to FIFO to get pressure
...			
0x0	0x00000077	0x1440400c	set barometer device bus address
0x0	0x00000003	0x14404008	set data length to be sent to 3
0x0	0x00008481	0x14404000	start read transfer, set interrupt
0x2	0x00000032	0x14404000	wait for interrupt, WCET is 50us
0x1	0x1015a118	0x14404010	read 1 byte pres. data from FIFO
0x1	0x1015a118	0x14404010	read 1 byte pres. data from FIFO
0x1	0x1015a118	0x14404010	read 1 byte pres. data from FIFO
0x0	0x00000010	0x14404000	clear buffer
0x0	0x00000302	0x14404004	set transfer DONE in state register

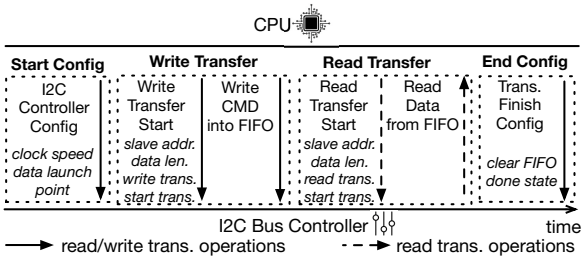


Fig. 16: I2C Bus Driver Template

corresponding I/O operation. The last four bytes are the memory address of the I2C controller MMIO register.

Execution of Bus Transactions: The execution flow of the debloated I2C driver is shown in Fig. 16. One I2C transaction contains four phases. The Start Config phase configures the I2C controller. The Write Transfer phase writes a command for sensor into I2C controller FIFO. The read Transfer phase reads data from the I2C bus controller FIFO into memory. The End Config phase clears the FIFO buffer and sets done state in the control register. Table. V shows the generated driver template for getting pressure operation in MS5611 Barometer. The implementation of the SPI driver is similar to I2C except that the SPI in Navio2 disabled the native CS (Control Register) in the SPI controller. Instead, it leverages GPIO to select devices. According to the device id, the corresponding GPIO pins are set to high. The two native chip select bits in the control status are set to 11 (reserved). After the completion of the transfer, GPIO pins are reset to disable the chip selection.

Peripheral Driver Sandboxing: CFI in RT-TEE sandboxing includes forward edge and backward edge CFI. To enforce forward edge CFI, we avoid function calls using function pointer in the split driver by unrolling all indirect function calls into switch statements including all potential targets in each case. To enforce backward edge CFI, we leverage shadow stack. RT-TEE provides two RTL-level (Register Transfer Language) GCC passes to modify the epilogue and prologue of each function. In the epilogue of the callee function, the return address is saved on the statically allocated shadow stack outside the sandbox. Upon returning to the caller in the epilogue, the previous saved address is used.

RT-TEE puts sandboxed split peripheral drivers in reserved address space of secure OS. Read/write instructions on physical memory are instrumented with the `bfxil` instruction with a reserved general-purpose register to enforce a mask. To sandbox read and write instructions used to access MMIO device address, we instrument each MMIO access read and write instruction with spatial reference monitor trampoline, in which we check the validity of the access destination address and the values to be written with the I/O policy. RT-TEE uses another GCC pass to add the data access sandboxing.

APPENDIX I
DETAILS ON SECURE CLOCK AND POWER

To prevent the non-secure OS from configuring power and clock maliciously, power and clock management configuration registers have to be assigned to the secure environment. The exact interface for power/clock management is hardware implementation-specific. On RPI3 SoC, the Clock Manager (CM) registers (0x7E101000-0x7E101FFF) are responsible for clock control of the peripherals. Power Management (PM) registers (0x7E100000-0x7E000FFF) are used to reset peripherals [81]. On LPC55S69 SoC, the System Controller (SYSCON) (0x40000000-0x40000FFF) is used to select and control clock and reset peripherals. Analog Control Register (ANACTRL) (0x40003000-0x40003103) is used to control the frequency of oscillators. Power Management Controller (PMC) (0x40020000-0x400200CB) is used to control the power of oscillators. It is possible to use SAU to secure these addresses. The prevention of malicious manipulation on processor reset can be commonly achieved by configuring the TrustZone-aware processor to prevent system reset from normal world. On ARM Cortex-A53, Reset Management Registers (RMR) which signal SoC reset controller are only accessible from EL3, effectively preventing reset requests from normal world. On ARM Cortex-M33, the SYSRESETREQS bit of AIRCR (Application Interrupt and Reset Control Register) can be used to disable normal world from making a system reset request.

APPENDIX J
ADDITIONAL RELATED WORK

RT-TEE is different from the existing hypervisor approach [82], [69], [68] regarding the primitive we used to accomplish time isolation. Furthermore, the protection granularity is also different in that hypervisor provides OS-level isolation but RT-TEE provides task-level fine-grained isolation. We are also different from existing TEE solutions[21], [25], [26], [24], [22], [23] in that we focus on system availability in addition to integrity and confidentiality.

Our work also has related work on how change of task timing can destabilize a system. The impact of schedule jitter was investigated in [31]. [32] examines how execution time of different SLAM algorithms impact the physical control. Frequency scaling was used to trigger processor faults in TrustZone in [83]. However, we are the first to examine the impact on timing of processor frequency scaling on cyber-physical systems.