Trusted Execution of Periodic Tasks for Cyber-Physical Systems

Anonymous Author(s)

***Abstract*—Many different systems require the execution of periodic tasks. In particular, systems that communicate with the environment around them usually employ some periodic communication mechanism. This class of systems include, among others, cyber-physical control systems. Embedded and cyber-physical controllers have been proven vulnerable to security attacks of various nature, including attacks that alter sensor data and actuator commands and attacks that disrupt the calculation of the control signals. While attack detection has been widely studied, countermeasures are scarce at best. In this paper, we propose and implement a mechanism to execute a periodic task and its communication interfaces in a trusted execution environment. This allows us, for example, to execute the code of a controller, thus offering higher isolation guarantees. We analyse the overhead of switching between the regular (possibly compromised) execution and the trusted execution and quantify the effect of this defence mechanism on the performance obtained from a cyber-physical control system.**

I. Introduction

Control systems are at the heart of many indispensable societal functions, from energy distribution to pacemakers. Although we rely on these systems, cyber-physical and embedded controllers have been increasingly targeted by security attacks. For instance, modern cars include a number of electronic control units that control functions necessary for the car's operation. Several different attacks have been proposed with researchers managing to gain full control of the car [37], [60]. Physical access to the car has been seen as a prerequisite for an attacker, but remote attack surfaces have also been demonstrated [13].

Attacks on cyber-physical systems may happen because sensor or actuator data is compromised [1], [4], [20], [25], [48], or the data transmission is impaired [29]. While a significant amount of research has been devoted to the (timely) detection of security attacks [11], [28], [38], [49], [52], [54], [58], it is quite hard to write programs that are not impacted by these attacks. Some attempts have been made to either limit the impact of the attack [16], [55], specify estimation or actuation policies that can resist an attack [18], [19], or recover from the attack [14], [66]. However, if we assume that the attacker has taken control of the embedded computing unit, all these techniques become less effective. In fact, in that case, an attacker can simply launch a high priority task that takes over the processor and does not allow the controller code to be executed.

As noted in the recent literature [59], cyber-physical control systems can belong to three broad classes: bare-metal applications, embedded systems that execute control functions alongside a real-time operating system, and general purpose computers that run full-fledged operating systems. We focus here on the second of these classes and target embedded systems that execute a real-time operating system and tasks that perform communication and computation of the commands to be given to a physical plant. This is the setting of many control applications, from automotive to avionics [3], [24]. An example of such systems is the cruise control in a car. The cruise control rely on periodic execution of the task, i.e., the periodic sampling of the sensors, the computation of a new control signal, and finally the actuation of the calculated control signal. In the automotive domain, it is common to use a single electronic control unit to run a single control task in order to ensure the timely execution and communication with the sensing and actuating interfaces. These tasks are thus sensitive to security attacks, as the most delicate car functionality may depend on their execution, e.g., the cruise control system may delay an emergency break due to another car changing lane.

One of the most important countermeasures to combat security attacks is to isolate the program that should be executed securely, ensuring, for example, that no other process running alongside it in the operating system has the ability to tamper with the program's resources. In security terms, this is called software isolation. Several mechanisms to perform software isolation exist, the most prominent one being *trusted execution*. Trusted execution uses special hardware features to restrict access to specific functions, hence ensuring interference freedom.

While trusted execution is a fairly established technique for cryptographic isolation, mainstream applications have not yet exploited the layer of security that trusted execution offers. This is mainly due to the limitations and complexity of programming in the trusted execution environment, which does not offer the full capabilities of a classical programming environment. Specifically, the trusted execution environment is not equipped to run periodic tasks, which are essential to implement embedded systems functionality like image analysis for object recognition, multimedia processing, and control algorithms. Furthermore, the overhead of switching between the normal execution environment and the protected one has been quantified as too taxing for embedded systems applications that need to operate with short deadlines and on a tight schedule. Specifically, in the ARM Cortex-A processors, an overhead of $18.5\,ms$ has been recorded [40], which would generally not be acceptable to run a control application.

In this paper, we implement support for periodic task executions inside a trusted execution environment, using an

ARM Cortex-M processor. In particular, our motivation is ensuring the timely execution of control tasks, with periods in the millisecond range. In our solution, we ensure that the communication interface and a hardware timer used to trigger the control task are only accessible and controlled from the trusted execution environment. We use the interrupt service routine that is triggered by the hardware timer to run the controller code and communicate with the physical plant under control.

To evaluate the proposed implementation, we use a ball and beam process, a classical open-loop unstable control example. We implement an attack against the process and show the difference of running the controller code in the rich execution environment (i.e., the normal environment) or in the trusted environment. For the controller implemented in the different environments, we then (i) perform measurements of the execution time of the controller code under attack and when no attack is present, and (ii) conduct a control-theoretical analysis of the achievable control performance. The analysis shows that the ball and beam controller is not able to regulate the process under attack. Instead, when the controller is implemented in the trusted environment according to our proposed method, even an attacker that can schedule a higher priority task that impair the control task execution is not successful; the controller is able to stabilise the process and deliver good quality of control. We show that the proposed secure controller implementation runs with limited overhead and predictable timing behaviour and periodicity, making this a viable solution to implement in single-node control systems.

The remainder of this paper is organised as follows: Section II introduces a minimal control background, presenting the equations of a controller and also how to evaluate the control performance. We briefly review some control-theoretical material, but focus most of the section on how controllers are usually implemented and on the timing of the controller code. Section III introduces the security background on process isolation and trusted execution environments, focusing on the ARM TrustZone environment that is used for our implementation in the Cortex-M ARM processor. The main challenge behind the implementation of a controller that runs in the secure execution environment is the absence of periodic tasks. In Section IV we present our solution, based on a protected hardware timer and the implementation of the controller in the interrupt service routine. We also present our attack model and the implementation of the attack that we evaluate empirically. Section V presents our experiments, where we analyse the overhead of switching the trusted execution environment in and out and its consequences for control systems, and test the security of the controller implementation against a starvation attack. Finally, in Section VI we review related work and in Section VII we conclude the paper.

II. Control Background

In this section we provide the control-theoretical background necessary to understand the remainder of this paper. In Section II-A we recall some control-theoretical notions, in Section II-B we delve into the implementation of control systems, and in Section II-C we discuss how it is possible to evaluate the performance of a control system from the control perspective. The motivation for the presented background is twofold. On one hand, we delve into the implementation of control tasks, to provide an understanding of what is needed to secure them. On the other hand, we provide the necessary notions for the understanding of the experimental evaluation presented in Section V.

A. Control Theory

The objective of a control system is to regulate a physical process $\mathcal{P}$, usually called *plant*. Generally, the plant dynamics are best described using continuous, nonlinear equations. However, in the vast majority of cases, it is possible to linearise the plant dynamics around an equilibrium point, obtaining a description that uses a linear time-invariant continuous-time system [8].

A plant model can then be written in the form

$$\mathcal{P}: \begin{cases} \dot{x}(t) = A\,x(t) + B\,u(t) + W\,w(t) \\ y(t) = C\,x(t) + D\,u(t), \end{cases} \tag{1}$$

where $x(t)$ is a vector of variables that contains the state of the plant, $u(t)$ is a vector of variables that contains the input to the plant, $y(t)$ is a vector that contains the output of the plant, and $w(t)$ is the process noise. The matrices A, B, W, C and D encode the dynamics of the plant.

Control engineers use the description in Equation (1) to synthesise controllers. The aim of the synthesis is (usually) to ensure that the plant output variables $y(t)$ follow a vector of prescribed reference signals $r(t)$. The synthesis also ensures the satisfaction of some properties: (i) stability, (ii) minimal control effort, (iii) fast rate of convergence to the setpoint, and (iv) fast disturbance rejection. Requiring that the system is stable enforces that none of the signals diverge, and is a necessary condition for the design. Additionally, if all the control requirements are fulfilled satisfactorily, the controller will drive the plant output to the setpoint with minimal strain on the actuators, whilst guaranteeing that the plant reaches the desired reference within a specified time.

The control synthesis process produces the equations of a controller $\mathcal{C}$. Modern controllers are meant to be implemented using digital hardware, and hence the continuous-time signals in the plant needs to be sampled periodically. The control signal is applied at prescribed time instants and kept constant between them. We denote with $k \in \mathbb{N}$ the integer variable that counts the sampling instants, and with v_k the sampled value of $v(t)$, i.e., $v_k = v(k \cdot p)$, where p is the sampling period.

A linear controller can be written as

$$\mathcal{C}: \begin{cases} z_{k+1} = F\,z_k + G_y\,y_k + G_r\,r_k \\ u_k = H\,z_k + K_y\,y_k + K_r\,r_k, \end{cases} \tag{2}$$

where y_k is a vector of measurement of $y(t)$ that is taken at the k-th sampling instant, r_k is a vector that samples the setpoint $r(t)$, u_k is the vector of control signals computed by

the controller at the k-th iteration, and z_k is the controller's internal state. The controller state z_k is a vector that is updated every time the controller code is executed and is used to store relevant information for the next control iteration, e.g., one of its element can accumulate the measured error between the setpoint and the output, aiding the implementation of the integral part of a PID controller [7]. The chosen synthesis process and control architecture (e.g., PID, linear-quadratic regulators, etc) determine the values assigned to the matrices F, G_y, G_r, H, K_y, and K_r in Equation (2).

Combining the equations for plant and controller, the control engineer can write the equations of the closed-loop system. On the closed-loop system, it is possible to verify that (in theory) the control requirements hold, i.e., the closed-loop system is stable, and with minimal effort converges to the setpoint whilst rejecting disturbances. However, this process relies on a few assumptions, among which: the controller execution is supposed to be instantaneous (i.e., no time elapses during the calculation of the control signal u_k), the digital hardware does not introduce any quantisation or distortion, floating-point arithmetic does not impact the precision of the signals. These are some of the differences between the control models and the actual implementations, that require the controller code to be tested with the actual plant for the satisfaction of the control requirements.

B. Control Implementation

The controller is implemented as a periodic task τ_c with period p, characterised by a sequence of jobs $(j_k)_{k\in\mathbb{N}}$. Each job j_k is activated at time a_k (ideally $a_k = k \cdot p$), and performs the computations for the k-th controller step.

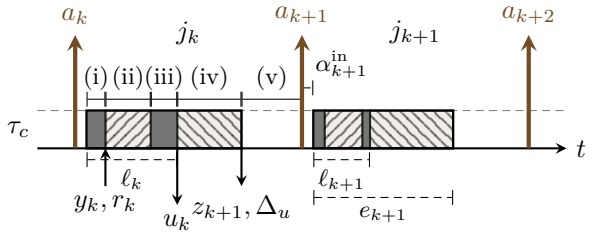

Fig. 1. Visualisation of the execution of the controller, highlighting the job operations.

The controller's job j_k executes the following operations:

(i) it samples the plant output y_k and reads the setpoint r_k,
(ii) it computes the control signal $u_k = \Delta_u + K_y\, y_k + K_r\, r_k$ (where Δ_u is computed in the previous period),
(iii) it sends the control signal u_k to the actuators,
(iv) it updates the state $z_{k+1} = F\, z_k + G_y\, y_k + G_r\, r_k$, and calculates the direct part $\Delta_u = H\, z_{k+1}$ (used in the following period), and
(v) it sleeps for the rest of the period, i.e., idling until a_{k+1}.

From the real-time perspective, the execution time can be split into two parts: (1) steps (i)–(iii) constitute the *input-output* calculation, with latency ℓ_k, (2) step (iv) is the controller state *update* calculation. Intuitively, step (iv) is performed after the actuation step in order to reduce ℓ_k.

We denote with f_k the absolute time at which job j_k completes the execution of step (iv). If the controller completes j_k within its period, then step (v) makes the controller idle for a non-zero time, and sets $a_{k+1} = a_k + p$. In the opposite case, the next control job is immediately activated, $a_{k+1} = f_k$, and the controller period is reset. Figure 1 visualises the phases of the execution. As can be seen in the figure, the execution of the control jobs does not start immediately at their activation. This is due to overhead introduced by the real-time operating system, for example for context switching, and is illustrated in the figure for j_{k+1} and denoted with α^{in}_{k+1}. Finally, we denote the execution time of the controller as $e_k = f_k - a_k - \alpha^{\text{in}}_k$; we illustrate it in the figure in the $(k+1)$-th iteration.

C. Control Evaluation

The control *performance* is usually analysed on the model of the closed-loop system (and on its implementation after data collection). The term performance is often misleading, as the definition of the control performance typically depends on the application and on the application domain. However, the analysis is generally based on the evaluation of a cost function, with lower numbers meaning better quality of control. A very common cost function is the quadratic time-averaged cost

$$J = \frac{1}{t}\int_0^t \mathbb{E}\left[x\,(s)^T\; Q_x\, x\,(s) + u\,(s)^T\; Q_u\, u(s)\right] \mathrm{d}s. \quad (3)$$

Here, J penalises large deviations of the plant states $x\,(t)$ and control signal $u\,(t)$ with respect to the positive semidefinite weight matrices Q_x and Q_u. Since the system is subject to the disturbance signal $w\,(t)$, the evaluation retrieves the cost function's expected value, indicated with the symbol $\mathbb{E}$. Control tools like JitterTime [12] implement standard calculations of the cost function for mixed continuous-discrete linear system models driven by white noise.

III. Trusted Execution

The control process is normally run in a real-time operating system alongside other processes. However, the real-time operating system can be compromised. The controller, then, is subject to the consequences of attacks. Since controllers often implements security-critical primitives, it would be best to execute the control process in isolation with respect to the other processes running on the same hardware.

Process isolation is a commonly used technique to improve the robustness, safety, and security of a computing system. Multiple *processes* are executing simultaneously on the same shared hardware, each of them being confined to its own fraction of the shared resources, like memory and peripherals. Process isolation was originally designed to prevent bugs (introduced by the programmer of one process) from destroying data belonging to another process on a shared mainframe computer. However, over the years, process isolation has been used also to avoid unauthorised accesses to shared resources by a malicious process [22].

To enforce a strong notion of process isolation, the computing system is often split in two separate parts: a *Rich Execution Environment* (REE), and a *Trusted Execution Environment* (TEE). The majority of the system tasks execute in the REE, while the security-critical tasks execute in the TEE. Usually, processes in the REE and TEE communicate using a shared memory area. Paraphrasing the GlobalPlatform definition [21], a TEE is an execution environment that (i) runs alongside but is isolated from the REE, (ii) has the ability of protecting assets from software attacks, and (iii) defines safeguards to data and functions that a program can access.

The technology that is used to implement a TEE can vary – e.g., [15], [32], [42], [46], [68] – and determines the achieved level of security. Hardware features, sometimes called security extensions, are included in modern processors to realise a TEE without introducing the dependency on software components. The details of these hardware features depend on the processor architecture. For example, Intel realised Intel Secure Guard Extensions (SGX) [15], that was found vulnerable to several attacks [43], [67]. AMD introduced the Secure Encrypted Virtualization with Secured Nested Paging (SEV-SNP) [51], also subject to some attacks [9]. Keystone provides a TEE for RISC-V processors [32].

ARM developed TrustZone-A for the Cortex-A line of processors. Recently ARM released TrustZone-M for the Cortex-M processors with the Armv7-M and Armv8-M architectures [42]. Compared to Cortex-A, the Cortex-M provide a lighter context switching. The ARM Cortex-M family dominates the market for resource-constrained devices, commonly used in embedded systems. Except for the Cortex-M0, ARM Cortex-M processors have a Memory Protection Unit (MPU), equivalent to a memory management unit without support for virtual memory.[1]

The MPU enforces the access permission for each memory region. Executable memory is set to *secure*, *non-secure*, or *non-secure callable*, and the processors state depend on where the running code is executed from [63]. Transitions between non-secure and secure memory is performed by branch or jump instructions. To prevent an adversary from corrupting the execution flow inside the TEE (by branching into the middle of a function), secure memory can only be reached by branching from non-secure callable memory. Functions inside non-secure callable memory start with a *secure gateway* instruction, and then branch to secure memory. The state can be temporarily switched from secure to non-secure, by using special *branch and return* instructions. Transitions between states can also be achieved with interrupts. Non executable memory regions can be set to *readable* and/or *writable*, while access can be limited to the TEE only, to the REE only, or to both.[2]

ARM led the development of the Platform Security Architecture (PSA) specification [5], now maintained by an industry consortium. PSA is intended to improve the security of embedded devices used in the Internet-of-Things, with the definition of a threat model, security analysis, and the reference implementation of a security layer, that includes functions and an Application Programming Interface. As an instantiation of PSA, ARM initiated the Open Source project Trusted Firmware-M (TF-M), that uses ARM TrustZone to provide the separation between TEE and REE for the Armv7-M and Armv8-M architectures. TF-M provides a modular set of functions (TF-M Core) that allows developers to call (from the REE) services such as secure storage, interrupt handling, and cryptographic operations (that use securely stored keys and device attestation). Furthermore TF-M allows developers to define and implement their own secure services.

The functions of an REE application that are executing in the TEE are called the Application Root of Trust (RoT). TF-M provides the mechanism (an interface based on inter-process communication) to call Application RoT functions and exchange data between the REE and the TEE. The Application RoT functions can issue calls to the PSA RoT functions, i.e., critical secure services that are provided by the PSA, but not directly exposed to the REE.

The aim of this paper is to periodically execute software in the TEE, to ensure protection and provide security guarantees. Executing a task in the TEE (and specifically in TF-M) implies that the task has: (i) higher priority than all the tasks executed outside TF-M, (ii) protected variables, code, and state, and (iii) access to secure GPIO-pins for transmitting control signals and reading sensor values (without interrupts). However, TF-M does not inherently support periodically executing tasks.

IV. Control with Periodic Trusted Execution

In this section, we discuss the implementation of a secure embedded controller. Specifically, in Section IV-A we specify an attack model and show an example of what could happen if the controller is attacked according to that model. Based on the attack and threat model, we derive the security requirements of our control implementation in Section IV-B and discuss the technical implementation details in Section IV-C.

A. Attack Model

We assume that an attacker is able to compromise and execute arbitrary code in the REE, having full control over it, e.g., being able to schedule any task, for any duration. The attacker can also control all peripherals that are configured as accessible from the REE. However, we assume the TEE to work as intended, and disregard side-channel attacks targeting the TEE. While side-channel attacks against alternative TEE implementations have been extensively researched, side-channel attacks against TrustZone-M remain relatively unexplored. A cache-based side channel has been proposed [23], but no concrete attack utilising the side-channel has been shown (to the best of the authors knowledge). Side-channel attacks against TEE implementations typically aim to leak secrets such as the TEE's cryptographic keys. Instead, this work uses the TEE to isolate computations and GPIO access

[1] https://developer.arm.com/Processors/Cortex-M33

[2] In ARM terms, the TEE is called Secure Processing Environment (SPE) and the REE is called Non-Secure Processing Environment (NSPE). In this paper, we use the more general terms TEE and REE.

from the REE. Therefore, we consider side-channel attacks as out of scope.

An example of a classical attack on control systems is the CPU-starvation attack. The attacker waits for the triggering of the control job j_k, and specifically for the command to read sensor data in the peripheral that connects the plant and the controller. When the command is detected, the attacker starts a high priority (malicious) task τ_m that takes over the computing unit, without doing meaningful work, for a given amount of time. This delays the execution of the control task. Figure 2 shows an example of such an attack happening at control period k. When j_k starts its execution and sends a read command, the higher priority task τ_m takes over, preempting τ_c and delaying the execution of the controller for an amount of time ℓ_k^m.

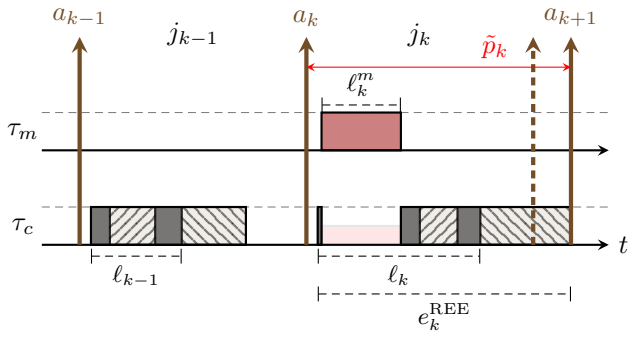

Fig. 2. Malicious task τ_m delaying the execution of the control task τ_c in the k-th period, when τ_c is executed in the REE. The controller execution time e_k^{REE} and period $\tilde{p}_k$ are affected by the execution of τ_m.

As can be seen in the figure, the delay introduced in j_k has two effect: it lengthens the input-output latency ℓ_k of the controller, and it also may push forward the activation a_{k+1} of the next control period. We denote with e_k^{REE} the time elapsed from the beginning of the controller task execution to its completion, and with $\tilde{p}_k$ the actual period of the controller that is experienced by the k-th job. Clearly, with a very long disruption, i.e., if ℓ_k^m is large, the attack can be detected easily. An attacker may want to avoid detection, and hence can set the duration ℓ_k^m to smaller values, but still cause problems to the plant.

Clearly, this is only one of many attacks that an attacker can inject into the system, once the REE is compromised. We introduce it here, because we use it as a test scenario in the experimental evaluation presented in Section V, to show that our secure controller is immune to attacks that target the REE.

B. Security Requirements and Limitations

To secure a control system, it is necessary to devise a solution with the following properties: (i) the adversary can not interfere with the calculations of the control signal, (ii) control signals and measurements must be resistant to tampering and spoofing by the adversary, (iii) the solution must be resilient against CPU-starvation by an adversary, (iv) the adversary has no possibility to delay or re-schedule the execution of the control task, (v) the interface between the secure controller and the REE must be minimal to reduce the attack surface, (vi) only trusted parties are permitted to send messages to the secure controller interface, (vii) the configuration interface is protected against replay of old commands.

Combining the real-time requirements for the implementation of a control system with these security requirements, we identify the following functional requirements of a secure system: (a) the secure controller must preempt the REE to ensure that the control software runs at periodic intervals, (b) the configuration interface must only accept signed configuration messages with a higher sequence number than previously accepted messages, (c) the access to the peripherals used by the secure controller must be limited to the TEE.

We view the act of moving software into a TEE as a building block to make cyber-physical systems secure. Moving the control software into the TEE leaves the configuration interface and the peripheral access functions as an attack vector to the TEE. Software vulnerabilities (e.g., software bugs) have been shown to work against code running inside a TEE [36]. No aspect of the TEE aims at protecting against these types of attacks; it is instead the developer of the TEE software that is responsible for its correctness. Formal verification together with structured and extensive testing could mitigate these software vulnerabilities. Additionally, preventing the REE from accessing the communication interface to the plant is an unfortunate necessity. The used communication protocol lack a mechanism to verify the authenticity of messages. Thus, a compromised REE could send false control signals to the plant, disrupting its operation.

We here provide a secure control implementation that satisfies these functional and security requirements, including the ability of periodically running code in TF-M.

C. Secure Control Implementation

Our solution uses the process separation features and the ability to restrict access to peripherals, introduced in ARM TrustZone and TF-M. Specifically, we rely on the following hardware and software components: (i) ARM TrustZone, (ii) TF-M, (iii) UART (or similar communication interface), configured to be accessible only from the TEE, (iv) a programmable timer that can generate an interrupt, accessible only from the TEE, and (v) an embedded real-time operating system (RTOS). We use UART in our implementation due to the plant communication interface. However, we emphasise that any communication interface, such as CAN bus or similar interfaces [65], can also be used, depending on the specifics of the plant under control. Furthermore, we chose the RTOS Zephyr OS to run in the REE; however, any real-time operating system with support for TF-M could be used.

In order to schedule the execution of our secure controller without any possibility of interference from a potentially compromised real-time operating system, we use the TEE-access only hardware timer to trigger an interrupt, after a given amount of time. Regardless of what the CPU is doing, the timer will continue its count in the background. When the

timer value equals the programmed one, an interrupt is raised. In the interrupt handler, we reset the hardware timer, calculate the control command, send the control command to the plant, and compute the next controller state and the direct term (used in the following control period).

TF-M provide two ways of handling interrupts: First-Level Interrupt Handling (FLIH) and Second-Level Interrupt Handling (SLIH), and does not in principle support the use of interrupt handlers for complex tasks. We use FLIH, where the interrupt is processed immediately, when triggered. The FLIH-handler only exposes limited functionality to the programmer, but by calling the hardware abstraction layer directly, we can access the necessary hardware functions for the implementation. However, it is not possible to, for instance, call the TF-M logging service. Figure 3 summarises our solution.

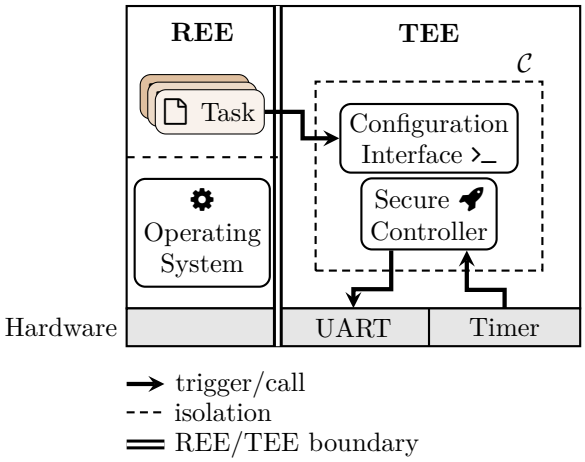

Fig. 3. Illustration of our solution. Tasks from the REE can call the configuration interface of the controller, but cannot impact the secure controller calculations. At the hardware level, UART and Timer can be accessed only from the TEE.

Our implementation is a *non-standard* TF-M Application RoT, where both the communication with the controller and its configuration follow a standard model, but the periodic triggering of the control jobs deviates from the intended TEE usage. An Application RoT has two special types of functions: the `init` function, and secure functions exposed to the REE. We use the `init` function to initialise the UART and the timer. To handle the timer, we use hardware-specific functions. This means that our solution is not cross-platform compatible. However, we restrict ourselves to only using peripherals available on all ARM TrustZone-M compatible platforms. Hence, it is possible, with little to no effort, to port our solution to another hardware platform that satisfies the implementation requirements.

Finally, along with the execution time e_k^{TEE} of j_k in the TEE environment, there is an overhead in switching the TEE in and out, respectively δ_k^{in} and δ_k^{out}. Figure 4 shows the controller task execution in the TEE. When the interrupt handler is triggered (i.e., at the start of δ_k^{in}), the execution of the interrupt handler function is equivalent (from the REE perspective) to executing a non-preemptive task with higher priority. The

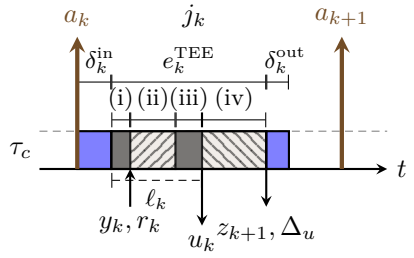

Fig. 4. Visualisation of the execution of the controller in the TEE, highlighting the job operations.

secure controller hence cannot be interrupted by neither tasks that reside in the REE nor lower priority functionality in the TEE (such as TF-M Core functions). The solution targets small embedded control applications, in edge-nodes, with a single plant, and logging functionality (which is not safety-critical) provided in the REE.

V. Experimental Evaluation

In order to evaluate the feasibility of executing the control task in the TEE, we developed a controller for a ball and beam plant $\mathcal{P}$ [61]. In Section V-A we provide some details of the process and controller that we use for our evaluation. In Section V-B we discuss the hardware we use to develop the control system and the measurement strategy that we adopt to obtain reliable measurements of the quantities of interest (i.e., execution times and overheads). We use Section V-C and V-D to provide experimental data of running the control process with and without attacks respectively in the REE and in the TEE, showing that running the controller in the TEE allows us to protect and isolate its execution effectively. Finally, in Section V-E we provide an evaluation of the theoretical control performance achievable by the controller using the delays measured in the real platform.

A. Control Design

The ball and beam system consists of a long beam which is tilted by a servo or electric motor together with a ball rolling back and forth on top of the beam. This plant is a popular textbook example in control theory, and is often used as a representative example because it is open-loop unstable. In fact, even if the beam is restricted to be in the near-horizontal position, without active feedback, the weight of the ball will make it swing to one of the two sides and the ball will roll off the beam. For stabilisation, it is necessary to actively measure the position of the ball and adjust the motor torque to keep the ball in balance. The control task is complex enough due to the ball moving with acceleration that is approximately proportional to the tilt of the beam. In our experiments, the control objective is not restricted to stabilisation, but we want to ensure that the ball follows a prescribed trajectory that is given as setpoint.

A linearised continuous-time model of the plant $\mathcal{P}$ specifies the matrices from Equation (1) as

$$A = \begin{bmatrix} 0 & 0 & 0 \\ -10 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, B = \begin{bmatrix} 4.5 \\ 0 \\ 0 \end{bmatrix}, C = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

and $W = I$, $D = 0$. We have access to only two measurements (the beam angle and the ball position). The exogenous signal $w(t)$ is assumed to be white noise with covariance R. The cost function J penalises the plant state and control signal according to Q_x and Q_u respectively

$$R = \begin{bmatrix} 405 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, Q_x = \begin{bmatrix} 5 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, Q_u = 1.$$

Note that the process noise's covariance is assumed to only enter the first state (the beam angle).

To control $\mathcal{P}$, we design a discrete linear-quadratic-Gaussian (LQG) controller $\mathcal{C}$. The choice of controller is strongly motivated by both the application and the performance evaluation. Specifically, LQG controllers are some of the most classical optimal controllers for linear systems driven by white noise processes. Deriving an LQG controller is equivalent to minimising the cost function J in Equation (3). There exists many strong, classical results on LQG controllers [6] but the core idea is to synthesise an optimal Kalman filter and an optimal state-feedback gain, thus estimating the plant states before calculating a control signal. The derived LQG controller $\mathcal{C}$ for our ball and beam model is

$$F = \begin{bmatrix} 0.709 & 0.054 & 0.041 \\ 0.011 & 0.997 & -0.219 \\ 0.004 & 0.010 & 0.934 \end{bmatrix},$$

$$G_y = \begin{bmatrix} 0.152 & 0.001 \\ -0.104 & 0.216 \\ -0.004 & 0.066 \end{bmatrix}, G_r = \begin{bmatrix} 0 & -0.001 \\ 0 & -0.216 \\ 0 & -0.066 \end{bmatrix},$$

$$H = \begin{bmatrix} -2.433 & 1.201 & 0.562 \end{bmatrix},$$

$$K_y = \begin{bmatrix} -0.672 & 0.368 \end{bmatrix}, K_r = \begin{bmatrix} 0 & -0.368 \end{bmatrix}.$$

where the matrices are given with respect to the controller Equation (2) and the chosen sampling period is $p = 10\,ms$. Note that we take only the second reference value into consideration, since we want the ball to follow the reference position.

B. Hardware Platform and Measurement Strategy

To control the physical plant, we use an nRF5340 DK board.[3] The board has an nRF5340 system-on-a-chip, with two ARM Cortex-M33 cores, respectively called the application core and the network core. The application core supports TrustZone-M and is clocked at 64 MHz. The board also contain 2.4 GHz radio, hardware acceleration for cryptographic functions, UART, and GPIO pins, among other features.

[3]https://www.nordicsemi.com/Products/Development-hardware/nRF5340-DK

We configure the serial communication port UART to read data from and send data to the plant from the TEE. We use blocking UART at 115200 baud, which induces a significant delay when reading and writing data to the plant. The UART is configured to be accessible only from the TEE in order to prevent the REE from writing on, reading from, or blocking the communication between controller and plant.

The communication in and out of the TEE is heavily restricted, to reduce the number of possible attack vectors. Thus, external tools are required to collect data consistently from the system. To avoid impacting the system's behaviour when measuring the overhead from switching in and out the TEE, i.e., δ^{in} and δ^{out}, and the execution time of the controller e^{TEE}, we include, at specific points of the controller execution, code to set GPIO pins that can be read and logged with external hardware. We can then measure the state of the GPIO pins using a logic-analyser. We configure a Raspberry Pi Pico[4] with custom firmware to act as a logic analyser[5] with the open source signal analysis framework sigrok.[6] In this setup, we can record the signals from the GPIO-pins with a sampling rate of $20 \cdot 10^6$ samples per second. We then analyse the recorded data to extract overheads, periods, and execution times per iteration of the controller.

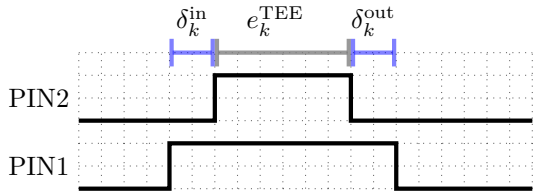

Fig. 5. Measurement setup configuration using GPIO pins `PIN1` and `PIN2`.

Specifically, we use two GPIO pins, `PIN1` and `PIN2`. In Figure 5, the logical states of `PIN1` and `PIN2` are shown during the measurements for the k-th controller iteration. When the hardware clock triggers the interrupt handler we set `PIN1` high; the pin remains high until the interrupt handler exits and is then set to low. Similarly, from the control task we set `PIN2` high when entering the control loop, and low when exiting it. By measuring the time between `PIN1` being set high until `PIN2` is set high, we get the time δ_k^{in}. After `PIN2` goes low, we measure the time until `PIN1` is low, extracting δ_k^{out}. Trivially, the execution time of the controller e_k^{TEE} is given by the time `PIN2` is high. We calculate the controller's actual period $\tilde{p}_k$ by measuring the time elapsed between two consecutively rising edges on `PIN2`. When the controller executes in the REE, we use the same mechanism to measure the execution time e_k and the attack latency ℓ_k^m at every iteration.

C. Attacking the RTOS with τ_c executing in the REE

To start our experimental evaluation, we motivate the need for the proposed solution by showing experiments of an

[4]https://www.raspberrypi.com/products/raspberry-pi-pico/
[5]https://github.com/pico-coder/sigrok-pico
[6]https://sigrok.org

attacker affecting the execution of a controller that is implemented to run in the REE.

We implement the attacker described in Section IV-A in which a malicious task waits for the controller to communicate with the process via UART, and spawns a malicious high priority task that runs for an amount of time ℓ_k^m at the k-th iteration of the controller. The malicious delay that the attacker introduces is selected from the set $\ell^m \in \{0.00\,p, 0.25\,p, 0.50\,p, 0.75\,p, 1.00\,p, 1.25\,p\}$. Each experiment performs 50000 controller iterations. We measure the execution time of the controller and the actual duration of the attack using the GPIO pins. Figure 6 summarises the results, showing that the execution time of the controller, e^{REE} increases with the attack latency ℓ^m.

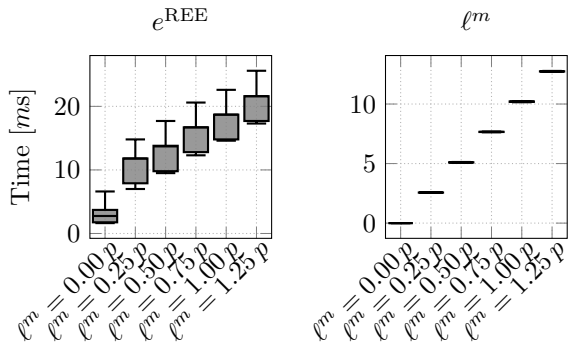

Fig. 6. Measurements of execution time of the controller in the REE, e^{REE}, when there is an ongoing attack, and of the attack latency ℓ^m that delays the execution of the control task.

We remark that the execution time e^{REE} of the controller executing in the REE experiences significant fluctuations even when there is no attack. The experiment with $\ell^m = 0.00\,p$ shows a minimum recorded value $\min e_k = 1632.99\,\mu$s, and a maximum $\max e_k = 6783.8\,\mu$s. The variance in the execution time likely follows from the floating-point arithmetic used to implement the control algorithm. Instead, rewriting the algorithm using proper fixed-point arithmetic, the execution times will be shorter and more consistent. When the attacker starves the computing unit, hence introducing a computational delay, the execution time of the controller increases. Even a delay of $0.25\,p$ causes the execution time of the controller to increase beyond the control period in a significant number of cases ($e^{\text{REE}} > p = 10\,ms$). When the delay is injected by the attacker, the first call to read data from UART is significantly slower than when there is no attack. This likely follows from the higher priority task starving the function processing the UART signal at the driver level.

In the real experiments, the controller is able to stabilise the ball on the beam until $\ell^m = 1.25\,p$; however, the ball position oscillates significantly. When the malicious delay increases, the oscillations are further amplified and it follows quickly that the system becomes unstable, e.g., for $\ell^m = 1.50\,p$ the ball drops off the beam.[7]

D. Attacking the RTOS with τ_c executing in the TEE

We now evaluate the controller execution in the trusted execution environment. We perform three series of tests:

(i) In the first experiment, we execute the controller in the TEE, without any attack present and without additional overhead in the system. We use this experiment to evaluate the overheads δ^{in} and δ^{out} and the execution time of the controller e^{TEE} in nominal conditions.

(ii) In the second experiment, we perform a test equivalent to the test presented in Section V-C. Here, a malicious attacker attempts a starvation attack by spawning a task that is confined to the REE and that is designed to continuously starve the processing unit of *all* its resources (equivalent to $\ell^m = \infty$).

(iii) In the third experiment, we perform a test that assumes a slightly more sophisticated attack, with an attacker trying to starve the TEE of resources by calling Application RoT functions that are exposed to processes in the REE.

Each experiment performs 50000 controller iterations, executing for 500 s in total.

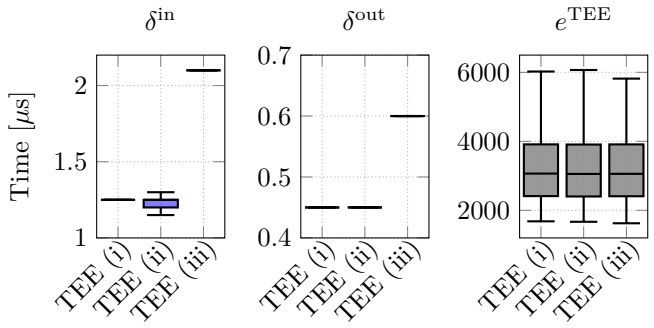

Fig. 7. Measurements of overheads to switch in and out the TEE and of the execution time of the controller in the TEE in the three different scenarios, (i) without any attack, (ii) with an attacker confined to the REE, and (iii) with an attacker that calls other functions exposed by the TEE to starve the TEE of resources.

Figure 7 shows box plots of the measurements of the overheads δ^{in} and δ^{out} to switch in and out the TEE environment, and of the execution time e^{TEE} of the controller. As can be seen, the overheads δ^{in} and δ^{out} are extremely predictable for all the experiments. Despite being able to continuously call exposed functions in the TEE, in Experiment (iii), the attacker still only manages to delay the execution of the controller by roughly $2.2\,\mu$s at most (in comparison to the nominal delay of $1.25\,\mu$s), which is perfectly acceptable for a control application. The overhead to switch out the TEE environment also increases, but is limited to $0.6\,\mu$s, which does not affect the controller. Furthermore, the execution time of the controller

[7]As supplementary material, an anonymised video showing respectively the REE and TEE controlled system under attack can be viewed at: `https://youtu.be/W_EYp9mEdlI`. The video shows the control system executing the experiments performed in Sections V-C and V-D. Upon acceptance, the video will be made publicly available.

is completely unaffected by the presence of the attacks, hence providing a strong confidence in the inability of the attacker to disrupt the correct controller behaviour.

Finally, the execution time including the overheads is well below the period p of the controller, hence the controller periodicity is extremely predictable, i.e., $\tilde{p}^{\text{TEE}} \approx p$. More specifically, we calculate the maximum difference between $p = 10\,ms$ and the actual period measured by executing in the TEE for the three experiments obtaining: for TEE (i) $\max_k |\tilde{p}_k^{\text{TEE}} - p| = 49.5\,\mu s$, for TEE (ii) $\max_k |\tilde{p}_k^{\text{TEE}} - p| = 34.9\,\mu s$, and for TEE (iii) $\max_k |\tilde{p}_k^{\text{TEE}} - p| = 41.2\,\mu s$. This shows that the periodicity of the controller is affected by the time it takes to reset the hardware timer, but not by the ongoing attack.

E. Control Performance Evaluation

Finally, we evaluate the control performance using the cost function specified in Equation (3) and the execution time data obtained in Sections V-C and V-D.

We perform the analysis with JitterTime [12], a MATLAB toolbox for analysing the time-varying state covariance of mixed continuous-discrete linear systems driven by white noise. The toolbox also calculates the accumulated quadratic cost, i.e., the accumulated value from Equation (3). We use JitterTime because it allows us to introduce real data, such as delay and jitter, lost samples or lost controls due to packet loss or execution overruns.

JitterTime REE

```
for e, a in experimental_data:
  execSys!(system, sampler)
  passTime!(system, a)
  execSys!(system, controller)
  passTime!(system, e)
  execSys!(system, actuator)
  if a + e < control_period
    passTime!(system, control_period - e - a)
  end
end
```

Fig. 8. Code sketch for the JitterTime analysis of the control task in the REE.

The analysis in JitterTime for a task executing in the REE is performed as shown in Figure 8. The system is composed of three discrete components: a sampler, a controller, and an actuator. First, the sampler is invoked, then the attacker takes over, introducing a malicious delay with duration a, taken from the logged data. Second, the attacker completes its attack, we can then execute the controller, which has an execution time e, also taken from the logged data. Finally, we invoke the actuator at the completion of the controller execution. If the entire iteration has executed for less than a control period, then we wait until the next activation. Otherwise, we immediately proceed with the next iteration. Similar considerations apply for the execution in the TEE, reading the data from the overhead values δ_k^{in}, δ_k^{out}, and the execution time e_k^{TEE}.

The cost of the closed-loop system is derived in stationarity, i.e., when the system dynamics no longer change. Formally,

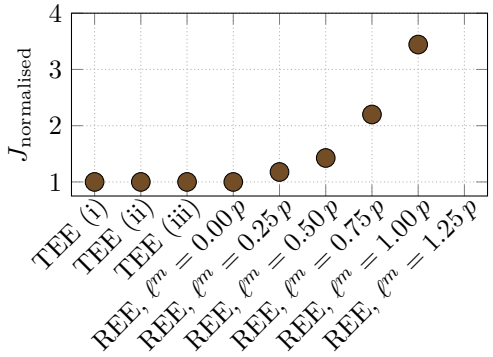

Fig. 9. Normalised cost $J_{\text{normalised}} = J_\infty / \bar{J}$ where $\bar{J}$ is the cost obtained for the execution in the REE without attacks, $\ell_k^m = 0.00\,p$.

the stationary cost, J_∞, is derived by finding the limit of the cost function (3), i.e.,

$$J_\infty = \lim_{t\to\infty} \frac{1}{t} \int_0^t \mathbb{E}\left[x(s)^T\, Q_x\, x(s) + u(s)^T\, Q_u\, u(s)\right] \mathrm{d}s.$$

To ensure an unbiased cost evaluation, we normalise the analytical stationary cost by a baseline cost $\bar{J}$ acquired from the controller executing in REE without any malicious overhead, i.e., $\ell_k^m = 0.00\,p$. Thus, the optimal achievable stationary cost is 1 and a larger value implies a larger control performance degradation. If the system dynamics diverge, i.e., the stationary cost is infinite $J_\infty = \infty$, the system is unstable under the execution pattern specified by the JitterTime model.

Figure 9 shows the normalised cost obtained with the different experiments. The JitterTime analysis confirm the results from the experimental evaluation, namely, that the normalised stationary cost of the TEE executed control system is unaffected by the delay introduced by the attacker. For TEE (i) the normalised cost is 1.00016, i.e., less than 0.02% off the optimal value. Similarly, for TEE (ii) and TEE (iii) the normalised costs are respectively 1.00015 and 1.00026. The slight cost increase follows from the jitter in the input-output latency ℓ_k rather than the controller being executed in the trusted execution environment. Note that despite the attacker trying to completely starve the REE of resources in TEE (ii), the control performance is close to the ideal. In fact, when the controller is executing in the TEE, the only possibility the attacker has to affect the control performance significantly is by completely shutting down the embedded controller. This is however trivial to detect and resolve. This supports the requirement posed on the solution that it must be resistant against CPU-starvation.

Compared to executing the controller in the TEE, the REE executed controller suffers significantly for longer delays introduced by the attacker. Already for short malicious delays, e.g., $\ell^m = 0.50\,p = 5\,ms$, the control performance is severely impaired, with a performance degradation of approximately 43%. With larger delays the cost grows exponentially until it eventually becomes infinite (i.e., the system becomes unstable) for $\ell^m \geq 1.25\,p = 12.5\,ms$. Note that the experiments

performed on the physical ball and beam show that, despite being noticeably oscillatory, the system is stable for $\ell^m \approx 1.25\,p$. The JitterTime model is slightly conservative in its analysis of the control performance; however, it provides a good indication of how the physical system's performance reacts to an attack. This solidifies the evaluation performed on the physical ball and beam plant, described in the previous section.

VI. Related Work

The interest of the real-time community in embedded systems security has increased in recent years. For instance, mitigation strategies against side-channel attacks specifically targeting embedded real-time systems was researched from the perspective of schedule randomisation [57], [64]. In [41] the authors then presented evidence that schedule randomisation can in fact increase the attack surface; thus halting the research into schedule randomisation as a mitigation strategy. For timed network communication, the authors of [39] propose T-Pack: a package for detecting network intrusion by analysing end-to-end delays of the packets.

The control-theoretical approach to security have either been designed for large-scale industrial control systems, e.g., the SCADA architecture [30], or mathematical models of control systems [53]. For embedded systems, [33] introduced a design methodology to provide guarantees on the input-output latency when adding message authentication codes to packets in order to mitigate man-in-the-middle attacks. The authors of [17] introduce a two-way coding approach to mitigate injection attacks. However, we are not aware of any control result that executed an entire controller in a trusted execution environment.

To the best of our knowledge, this is the first paper that investigates the physical application of trusted execution environments for real-time control.

From their introduction and onwards, trusted execution environments have been studies extensively in the scientific literature, mostly investigating their possible use cases to improve the security of computing systems. For example, in control theory the TEE has been used as an intermediary [62] to establish secure communication with industrial control devices. In the work-in-progress paper [26], the authors discuss the possibilities of a restart-based real-time system utilising TEE to achieve secure boot and authentication of all the component images. Similarly, the work-in-progress paper [2] analyses the possible side-channels introduced from real-time schedules running on ARM Cortex-M. The authors analyse side-channels in order to infer when a task is executing in the TEE and predict execution patterns.

The ARM TrustZone security extensions have been studied extensively. In particular this is true for ARM TrustZone-A, where researchers have studied how the TEE behaves [42], [46], [56] and what kind of attacks it is vulnerable to [10], [31], [35], [47]. The application scenarios for ARM TrustZone-A involve systems from data centres to individual machines, but rarely embedded devices (with the exception of smartphones). An example of research conducted on top of the ARM TrustZone-A is IIoTEED [45], which runs a real-time operating system inside a TrustZone-A TEE, called T-RTOS. T-RTOS complements a RTOS that is running on the REE when T-RTOS is idle. Real-time tasks can be run in T-RTOS, obtaining something similar to our periodic execution from the TEE. However, T-RTOS does not interact with the physical world through peripherals. Furthermore, the hardware used in [45] is significantly more powerful than our control node, as IIoTEED runs on a 600 MHz ARM Cortex-A CPU.

The overhead of switching in and out the TEE environment has been evaluated for ARM TrustZone-A [40] on a Raspberry Pi 3b and running an open-source TEE implementation. The authors assume that all the tasks are executed in the REE with secure components of the task execution being executed in the TEE. The Raspberry Pi 3b uses an ARM Cortex-A, which requires the TEE to execute using TF-A. In this case, the overhead to switch in the TEE is measured to be $18.5\,ms$, i.e., almost 4 magnitudes more than our measured values (that are in the μs range). A Raspberry Pi was also used in the evaluation of a TEE-base control solution [50] for a smart connected power inverter for renewable energy. An overhead of $18.5\,ms$ is significantly higher (almost by a factor of 2) then the period of the controller in our experiments, and hence one would conclude that the TEE is not a viable solution to protect a control system.

However, the modern TF-M has minimised the overhead of switching in and out of the TEE, as it targets embedded time-constrained devices. Research into TrustZone-M is mainly focused on security architectures [27], [34] and hypervisors [36]. For example, researchers have been using TrustZone-M in combination with virtualisation to enforce strict timing predictability [44]. On the contrary, we have shown that under the assumption of using TF-M, protecting the controller using a TEE is a reasonable and effective alternative. However, for the solution to be effective, extra care was required in the controller implementation. For example, our controller executes in an interrupt handler, thus removing the delays introduced by a real-time operating system and its scheduler.

VII. Conclusion

Motivated by recent attacks on cyber-physical and embedded control systems, this paper proposes a novel approach to implementing embedded, periodic, real-time tasks in a trusted execution environment (TEE). In the TEE, the program is guaranteed full process isolation, a protected memory region, and isolated communication pins, thus protecting the task from an attacker with full control of the real-time operating system.

We perform an extensive experimental campaign on real hardware, evaluating the performance and timing properties of a periodic control tasks executing in the TEE and comparing the results with a periodic control task executing in the real-time operating system. The timing evaluation is performed for the cases when the system is (i) executing under nominal conditions, (ii) under starvation attack launched from and

confined to the real-time operating system, and (iii) under starvation attack launched from the real-time operating system but using functions exposed by the TEE. In all the experiments, the control performance and timing properties of the control task executing in the TEE are shown to be close to independent of whether the system is under attack or not. On the contrary, for a controller executing in the rich execution environment (REE), we show that an attacker can severly degrade the controller's performance, eventually causing critical failures.

REFERENCES

[1] S. Adepu and A. Mathur. Generalized attacker and attack models for cyber physical systems. In *IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 283–292, 2016.

[2] M. Aguida and M. Hasan. Work-in-progress: Exploring schedule-based side-channels in trustzone-enabled real-time systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2022.

[3] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. I. Davis. An empirical survey-based study into industry practice in real-time systems. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, 2020.

[4] F. Akowuah and F. Kong. Real-time adaptive sensor attack detection in autonomous cyber-physical systems. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 237–250. IEEE, 2021.

[5] ARM. Platform security model 1.1. Technical report, ARM Limited, 2021.

[6] K. Åström. *Introduction to stochastic control theory*, volume 70 of *Mathematics in science and engineering*. Academic Press, United States, 1970.

[7] K. J. Åström and T. Hägglund. *Advanced PID Control*. The Instrumentation, Systems and Automation Society, 2006.

[8] K. J. Åström and R. M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, USA, 2008.

[9] R. Buhren, H.-N. Jacob, T. Krachenfels, and J.-P. Seifert. One glitch to rule them all: Fault injection attacks against amd's secure encrypted virtualization. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2875–2889, 2021.

[10] S. K. Bukasa, R. Lashermes, H. L. Bouder, J.-L. Lanet, and A. Legay. How trustzone could be bypassed: Side-channel attacks on a modern system-on-chip. In *IFIP International Conference on Information Security Theory and Practice*, pages 93–109. Springer, 2017.

[11] A. A. Cárdenas, S. Amin, Z.-S. Lin, Y.-L. Huang, C.-Y. Huang, and S. Sastry. Attacks against process control systems: risk assessment, detection, and response. In *ACM Symposium on Information, Computer and Communication Security*, pages 355–366, 2011.

[12] A. Cervin, P. Pazzaglia, M. Barzegaran, and R. Mahfouzi. Using jittertime to analyze transient performance in adaptive and reconfigurable control systems. In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1025–1032, 2019.

[13] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *20th USENIX Security Symposium (USENIX Security 11)*, 2011.

[14] H. Choi, S. Kate, Y. Aafer, X. Zhang, and D. Xu. Software-based realtime recovery from sensor attacks on robotic vehicles. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 349–364, 2020.

[15] V. Costan and S. Devadas. Intel SGX explained. *Cryptology ePrint Archive*, 2016.

[16] L. F. Cómbita, A. A. Cárdenas, and N. Quijano. Mitigation of sensor attacks on legacy industrial control systems. In *IEEE Colombian Conf. on Automatic Control (CCAC)*, pages 1–6, 2017.

[17] S. Fang, K. H. Johansson, M. Skoglund, H. Sandberg, and H. Ishii. Two-way coding in control systems under injection attacks: From attack detection to attack correction. In *International Conference on Cyber-Physical Systems*, page 141–150, 2019.

[18] H. Fawzi, P. Tabuada, and S. Diggavi. Secure estimation and control for cyber-physical systems under adversarial attacks. *IEEE Transactions on Automatic control*, 59(6):1454–1467, 2014.

[19] J. Giraldo, S. H. Kafash, J. Ruths, and A. A. Cardenas. Daria: Designing actuators to resist arbitrary attacks against cyber-physical systems. In *IEEE European Symposium on Security and Privacy (IEEE Euro S&P)*, 2020.

[20] J. Giraldo, D. Urbina, A. Cardenas, J. Valente, M. Faisal, J. Ruths, N. O. Tippenhauer, H. Sandberg, and R. Candell. A survey of physics-based attack detection in cyber-physical systems. *ACM Computing Surveys*, 51(4):1–36, 2018.

[21] GlobalPlatform. TEE system architecture v1.2. Technical report, GlobalPlatform, Inc., 2018.

[22] D. Gollmann. *Computer Security*. Wiely, 2011.

[23] P. Guo, Y. Yan, C. Zhu, and J. Wang. Research on arm trustzone and understanding the security vulnerability in its cache architecture. In *International Conference on Security, Privacy and Anonymity in Computation, Communication and Storage*, pages 200–213. Springer, 2020.

[24] A. Hamann, D. Dasari, S. Kramer, M. Pressler, F. Wurst, and D. Ziegenbein. Waters industrial challenge 2017. In *Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2017.

[25] N. Hashemi, C. Murguia, and J. Ruths. A comparison of stealthy sensor attacks on control systems. In *American Control Conference (ACC)*, pages 973–979, 2018.

[26] S. Hounsinou, V. Banerjee, C. Peng, M. Hasan, and G. Bloom. Work-in-progress: Enabling secure boot for real-time restart-based cyber-physical systems. In *IEEE Real-Time Systems Symposium*, 2021.

[27] T. Kawada, S. Honda, Y. Matsubara, and H. Takada. Tzmcfi: Rtos-aware control-flow integrity using trustzone for armv8-m. *International Journal of Parallel Programming*, 49(2):216–236, 2021.

[28] A. Khraisat, I. Gondal, P. Vamplew, and J. Kamruzzaman. Survey of intrusion detection systems: techniques, datasets and challenges. *Cybersecurity*, 2(1), 2019.

[29] S. Knorn and A. Teixeira. Effects of jamming attacks on a control system with energy harvesting. *IEEE Control Systems Letters*, 3(4):829–834, 2019.

[30] M. Krotofil and D. Gollmann. Industrial control systems security: What is happening? In *IEEE International Conference on Industrial Informatics*, 2013.

[31] B. Lapid and A. Wool. Cache-attacks on the arm trustzone implementations of aes-256 and aes-256-gcm via gpu-based analysis. In *International Conference on Selected Areas in Cryptography*, pages 235–256. Springer, 2018.

[32] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[33] V. Lesi, I. Jovanov, and M. Pajic. Integrating security in resource-constrained cyber-physical systems. *ACM Transactions on Cyber-Physical Systems*, 4(3), 2020.

[34] C. Lesjak, D. Hein, and J. Winter. Hardware-security technologies for industrial iot: Trustzone and security controller. In *41st Annual Conference of the IEEE Industrial Electronics Society*, pages 002589–002595. IEEE, 2015.

[35] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. {ARMageddon}: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security)*, pages 549–564, 2016.

[36] L. Luo, Y. Zhang, C. Zou, X. Shao, Z. Ling, and X. Fu. On runtime software security of trustzone-m based iot devices. In *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*, pages 1–7, 2020.

[37] C. Miller and C. Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015(S 91), 2015.

[38] R. Mitchell and I.-R. Chen. A survey of intrusion detection techniques for cyber-physical systems. *ACM Computing Surveys*, 46(4):55, 2014.

[39] S. Mittal and F. Mueller. T-pack: Timed network security for real time systems. In *International Symposium on Real-Time Distributed Computing*, pages 20–28, 2021.

[40] A. Mukherjee, T. Mishra, T. Chantem, N. Fisher, and R. Gerdes. Optimized trusted execution for hard real-time applications on cots processors. In *International Conference on Real-Time Networks and Systems*, 2019.

[41] M. Nasri, T. Chantem, G. Bloom, and R. M. Gerdes. On the pitfalls and vulnerabilities of schedule randomization against schedule-based attacks. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2019.

[42] B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin. Trustzone explained: Architectural features and use cases. In *IEEE International Conference on Collaboration and Internet Computing (CIC)*, pages 445–451, 2016.

[43] A. Nilsson, P. N. Bideh, and J. Brorsson. A survey of published attacks on intel SGX. *arXiv preprint arXiv:2006.13598*, 2020.

[44] S. Pinto, H. Araujo, D. Oliveira, J. Martins, and A. Tavares. Virtualization on trustzone-enabled microcontrollers? voilà! In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 293–304, 2019.

[45] S. Pinto, T. Gomes, J. Pereira, J. Cabral, and A. Tavares. IIoTEED: an enhanced, trusted execution environment for industrial iot edge devices. *IEEE Internet Computing*, 21(1):40–47, 2017.

[46] S. Pinto and N. Santos. Demystifying arm trustzone: A comprehensive survey. *ACM Computing Surveys*, 51(6), jan 2019.

[47] K. Ryan. Hardware-backed heist: Extracting ecdsa keys from qualcomm's trustzone. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 181–194, 2019.

[48] G. Sabaliauskaite, G. S. Ng, J. Ruths, and A. Mathur. Comparison of corrupted sensor data detection methods in detecting stealthy attacks on cyber-physical systems. In *IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 235–244, 2017.

[49] H. Sandberg and A. M. H. Teixeira. From control system security indices to attack identifiability. In *Science of Security for Cyber-Physical Systems Workshop*, pages 1–6, 2016.

[50] D. J. Sebastian, U. Agrawal, A. Tamimi, and A. Hahn. Der-tee: Secure distributed energy resource operations through trusted execution environments. *IEEE Internet of Things Journal*, 6(4):6476–6486, 2019.

[51] A. SEV-SNP. Strengthening VM isolation with integrity protection and more. *White Paper, January*, 2020.

[52] M. Striki, K. Manousakis, D. Kindred, D. Sterne, G. Lawler, N. Ivanic, and G. Tran. Quantifying resiliency and detection latency of intrusion detection structures. In *IEEE Military Communications Conference*, pages 1–8, 2009.

[53] A. Teixeira, D. Pérez, H. Sandberg, and K. H. Johansson. Attack models and scenarios for networked control systems. In *Proceedings of the 1st International Conference on High Confidence Networked Systems*, page 55–64, 2012.

[54] A. Teixeira, I. Shames, H. Sandberg, and K. H. Johansson. Revealing stealthy attacks in control systems. In *50th Annual Allerton Conference on Communication, Control, and Computing*, pages 1806–1813, 2012.

[55] D. I. Urbina, J. A. Giraldo, A. A. Cardenas, N. O. Tippenhauer, J. Valente, M. Faisal, J. Ruths, R. Candell, and H. Sandberg. Limiting the impact of stealthy attacks on industrial control systems. In *ACM SIGSAC Conference on Computer and Communication Security*, pages 1092–1105, 2016.

[56] D. C. G. Valadares, N. C. Will, J. Caminha, M. B. Perkusich, A. Perkusich, and K. C. Gorgônio. Systematic literature review on the use of trusted execution environments to protect cloud/fog-based internet of things applications. *IEEE Access*, 9:80953–80969, 2021.

[57] N. Vreman, R. Pates, K. Krüger, G. Fohler, and M. Maggio. Minimizing side-channel attack vulnerability via schedule randomization. In *IEEE 58th Conference on Decision and Control*, 2019.

[58] H. Wang, D. Zhang, and K. G. Shin. Detecting syn flooding attacks. In *IEEE International Conference on Computer Communication*, volume 3, pages 1530–1539, 2002.

[59] B. C. Ward, R. Skowyra, S. Jero, N. Burow, H. Okhravi, H. Shrobe, and R. Khazan. Security considerations for next-generation operating systems for cyber-physical systems, 2019.

[60] R.-P. Weinmann and B. Schmotzle. TBONE – a zero-click exploit for tesla MCUs. Technical report, Comsecuris UG, 2020.

[61] P. E. Wellstead, V. Chrimes, P. R. Fletcher, and A. J. R. R. Moody. The ball and beam control experiment. *The International Journal of Electrical Engineering & Education*, 15(1), 1978.

[62] Q. Ye, H. C. Tan, D. Mashima, B. Chen, and Z. Kalbarczyk. Position paper: On using trusted execution environment to secure cots devices for accessing industrial control systems. Technical report, TechRxiv, 2021.

[63] J. Yiu. Armv8-m architecture technical overview. *ARM white paper*, 2015.

[64] M.-K. Yoon, S. Mohan, C.-Y. Chen, and L. Sha. Taskshuffler: A schedule randomization protocol for obfuscation against timing inference attacks in real-time systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2016.

[65] H. Zeng, P. Joshi, D. Thiele, J. Diemer, P. Axer, R. Ernst, and P. Eles. *Networked Real-Time Embedded Systems*, pages 753–792. Springer Netherlands, 2017.

[66] L. Zhang, P. Lu, F. Kong, X. Chen, O. Sokolsky, and I. Lee. Real-time attack-recovery for cyber-physical systems using linear-quadratic regulator. *ACM Transactions on Embedded Computing Systems (Special issue: EMSOFT)*, 20(5s), sep 2021.

[67] Y. Zhang, M. Zhao, T. Li, and H. Han. Survey of attacks and defenses against SGX. In *2020 IEEE 5th Information Technology and Mechatronics Engineering Conference (ITOEC)*, pages 1492–1496, 2020.

[68] S. Zhao, Q. Zhang, Y. Qin, W. Feng, and D. Feng. Sectee: A software-based approach to secure enclave architecture using TEE. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, page 1723–1740, 2019.