Trusted Execution of Periodic Tasks for Embedded Systems[⋆]

Martin Gunnarsson [*] **Nils Vreman** [**] **Martina Maggio** [***]

[*] *RISE Cybersecurity, RISE Research Institutes of Sweden, Sweden*
[**] *Department of Automatic Control, Lund University, Sweden*
[***] *Department of Computer Science, Saarland University, Germany*

Abstract: Systems that interact with the environment around them generally run some periodic tasks. This class of systems include, among others, embedded control systems. Embedded controllers have been proven vulnerable to various security attacks, including attacks that alter sensor and actuator data and attacks that disrupt the calculation of the control signals. 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 to execute an isolated controller, thus offering higher security guarantees. We analyse the overhead of switching between the regular (possibly compromised) execution environment and the trusted execution environment and quantify the effect of this defence mechanism on the control performance.

Keywords: Security, Embedded computer architectures, Embedded computer control systems and applications

1. 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 controllers have been increasingly targeted by security attacks. Various types of control systems have been targeted by attacks, from critical infrastructure such as power grids to commercial IoT devices (Duo et al., 2022).

Cyber-physical control systems normally belong to one of three broad classes: bare-metal applications, embedded systems executing control functions alongside a real-time operating system, and general purpose computers running full-fledged operating systems. We focus here on the second of these classes and target embedded systems that execute a real-time operating system. This is the setting of many control applications, from automotive to avionics.

Many types of attacks on cyber-physical systems exist, e.g., where sensor or actuator data can be compromised or the transmission channel can be impaired. A remote adversary can through software vulnerabilities gain the ability to execute arbitrary code on a cyber-physical system. The attacker can then utilise different techniques to elevate its privilege and get full control of the device.

One of the most important countermeasures to combat security threats is to isolate the program that should be executed securely. Ensuring that no (potentially malicious) process access another process is called *software isolation.* Several mechanisms to perform software isolation exist, one being *trusted execution.* Trusted execution uses special hardware features to restrict access to specific functions.

While trusted execution is an established technique for process isolation, it has mainly been used to isolate sensitive cryptographic operations and provide secure storage separate from the main program. Additionally, trusted execution environments are not designed to run periodic tasks, which are essential to implement embedded system functionality, e.g., control algorithms.

In this paper, we implement support for light-weight 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. The proposed solution ensures that the communication interfaces and a hardware timer, used to trigger the control task, are only accessible from within the trusted execution environment.

To evaluate the proposed implementation, we implement an attack against a physical ball and beam plant and show the difference of running the controller in the normal execution environment and 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 (ii) conduct a control-theoretical analysis of the achievable control performance. The analysis shows that the proposed secure controller implementation is able to stabilise the plant whilst delivering good quality of control; despite the attacker starving the real-time operating system of all resources. Additionally, 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 authors are members of the ELLIIT Strategic Research Area at Lund University. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement Number 871259 (ADMORPH project) and SSF grant RIT17-0032 (SEC4Factory). This publication reflects only the authors' view and the European Commission is not responsible for any use that may be made of the information it contains.

2. CONTROL BACKGROUND

We consider a controllable and fully observable, *continuous-time*, LTI system

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

where $x(t)$ is the plant state, $u(t)$ the control signal, $y(t)$ the plant output, and $w(t)$ the process noise. The matrices A, B, W, C and D encode the plant dynamics.

The plant is controlled by a *discrete-time* LTI controller

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

where y_k is the k-th sampling instant of $y(t)$, r_k is the setpoint, and z_k is the controller's internal state. The chosen controller synthesis method and 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.

2.1 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.

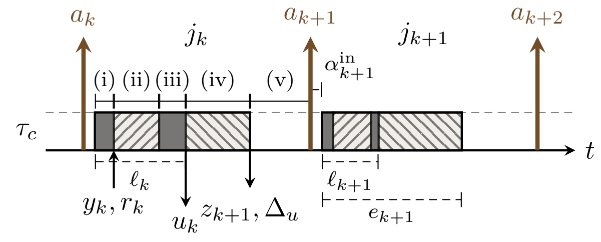

Fig. 1. Visualisation of the execution of the controller, highlighting the job operations.

The controller's job j_k executes the following operations:

(i) it samples the plant output y_k and the setpoint r_k,
(ii) it computes a control value $u_k = \Delta_u + K_y\,y_k + K_r\,r_k$,
(iii) it sends the control signals 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 until the period ends, 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 time at which job j_k completes 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. The execution of the control jobs does not start immediately after activation; this is due to overhead introduced by the real-time operating system, and is denoted with α^{in}_k. Finally, we denote the execution time of the controller as $e_k = f_k - a_k - \alpha^{\text{in}}_k$.

2.2 Control Evaluation

We analyse the performance of the control system using the time-averaged quadratic cost function

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

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. Control tools like JitterTime (Cervin et al., 2019) implement standard calculations of the cost function for mixed continuous-discrete linear system models driven by white noise.

3. TRUSTED EXECUTION IN EMBEDDED SYSTEMS

The control process is generally run in a real-time operating system alongside other processes. However, the real-time operating system can be compromised, thus (possibly) subjecting the controller to attacks. Since controllers often implement security-critical tasks, it would be beneficial to execute the control process in isolation with respect to the other processes running on the 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. To enforce a strong notion of process isolation, the 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 (GlobalPlatform, 2018), 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.

ARM developed TrustZone-A for the Cortex-A line of processors. Recently TrustZone-M was released for the Cortex-M processors with Armv7-M and Armv8-M architectures; Using Cortex-M provide faster context switching in comparison to Cortex-A, according to Ngabonziza et al. (2016). The ARM Cortex-M family dominates the market for resource-constrained embedded systems.

Most 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, where executable memory is set to *secure*, *non-secure*, or *non-secure callable*. 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, secure memory can only be reached by branching from non-secure callable memory. Non executable memory regions can be set to *readable* and/or

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

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 (ARM, 2021). PSA is intended to improve security of embedded devices used in the Internet-of-Things with a threat model, security analysis, and a reference implementation of a security layer. As an instantiation of PSA, ARM initiated Trusted Firmware-M (TF-M) that uses ARM TrustZone-M to provide separation between TEE and REE. TF-M provides a modular set of functions that allows developers to call (from the REE) services such as secure storage, interrupt handling, and cryptographic operations with securely stored keys. Additionally, TF-M allows developers to define and implement their own secure services.

The parts of an REE application that are executed 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 services that are provided by the PSA but not directly exposed to the REE.

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

4. CONTROL WITH PERIODIC TRUSTED EXECUTION

In this section, we discuss the implementation of a secure embedded controller. Specifically, we specify an attack model and show an example of what could happen if the controller is attacked. We then derive the security requirements of our control implementation and discuss the technical implementation details.

This problem has been studied before. Wang et al. (2022) propose RT-TEE, that protect critical tasks and IO inside a TEE from a fully compromised adversary in the REE. Compared to our solution, RT-TEE provide more functionality, allowing the scheduling of multiple tasks inside the TEE. However, for single-task control system our approach provide the same functionality with less complexity.

Pinto et al. (2017) propose a real-time operating system inside a TrustZone-A TEE, called T-RTOS, where real-time tasks can run periodically. Unlike the light-weight solution we propose, T-RTOS does not permit interactions with the physical world through peripherals.

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

4.1 Attack Model

We assume that an attacker is able to compromise and execute arbitrary code in the REE, i.e., having full control over it and can execute any code for any duration. The attacker can also control all peripherals that are configured as accessible from the REE. Side-channel attacks against TrustZone-M aims at leaking secrets from the TEE, such as cryptographic keys. In this work, we use the TEE to isolate computations and IO access from the REE. Therefore, we consider side-channel attacks as out of scope. A classic example of an attack on control systems is the CPU-starvation attack. The attacker waits for the triggering of the control job j_k; specifically, the command that reads sensor data. When the command is detected, the attacker starts a high priority (malicious) task τ_m that takes over the computing unit for a given amount of time. This delays the execution of the control task. Figure 2 shows an example of a starvation attack against the k-th job. When j_k sends a read command, the higher priority task τ_m takes over, preempting τ_c and delaying the controller execution ℓ_k^m time units.

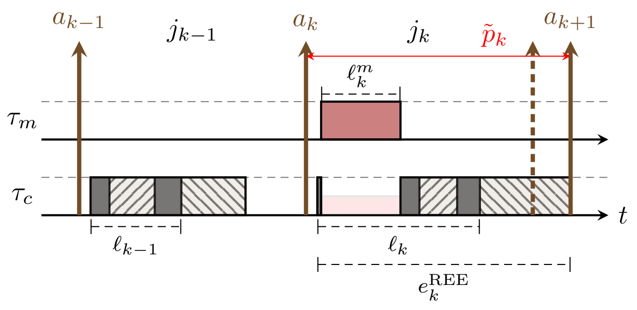

Fig. 2. Malicious task τ_m delaying the execution of the control task τ_c in the k-th period, when τ_c is executed in the REE. The controller execution time e_k^{REE} and period $\tilde{p}_k$ are affected by the execution of τ_m.

As can be seen in Figure 2, the introduced delay has two effects: it lengthens the input-output latency ℓ_k of the controller and it may also 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 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 long disruption, the attack can be detected easily. An attacker want to avoid detection, and hence set the duration ℓ_k^m to smaller values, but still cause problems to the plant.

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 5.

4.2 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 control computations, (ii) control signals and measurements must be resistant to tampering and spoofing by the adversary, (iii) the solution must be resilient against CPU-starvation, (iv) the adversary has no possibility to delay or re-schedule the execution

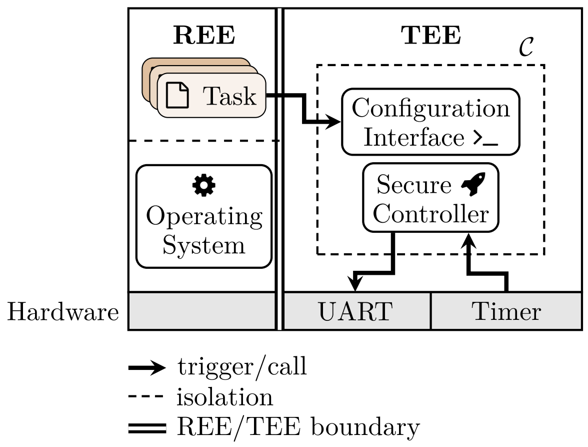

Fig. 3. Tasks from the REE can call the configuration interface of the controller, but cannot impact the secure controller calculations. The UART and Timer can be accessed only from the TEE.

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.

Combining the control system's implementation requirements 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 process executes periodically, (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.

Moving software into a TEE can be one step in securing an embedded system. The TEE does not intrinsically make software more secure. Software vulnerabilities (e.g., bugs) have been shown to work against code running inside a TEE. It is essential to leave a minimal and secure interface between the TEE and REE. Formal verification together with structured testing could mitigate these vulnerabilities.

We here provide a secure control implementation that satisfies these functional and security requirements, including the ability of running code periodically in TF-M. [3]

4.3 Secure Control Implementation

The proposed solution uses the process separation and 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) a communication interface (e.g., UART) configured to be accessible only from the TEE, (iv) a programmable timer that can generate interrupts 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, any communication interface can be used. Additionally, we run

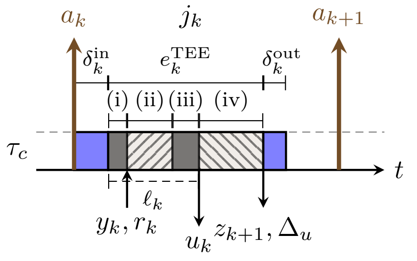

Fig. 4. Visualisation of the execution of the controller in the TEE, highlighting the job operations.

the RTOS Zephyr OS in the REE; however, any RTOS with support for TF-M could be used.

To schedule the execution of our secure controller without any possibility of interference from a (potentially) compromised RTOS, we use the TEE-access only hardware timer to trigger an interrupt after a given amount of time. In the interrupt handler we reset the hardware timer, calculate the control command, send the control command to the plant, and compute the next control state and direct term.

TF-M provide two ways of handling interrupts: First-Level Interrupt Handling (FLIH) and Second-Level Interrupt Handling (SLIH). 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. Figure 3 summarises the proposed implementation.

The implementation is a *non-standard* TF-M Application RoT; the communication with the controller and its configuration follow a standard model, but the periodic triggering of the control jobs deviate from the intended 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 timer and UART. To trigger the timer, we use hardware-specific functions. It is possible to port the solution to other hardware platforms that satisfy 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 control task's 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. Hence, the secure controller cannot be interrupted by either tasks that reside in the REE or lower priority functionality in the TEE (such as TF-M Core functions).

5. EXPERIMENTAL EVALUATION

We will in the following section analyse the proposed implementation scheme on a ball and beam [4]. The objective is to stabilise the plant while ensuring that the ball follows a given trajectory. A continuous-time LTI model of the ball and beam $\mathcal{P}$ is given with respect to Equation (1)

[3] Our code is available as an open-source repository: github.com/Gunzter/Lightweight-TEE-Controller

[4] The ball and beam consists of a beam which is tilted by a servo motor together with a ball rolling back and forth on top of the beam.

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

and $W = I$, $D = 0$. We have access to only two measurements (the beam angle and the ball position), and $w(t)$ 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

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

To control $\mathcal{P}$, we design a discrete linear-quadratic-Gaussian (LQG) controller $\mathcal{C}$

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

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

$$H = [-2.433 \;\; 1.201 \;\; 0.562],$$

$$K_y = [-0.672 \;\; 0.368], \, K_r = [0 \;\; -0.368].$$

where the matrices are given with respect to the controller Equation (2) and the chosen sampling period is $p = 10\,ms$.

5.1 Hardware Platform and Measurement Strategy

To control the physical plant, we use an nRF5340 DK board[5], which accomodates 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.

To communicate with the plant, we use blocking UART at 115200 baud, which induces a significant delay when reading and writing data to the plant.

We want 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}. Thus, we set PIN1 high in the hardware timer interrupt handler until the interrupt handler exits. A second pin (PIN2) is set high when entering the control loop and low when exiting it. The GPIO pins can be read and analysed using a logic-analyser.[6] By analysing the time differences between the two pins during the k-th controller execution, we can extract δ_k^{in}, e_k^{TEE}, and δ_k^{out}.

5.2 Attacking the RTOS with τ_c executing in the REE

We implement the attacker described in Section 4.1, in which a malicious task waits for the controller to communicate with the process via UART, before it spawns a malicious high priority task that runs for an amount of time ℓ_k^m at the k-th controller iteration. 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 5 summarises the results, revealing how the controller's execution time, e^{REE}, increases with the attack latency, ℓ^m.

[5] nordicsemi.com/Products/Development-hardware/nRF5340-DK
[6] github.com/pico-coder/sigrok-pico

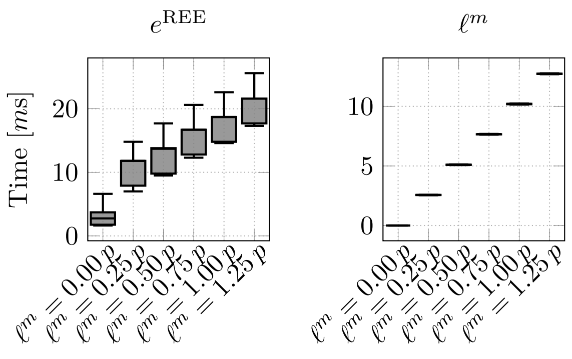

Fig. 5. Measurements of execution time of the controller in the REE, e^{REE}, when there is an ongoing attack delaying the control execution by ℓ^m.

We remark that the execution time e^{REE} of the controller executing in the REE experiences significant fluctuations even when there is no attack, specifically $e_{\min} = 1632.99\,\mu$s and $e_{\max} = 6783.8\,\mu$s. The variance in the execution time likely follows from the floating-point arithmetic used to implement the control algorithm. When the attacker starves the computing unit, even a delay of $0.25\,p$ causes the maxium execution time of the controller to increase beyond the control period ($e^{\text{REE}} > p = 10\,ms$). 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[7].

5.3 Attacking the RTOS with τ_c executing in the TEE

We now evaluate the controller execution in the trusted execution environment, using three series of tests:

(i) We execute the controller in the TEE without any attack present and without additional overhead.
(ii) We perform a test equivalent to the test presented in Section 5.2, i.e., a malicious attacker attempts a starvation attack by spawning a task that is confined to the REE, with the intention to starve the processing unit of *all* resources (equivalent to $\ell^m = \infty$).
(iii) 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.

Figure 6 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 consistent 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

[7] A video showing respectively the REE and TEE controlled system under attack can be viewed at: youtu.be/miDM5NhKvRI. The video shows the experiments performed in Sections 5.2 and 5.3.

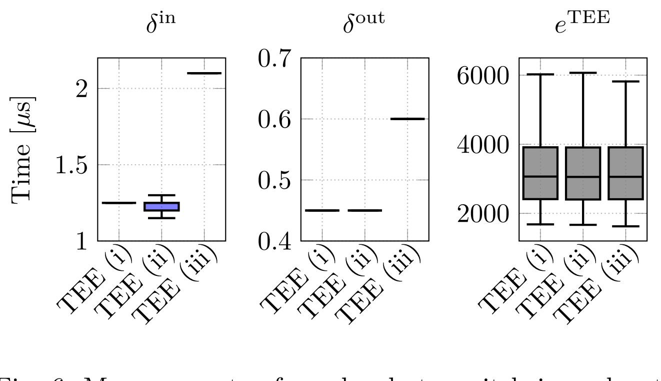

Fig. 6. Measurements of overheads to switch in and out the TEE and the execution time of the controller in the TEE in the scenarios described above.

the controller by roughly $2.2\,\mu$s at most (in comparison to the nominal delay of $1.25\,\mu$s). The overhead to switch out the TEE environment also increases, but is limited to $0.6\,\mu$s. Furthermore, the execution time of the controller is completely unaffected by the presence of the attacks, providing a strong confidence in the approach's ability to prevent attacks on the controller behaviour.

5.4 *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 5.2 and 5.3. We perform the analysis using JitterTime (Cervin et al., 2019) – a toolbox for analysing mixed continuous-discrete linear systems. JitterTime allows us to introduce real data, such as delay, jitter, and execution overruns while calculating the accumulated cost from Equation (3).

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

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

To ensure an unbiased 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.

Figure 7 shows the normalised cost obtained with the data from the different experiments. 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. On the other hand, 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. The JitterTime analysis confirms that the normalised stationary cost of the TEE executed control system is unaffected by the delay introduced by the attacker.

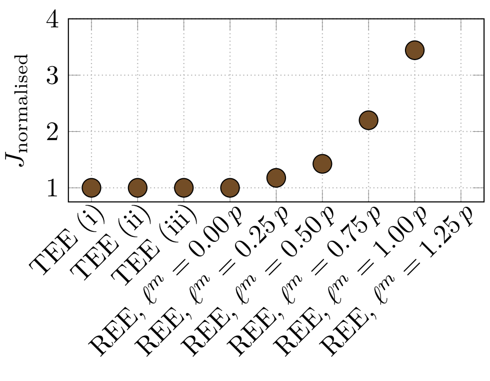

Fig. 7. Normalised cost $J_{\text{normalised}} = J_\infty/\bar{J}$ where $\bar{J}$ is the cost obtained for the execution in the REE without attacks, $\ell_k^m = 0.00\,p$.

6. CONCLUSION

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 process isolation, a protected memory region, and isolated communication pins, thus protecting the task from an attacker with control of the real-time operating system. We perform an extensive experimental campaign on real hardware, evaluating performance and timing properties of a periodic control task implemented in the TEE. In all experiments, the performance and timing properties are shown to be close to independent of whether the system is under attack or not. Future work involve implementing the proposed solution for additional real-time operating systems and communication protocols, thus making the framework suitable for a wider class of embedded applications, e.g., automotive.

REFERENCES

ARM (2021). Platform security model 1.1. Technical report, ARM Limited.

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

Duo, W., Zhou, M., and Abusorrah, A. (2022). A survey of cyber attacks on cyber physical systems: Recent advances and challenges. *IEEE/CAA Journal of Automatica Sinica*, 9(5), 784–800.

GlobalPlatform (2018). TEE system architecture v1.2. Technical report, GlobalPlatform, Inc.

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

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

Wang, J., Li, A., Li, H., Lu, C., and Zhang, N. (2022). RT-TEE: Real-time system availability for cyber-physical systems using ARM TrustZone. In *IEEE Symposium on Security and Privacy.* IEEE Computer Society.