Preserving Physical Safety Under Cyber Attacks

Fardin Abdi, Chien-Ying Chen, Monowar Hasan, Songran Liu, Sibin Mohan and Marco Caccamo

Abstract—Physical plants that form the core of the Cyber-Physical Systems (CPS) often have stringent safety requirements and, recent attacks have shown that cyber intrusions can cause damage to these plant. In this paper, we demonstrate how to ensure the *safety* of the physical plant even when the platform is compromised. We leverage the fact that due to physical inertia, an adversary cannot destabilize the plant (even with complete control over the software) instantaneously. In fact, it often takes finite (even considerable time). This work provides the analytical framework that utilizes this property to compute safe operational windows in run-time during which the safety of the plant is guaranteed. To ensure the correctness of the computations in runtime, we discuss two approaches to ensure the integrity of these computations in an untrusted environment; *(i)* full platform-wide restarts coupled with a root-of-trust timer and *(ii)* utilizing Trusted Execution Environment (TEE) features available in hardware. We demonstrate our approach using two realistic systems – a 3 degree-of-freedom helicopter and a simulated warehouse temperature management unit and show that our system is robust against multiple emulated attacks – essentially the attackers are not able to compromise the safety of the CPS.

Index Terms—Cyber-Physical Systems, safety-critical systems, security, real-time systems, embedded systems.

1 Introduction

Some of the recent attacks on cyber-physical systems (CPS) are focused on causing physical damage to the plants. Such intruders make their way into the system using cyber exploits but then initiate actions that can destabilize and even damage the underlying (physical) systems. Examples of such attacks on medical pacemakers [22], or vehicular controllers [25] exist in the literature. Any damage to such physical systems can be catastrophic – to the systems, the environment or even humans. The drive towards remote monitoring/control (often via the Internet) only exacerbates the safety-related security problems in such devices.

When it comes to security, many techniques focus on preventing the software platform from being compromised or detecting the malicious behavior as soon as possible and taking recovery actions. Unfortunately, there are always unforeseen vulnerabilities that enable intruders to bypass the security mechanisms and gain administrative access to the controllers. Once an attacker gains such access, all bets are off with regards to the safety of the physical subsystem. For instance, the control program can be prevented from running, either entirely or even in a timely manner, sensor readings can be blocked or tampered with, and false values forwarded to the control program and similarly actuation commands going out to the plants can be intercepted/tampered with, system state data can be manipulated, *etc.* These actions, either individually or in conjunction with each other, can result in significant damage to the plant(s). At the very least, they will significantly hamper the operation of the system and prevent it from making progress towards its intended task.

In this paper, we *develop analytical methods that can formally guarantee the baseline safety of the physical plant even when the controller unit's software has been entirely compromised*. The main idea of our paper is to carry out consecutive evaluations of physical safety conditions, inside secure execution intervals, separated in time such that an attacker with full control will not have enough time to destabilize or crash the physical plant in between two consecutive intervals. We refer to these intervals by Secure Execution Interval (SEI). In this paper, the time between consecutive SEIs is dynamically calculated in real time, based on the mathematical model of the physical plant and its current state. The key to providing such formal guarantees is to make sure that each SEI takes places before an attacker can cause any physical damage.

To further clarify the approach, consider a simplified drone example. The base-line safety for a drone is to not crash into the ground. Using a mathematical model of the drone, we demonstrate, in Section 4.2, how to calculate the shortest time that an adversary with full control over all the actuators would need to take the drone into zero altitudes (an unsafe state) from its current state (*i.e.*, current velocity and height). The key is, once inside the SEI, to schedule the starting point of the upcoming SEI *before* the shortest possible time to reach the ground. During the SEI, depending on whether the drone was compromised or not, it will be either stabilized and recovered or, it will be allowed to resume its normal operation. With this design in place, despite a potentially compromised control software, the drone will remain above the ground (safe).

Providing formal safety guarantees, even for the simple example above is non-trivial and challenging. As an example, an approach is needed to compute the shortest time to reach the ground in run-time. Each SEI must be scheduled to take place at a state that not only is safe (before hitting the ground), but also the controller can still stabilize the drone from that velocity and altitude, before it hits the ground, considering the limits of drone motors. Mechanisms are needed to prevent attackers from interfering with the SEIs in any way possible. In this paper, we address all the challenges required to provide safety.

Significant parts of this work have been published earlier in the proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems (IC-CPS'18) with DOI: 10.1109/ICCPS.2018.00010 [3].

One of the primary technical necessities for the proposed design is a trusted execution environment where the integrity of the executed code can be trusted. In this paper, we utilize two different approaches to achieve this goal; *(i) restart-based implementation* which utilizes full system restarts and software reloads *(ii) TEE-based implementation* which utilizes Trusted Execution Environment (TEE) such as ARM TrustZone [43] or Intel's Trusted Execution Technology (TXT) [24] that are available in some hardware platforms.

Under the restart-based implementation, control platform is restarted in each cycle, and the uncompromised image of the controller software is reloaded from read-only storage. Restarting the platform enables us to *(i)* eliminate all the possible transformations carried out by the adversary during previous execution cycle[1] and also *(ii)* provides a window for trusted computation in an untrusted environment which we use to compute the next SEI triggering time (Section 4.1). This design utilizes an external HW timer to trigger the restart at the scheduled times. This simple design prevents the adversary from interfering with the scheduled restarting event.

Another alternative approach that is introduced in this paper to enable the SEIs is to the use TEE features that are available in HW platforms. In particular, we use ARM TrustZone [43] and LTZVisor [28] which is a hypervisor based on TrustZone (Section 5.1). The TEE-assisted implementation does not require the platform to be restarted in every SEI cycle. Thus, there is no restarting overhead, and additionally, the controller state is not lost in every SEI cycle. This design can significantly improve the applicability of our method to physical plants with faster dynamics. As we have shown in the evaluation section, the maneuverability region of the 3DOF plant is increased by 234 percent when the controller is implemented by the TEE-based method.

For some CPS applications, one of the above implementation options might be a more suitable choice than the other one. If the physical plant has high-speed dynamics – relative to the restart time of the platform – or if the past state of the controller is necessary to carry out the mission – *e.g.,* authentication with ground control – the TEE-based option the reasonable choice. On the other hand, restart-based implementation is feasible for low-cost micro-controllers whereas platforms equipped with TEE are generally more expensive. Furthermore, many of the CPS applications have physical plants with slow physical dynamics – compared to the restart time of their embedded platform – and the restart-based implementation will perform just as good as the TEE-based implementation (as we will show in Section 6.4). For such cases, restart-based implementation is a better choice, and TEE-assisted implementation might only unnecessarily increase the cost and complexity of the system.

In summary, the contributions of the work are:

1) We introduce a design method for embedded control platforms with *formal guarantees on the base-line safety of the physical subsystem* when the software is under attack.
2) We propose a restart-based design implementation that enables trusted computation in an untrusted environment using platform restarts and common-off-the-shelf (COTS) components, without requiring chip customizations or specific hardware features.
3) We propose an alternative design implementation using TEE features that eliminates the restarting overhead and enables the core safety-guarantees to be provided on more challenging physical plants.
4) We have implemented and tested our approach against attacks through a prototype implementation for a realistic physical plant and a hardware-in-the-loop simulation. We compare both design implementation options and illustrate their use cases.

Significant parts of this work have been published in an earlier conference paper [3]. The critical improvement upon earlier results here is the use of TrustZone to implement secure execution intervals that eliminates the overhead of system-reboots and improves the maneuverable are of the 3DoF helicopter by 234 percent. We have also performed all the experiments to evaluate the new aspects of the approach.

2 Applications, Threats and Adversaries

This paper focuses on end-point devices that control and drive a safety-critical physical plant *i.e.,* the plant has safety conditions that need to be respected at all times. Components such as sensing nodes that do not directly control a physical plant are not in the scope of this paper. Safety requirements of the plant are defined as an admissible region in a connected subset of the state space. If the physical plant reaches the states outside of the admissible region, it could damage itself as well as the surrounding environment. Thus, to preserve the physical safety, the plant must only operate within the admissible region.

2.1 Adversary and Threat Model

Embedded controllers of CPS face threats in various forms depending on the system and the goals of the attacker. The particular attacks that we aim to thwart in this paper are those that target damaging the physical plant. In this paper, we assume attackers require an *external interface* such as the network, the serial port or the debugging interface to intrude into the platform. We assume that the attackers do not have physical access to the platform. Once a system is breached, we assume the attacker has full control (root access) over the software (non-secure world), actuators, and peripherals.

The following assumptions are made about the platform and the adversary's capabilities:

i) *Integrity of original software image:* We assume that the original images of the system software *i.e.,* real-time operating system (RTOS), control applications, and other components are not malicious. These components, however, may contain security vulnerabilities that could be exploited to initiate attacks.

ii) *Read-only storage for the original software image*: We assume that the original trusted image of the system software is stored on a read-only memory unit (*e.g.,*

1. It is possible that the adversary launches a new instance of the attack after a restart. Yet, the plant is protected against each attack instance and *malicious states are not carried across restarts*. As a result, the proposed approach is able to prevent the attacker from damaging the system every time and guarantees safety of the entire system.

E^2PROM). This content is not modifiable at runtime by anyone including adversary. Updating this image requires physical access and is completed off-line when the system is not operating[2].

iii) *Trusted Execution Environment (TEE)*: Hardware-assisted TEEs such as TrustZone partition the platform into a secure world and a non-secure world. Resources (*i.e.,* code and data) in the secure world are isolated from the non-secure world and are only accessible by the software running in the secure world. A compromise in the non-secure world may not affect the execution and data in the secure world. In this paper, we assume that the software in the secure world is trusted from the beginning and may not be compromised (in our design, the secure world only interacts with sensors and actuators and does not have an exposed interface that can be a point of exploitation).

iv) Immediately after a reboot, as long as the external interfaces of the device (*i.e.,* network, debugging interface) remain disabled[3], software running on the platform is assumed to be uncorrupted.

v) *Integrity of Root of Trust (RoT)*: RoT – which is only necessary for the restart-based implementation – is an isolated hardware timer responsible for issuing the restart signal at designated times. As shown in Section 4.1, it is designed to be programmable *only* once in each execution cycle and *only* during an interval that we call the SEI.

Additionally, we assume that the system is not susceptible to external sensor spoofing or jamming attacks (*e.g.,* broadcasting incorrect GPS signals, electromagnetic interference on sensors *etc.*). An attacker may, however, spoof the sensor readings within the OS or applications. Our approach does not protect from data leak related attacks such as those which aim to steal secrets, monitor the activities, or violate the privacy. Our design does not protect from network attacks such as man-in-the-middle or denial-of-service attacks that restrict the network access. An attacker may enter the system via any external interface (*e.g.,* a telemetry channel, a network interface) and use known vulnerabilities such as buffer overflow or code injection to manipulate the system. However, as we show, the physical plant remains safe during such attacks.

3 Background

In this section, we provide a brief background on Safety Controller and Real-Time reachability. We will utilize these tools in the rest of this paper. Before going into their details, we first present some useful definitions.

Definition 1. *Admissible and Inadmissible States: States that do not violate any of the operational constraints of the physical plant are referred to as admissible states and denoted by $\mathcal{S}$. Likewise, those states that do violate the constraints are referred to as inadmissible states and denoted by $\mathcal{S}'$.*

Definition 2. *Recoverable states: are defined with regards to a given Safety Controller (SC) and denoted by $\mathcal{R}$. $\mathcal{R}$ is a subset of $\mathcal{S}$ such that if the given SC starts controlling the plant from the state $x \in \mathcal{R}$, all future states will remain admissible.*

In other words, the physical plant is considered momentarily safe when the state is in $\mathcal{S}$. Moreover, SC can stabilize the physical plant, if its state is in $\mathcal{R}$. Operational limits and safety constraints of the physical system dictate what $\mathcal{S}$ is and it is outside of our control. However, $\mathcal{R}$ is determined by the design of the safety controller. Ideally, we would want a SC that can stabilize the system from all the admissible states $\mathcal{S}$. However, it is not usually possible.

In the following, one possible way to design a safety controller is discussed. This method is based on solving linear matrix inequalities and has been used in the design of systems as complicated as automated landing maneuvers for an F-16 [33].

3.1 Safety Controller

According to this design approach [33], [34], SC is designed by approximating the system with linear dynamics in the form of $\dot{x} = Ax + Bu$, for state vector x and input vector u. In addition, *the safety constraints of the physical system are expressed as linear constraints* in the form of $H \cdot x \leq h$ where H and h are constant matrix and vector. Consequently, the set of admissible states are $\mathcal{S} = \{x : H \cdot x \leq h\}$. The choice of linear constraints to represent $\mathcal{S}$ is based on the Simplex Architecture and many of the following works [6], [10], [11], [33], [34], [35].

In this approach, the operational safety constraints, as well as actuator saturation limits, are expressed as linear constraints in an LMI. These constraints, along with linear dynamics for the system are input into a convex optimization problem that produce both linear proportional controller gains K as well as a positive-definite matrix P. The resulting linear-state feedback controller, $u = Kx$, yields closed-loop dynamics in the form of $\dot{x} = (A+BK)x$. Given a state x, when the input $u = Kx$ is used, the P matrix defines a Lyapunov potential function, $V = x^T Px$, such that: $V > 0$, $\dot{V} < 0$, and $V = 0$ if and only if $x = 0$, thus guaranteeing stability of the linear system using Lyapunov's direct or indirect methods. Furthermore, the matrix P is constructed by the method such that it defines an ellipsoid in the state space where all the constraints are satisfied when $x^T Px < 1$. Since the states where saturation occurs were provided as input constraints to the method, this means that states inside the ellipsoid result in control commands that are not beyond the actuator limits (where saturation would occur). States that are in $\mathcal{S}$ but not in $x^T Px < 1$ ellipsoid, may result in control commands that are beyond the actuator limits. It follows that the states which satisfy $x^T Px < 1$ are a subset of the safety region. Because the potential function is strictly decreasing over time, any trajectory starting inside the region $x^T Px < 1$ will remain there for an infinite time window. As a result, no inadmissible states will be reached. Hence, the linear-state feedback controller $u = Kx$ is the SC and $\mathcal{R} = \{x :$

2. This is common for many safety-critical IoT systems such as medical devices and some components in automotive systems – to prevent from runtime malfunctioning due to unwanted firmware corruption at the time of update and well as to prevent the adversary from tampering with the system's image remotely)

3. This is achieved by not initiating a socket connection, not reading/writing from/to any of the ports and not performing any of the hand shaking steps.

$x^T Px < 1\}$ is the recoverable region. Designing SC in such a way ensures that the physical system would always remain safe [35].

Note: Safety Controller is only capable of keeping plant safe and does not push it towards its goal/mission. A meaningful system, therefore, cannot run under SC at all times and requires another *mission controller* to make progress.

3.2 Real-Time Reachability

For runtime computation of reachable states of a plant within a future time, we utilize a real-time reachability tool that is introduced in [11]. This low-cost algorithm is specifically designed for *embedded systems with real-time constraints and low computation power*.

Note that constructing a safety controller similar to that specified in section 3.1 (*e.g.,* having a recoverable region where any trajectory starting from that region will stay within that region) is generally not possible for non-linear systems. However, for specific classes of non-linear systems, our approach will be applicable if: *(i)* a safety controller with the properties mentioned above can be constructed and *(ii)* we can define a function that returns the minimum and maximum derivative in each dimension given an arbitrary box in the state space. This technique can also handle hybrid systems where the state invariants are disjoint and cover the continuous state $\mathbb{R}^n$, there are no reset maps in the transitions between discrete states and the state invariants define the guards of incoming transitions. In these piecewise systems, the state of the hybrid automaton can be determined solely by the continuous state; although separate differential equations can be used in various parts of the state space. This algorithm requires that the derivatives are defined in the entire state space and that they are bounded.

This technique uses the mathematical model of the dynamics of the plant and a n-dimensional box to represent the set of possible control inputs and the reachable states. A set of *neighborhoods*, $N[i]$ are constructed around each $face_i$ of the tracked states with an initial width. Next, the maximum derivative in the outward direction, d_i^{max}, inside each $N[i]$ is computed. Then, crossing time $t_i^{crossing} = width(N[i])/d_i^{max}$ is computed over all neighborhoods and the minimum of all the $t_i^{crossing}$ is chosen as time to advance, t^a. Finally, every face is advanced to $face_i + d_i^{max} \times t^a$. For further details on inward neighborhood versus outward neighborhoods, and the choosing of neighborhood widths and time steps refer to [11]. In this algorithm a parameter called `reach-time-step` is used to control neighborhood widths. This parameter lets us tune the total number of steps used in the method, and therefore alter the *total runtime* to compute the reachable set. This allows us to cap the total computation time of the reachable set – which is essential in any real-time setting.

Moreover, authors have demonstrated that this algorithm is capable of producing useful results within very short computation times *e.g.,* result achieved with computation times as low as $5ms$ using embedded platforms [11]. All these features make this approach a suitable tool for our target platforms as well.

4 Methodology

To explain our approach, let us assume that it is possible to create secure execution intervals (SEI) during which we can trust that the system is going to execute uncompromised software and adversary cannot interfere with this execution in any way. Under such assumption, we will show that it is possible to guarantee that a physical plant will remain within its admissible states as long as the following conditions remain true: *(i)* the timing between these intervals are separated such that, due to the physical inertia, the plant will not reach an inadmissible state until the beginning of the consequent SEI. *(ii)* The state of the plant at the beginning of the following SEI will be such that the SC can still stabilize the system. Under these conditions, the plant will be safe in between two SEIs (due to condition 1). If an adversary pushes the system close to the boundaries of inadmissible states, during the following SEI, we can switch to SC, and it can stabilize the plant (condition 2).

In the rest of this section, we present an analytical framework that shows how appropriately timed separations between the consequent SEIs guarantee the physical safety. Additionally, we show how these time values can be calculated in run-time. Finally, we discuss two different mechanisms – restart-based implementation and TEE-assisted implementation – to enable a trusted computation environment – SEI – during which the time intervals between SEI will be computed, without any adversarial interference.

4.1 Restart-based Secure Execution Intervals (SEI)

One essential element of the approach introduced in this paper is the run-time computation of the time separation between consecutive executions of the safety-critical tasks – the tasks that evaluate the safety conditions (next section) and stabilize the plant if necessary. The ultimate safety guarantees of our approach depend on the integrity of these computations. To achieve safety, therefore, it is essential to have a means to completely protect these tasks from any adversarial interference – adversary should not be able to stop or delay the execution or, corrupt the results of the computations. In this paper, we use the term Secure Execution Interval (SEI) to refer to execution intervals during which the integrity of the code is preserved.

One way to create SEIs in an untrusted environment is to rely on the *full platform restarts* and the *software reloads*. The procedure is as follows. For each SEI, the platform needs to restart entirely and then immediately load the clean software image from the read-only storage. Additionally, after the restart, all the external interfaces of the platform – those that might be an exploitation point for external adversaries – will remain disabled. As soon the platform boots, it can execute the safety-related tasks trustworthily and produce correct results. Once the execution of the critical tasks is finished, the time to trigger the following restart – the next SEI – is scheduled. Finally, the SEI ends, the external interfaces are activated, and the mission controller and other necessary components are launched.

An additional mechanism is necessary to schedule a restart and trigger it such that the adversary cannot prevent

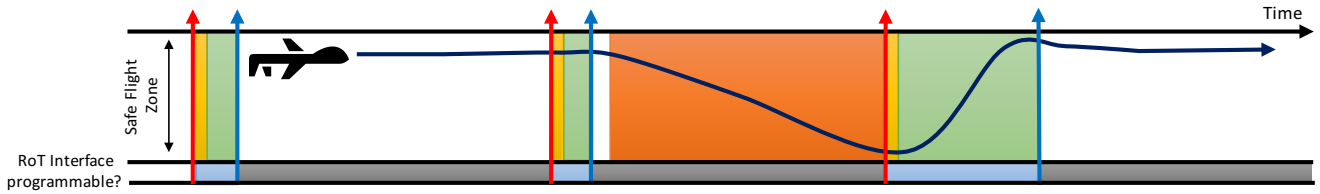


Fig. 1: An example sequence of events for the restart-based implementation of the SEI. White: mission controller is in charge and platform is not compromised. Yellow: system is undergoing a restart. Green: SEI is active, SC and `find_safety_window` are running in parallel. Orange: adversary is in charge. Blue: RoT accepts new restart time. Gray: RoT does not accept new restart time. Red arrow: RoT triggers a restart. Blue arrow: SEI ends, the next restart time is scheduled in RoT, and the mission controller starts.

it. We designate a separate HW module, called root-of-trust (RoT) to do this. RoT is essentially an external timer that can send a restart signal to the HW restart pin of the controller board at the scheduled time. It has an interface that allows the main controller to set the time of the next restart signal. We refer to this interface by `set_SEI_trigger_time`. The only difference of RoT with a regular timer is that it allows the processor to call the `set_SEI_trigger_time` interface only *once* after each restart and ignores any additional calls to this interface until the timer expires. Once the RoT timer is configured, adversaries cannot disable it until it has expired and the platform is restarted. Figure 1 illustrates the sequence of events in the system.

4.2 Finding the Safety Window in Run-Time

During the SEI, platform executes two tasks in parallel: *(i)* `find_safety_window` task which calculates the time window in which the plant will remain safe due to its physical inertia and uses this result to set the triggering time of the next SEI. And, *(ii)* SC that keeps the plant stable while `find_safety_window` is computing. Figure 1 presents an example sequence of the system events. If no malicious activity had taken place during the previous execution cycle (first cycle of Figure 1), the next SEI triggering time is computed and scheduled quickly, and the mission controller resumes. However, if an attacker had been able to compromise the platform within the previous cycle and managed to push the plant close to the inadmissible states (second cycle of Figure 1), the SC will need some time to stabilize the plant – push it further into the recoverable region – and SEI will be longer.

The fundamental idea here is how should `find_safety_window` calculate the triggering time of the next SEI such that up to the beginning of the next SEI, the physical plant would not be able to reach an unsafe state and at the beginning of next SEI, the state would still be *recoverable* by the SC. The rest of this subsection answers this question.

Before we proceed, it is useful to define some notations. We use the notation of $\mathsf{Reach}_{=T}(x, C)$ to denote the set of states that are reachable by the physical plant from an initial set of states x after exactly T units of time have elapsed under the controller C. $\mathsf{Reach}_{\leq T}(x, C)$ can be defined as $\bigcup_{t=0}^{T} \mathsf{Reach}_{=t}(x, C)$ *i.e.,* union of all the states reachable within all times t up to T time units. Also, we use SC to refer to the *safety controller* and UC to refer to an *untrusted controller*, *i.e.,* one that might have been compromised by an adversary. We use notation $\Delta(x_1, x_2)$ to represent the shortest time required for the physical plant to reach state x_2, starting from x_1.

Definition 3. *True Recoverable states are all the states from which the given SC can eventually stabilize the plant. Formally,* $\mathcal{T} = \{x \mid \exists \alpha > 0 : \mathsf{Reach}_{\leq \alpha}(x, SC) \subseteq \mathcal{S}\ \&\ \mathsf{Reach}_{=\alpha}(x, SC) \subseteq \mathcal{R}\}$. *The set of true recoverable states is represented with* $\mathcal{T}$.

Definition 4. $\mathcal{T}_\alpha$ *denotes the set of states from which the given SC can stabilize the plant within at most* α *time. Formally, we have* $\mathcal{T}_\alpha = \{x \mid \mathsf{Reach}_{\leq \alpha}(x, SC) \subseteq \mathcal{S}\ \&\ \mathsf{Reach}_{=\alpha}(x, SC) \subseteq \mathcal{R}\}$. *From definition it follows that* $\forall \alpha : \mathcal{T}_\alpha \subseteq \mathcal{T}$.

Let us call T_s, the switching time, and use it for referring to the time between the triggering time of the SEI until SEI is active and ready to execute tasks. For the restart-based SEI implementation, T_s is equal to the length of one restart cycle of the embedded platform[4]. Furthermore, let us use γ to represent the shortest time that is possible to take a physical system from its current state $x(t) \in \mathcal{T}$ to a state outside of $\mathcal{T}$. We can write

$$\gamma(x) = \min \{\Delta(x, x') \mid \text{ for all } x' \notin \mathcal{T}\} \tag{1}$$

It follows that

$$\text{If } x(t) \in \mathcal{T} \text{ then } x(t+\tau) \in \mathcal{T} \text{ where } \tau < \gamma(x(t)). \tag{2}$$

From Equation 2 we can conclude

$$\begin{matrix} \mathsf{Reach}_{\leq \gamma(x(t))-\epsilon}(x(t), UC) \subseteq \mathcal{S} \\ \mathsf{Reach}_{=\gamma(x(t))-\epsilon}(x(t), UC) \subseteq \mathcal{T} \end{matrix} \quad \text{where } \epsilon \to 0 \tag{3}$$

Equation 3 indicates that if it was possible to calculate $\gamma(x(t))$ in an SEI, we could have scheduled the consecutive SEI to be triggered at time $t + \gamma - T_s - \epsilon$. This process would have ensured that by the time the following SEI had started, the state of the plant was truly recoverable and admissible.

The value of $\gamma(x)$ depends on the dynamics of the plant and the limits of the actuators. Unfortunately, it is not usually possible to compute a closed-form representation for $\gamma(x)$. Because computing a closed-form representation for the $\mathcal{T}$ of the given SC is not a trivial problem. Actuator limits is another factor that needs to be taken into account in the calculation of $\mathcal{T}$. Therefore, in many cases, finding γ

4. T_s is the length of the interval from the triggering point of restart until the reboot is completed, filters are initialized and control application is ready to control the plant.

would require performing extensive simulations or solving numerical or differential equations.

An alternative approach is to check the conditions of Equation 3 for a specific value of time, λ:

$$\mathsf{Reach}_{\leq\lambda}(x(t), UC) \subseteq \mathcal{S} \;\&\; \mathsf{Reach}_{=\lambda}(x(t), UC) \subseteq \mathcal{T}_\alpha \quad (4)$$

Fortunately, having a tool to compute the reachable set of states in run-time allows us to evaluate all the components of Equation (4). Real-time reachability can compute the reachable set of states up to the λ time with an untrusted controller UC to check the first part of the equation (4). To evaluate the second part, we use the calculated reachable set at time λ as the starting set of states to perform another reachability computation for α time under SC and check $\mathsf{Reach}_{\leq\alpha}(\mathsf{Reach}_{=\lambda}(x(t), UC), SC) \subseteq \mathcal{S}$ and $\mathsf{Reach}_{=\alpha}(\mathsf{Reach}_{=\lambda}(x(t), UC), SC) \subseteq \mathcal{R}$. These two conditions are equivalent to the second part of the equation above.

The λ that is calculated for the state $x(t)$ is a *safety window* of the physical system in state $x(t)$, that is the interval of time, starting from time t, that the plant will remain safe and recoverable, even if the adversary controls it. Hence, we can conclude that the time $t + \lambda - T_s$, is a point where the platform can be safely restarted – *i.e.,* the next SEI can be triggered. Algorithm 1, performs a binary search and tries to find the largest safety window of the plant from a given $x(t)$ within a bounded computation time, T_{search}. Given a large T_{search}, Algorithm 1 would calculate the the *maximum safety window* of the plant for that state. In run-time, however, T_{search} has to be limited and therefore choosing the initial candidate $\lambda_{\text{candidate}}$ is crucial. It is also possible to use an adaptive λ_{init} by dividing the state space into subregions and assigning a λ_{init} to each region. At runtime, choose the λ_{init} associated with the state and initialize the Algorithm 1.

Algorithm 1: Finding physical safety window from state x. Here, $T_{\text{eq-4}}$ refers to the time required to evaluate the conditions if Equation 4. We can compute the exact value of $T_{\text{eq-4}}$ because the reachability computation time is capped (one of the imporant features of [11]) and, in total, there are 4 Reach operations to be performed.

find_safety_window(x, λ_{init})
1: `startTime` := currentTime()
2: $\lambda_{\text{candidate}} := \lambda_{\text{init}}$
3: `RangeStart` := T_s; `RangeEnd` := $\lambda_{\text{candidate}}$
4: **while** currentTime() - `startTime` $< T_{\text{search}} - T_{\text{eq-4}}$ **do**
5: **if** conditions of Equation (4) are true for $\lambda_{\text{candidate}}$ **then**
6: $\lambda_{\text{safe}} := \lambda_{\text{candidate}}$
7: `RangeStart` := λ_{safe}; `RangeEnd` := $2\lambda_{\text{safe}}$
8: **else**
9: `RangeEnd` := $\lambda_{\text{candidate}}$
10: **end if**
11: $\lambda_{\text{candidate}} := (\texttt{RangeStart} + \texttt{RangeEnd})/2$
12: **end while**
13: **return** -1

Note that the real actions of the adversary are unknown ahead of the time. As a result, in the conditions of Equation (4), the reachability of the plant under *all* possible control values need to be calculated. Consequently, the computed reachable set under UC ($\mathsf{Reach}(x, UC)$) is the largest set of states that might be reached from the given initial state, *within the specified time*. The real-time reachability tool in [11] allows this sort of computation due to the usage of a box representation for control inputs. Control inputs are set to the full range available to the actuators. As a result, the computed set the states that might be achieved under all of the actuator values. Notice that this procedure does not impact the time required for reachability computation.

When an intelligent adversary compromises the system, it can quickly push the plant towards the inadmissible states and very close to the boundary of the unsafe region. When operating close to the inadmissible states, there is a very narrow margin for misbehavior. If the adversary takes over again, they can violate the physical safety. Therefore, when SEI starts and the plant is in states very close to the boundary of the unsafe region, safety controller would need to execute for longer than usual until the plant is sufficiently pushed into the safe area. Deciding on how long the SC needs to run automatically happens based on the result of `find_safety_window` as presented in Algorithm 2. If the plant's state is too close to the boundary of the unrecoverable region, the safety window of the plant will be very short, and `find_safety_window` will most likely return -1. In Algorithm 2, this will force the while loop and consequently the SC to continue running for another cycle. This cycle will continue until SC has sufficiently distanced the plant from the unsafe region. At this point, `find_safety_window` will be able to compute a safety window and the SEI will end.

It's worth noting that what real-time reachability yields is a superset of the actual reachable set of states. Therefore, the calculated λ ensures that the system always remains within the safe region.

Algorithm 2: One operation cycle with restart-based SEI

1: Start Safety Controller. */* SEI begins */*
2: $\lambda_{\text{safe}} = \lambda_{\text{init}}$ /*Initializing the safety window*/
3: **repeat**
4: `start_time` := systemTime()
5: x := obtain the most recent state of the system from Sensors
6: λ_{safe} :=`find_safety_window`(x, λ_{safe})
7: `elapsed_time` := systemTime() - `start_time`
8: **until** $\lambda_{\text{safe}} \neq -1$ and $\lambda_{\text{safe}} > T_s +$ `elapsed_time`
9: Send $\lambda_{\text{safe}} -$ `elapsed_time` $- T_s$ to RoT. */* Set the next restart time. */*
10: Activate external interfaces. */* SEI ends. */*
11: Terminate SC and launch the mission controller.
12: When RoT sends the restart signal to hardware restart pin:
13: Restart the platform
14: Repeats the procedure from beginning (from Line 1)

5 TEE-Assisted Design Implementation

The restart-based approach to enable SEIs requires a restart in each operation cycle and imposes two main types of overheads on the system: *(i)* restart-time and *(ii)* memory erasure due to the restarts. Implementing this approach on some CPSs can be challenging especially if the platform restart time is not negligible compared to the speed of the dynamics of the plant. Another issue with this design implementation arises from the fact that the system restarts erase the platform memory. For some applications, such frequent memory erasures can be problematic. For instance, to establish a remote connection, the controller might need

to perform handshaking steps and store the state in the memory. If the system is frequently restarted, the controller may not be able to establish a reliable communication.

To mitigate some of these issues, we propose an alternative implementation where we use ARM TrustZone technology [43] and in particular LTZVisor [28] – which is a lightweight TrustZone assisted hypervisor with real-time features for embedded systems[5]. Here, instead of relying on the platform restarts to create SEIs, we exploit the isolated execution environments that are attainable through TrustZone.

In the rest of this section, we present some background on TrustZone and LTZVisor, and then we discuss the implementation of the approach.

5.1 Background on TrustZone and LTZVisor

TrustZone [43] hardware architecture can be seen as a dual-virtual system, partitioning all systems physical resources into two isolated execution environments. A new 33rd processor bit, the Non-Secure (NS) bit, indicates in which world the processor is currently executing, and is propagated over the memory and peripherals buses. An additional processor mode, the monitor mode, is added to store the processor state during the world switch. TrustZone security is extended to the memory infrastructure through the TrustZone Address Space Controller (TZASC) that can partition the DRAM into different memory regions. Secure world applications can access non-secure world memory, but the reverse is not possible. Additional enhancements in TrustZone provide the same level of isolation in cache and system devices.

LTZVisor [28] is a lightweight hypervisor that allows the consolidation of two virtual machines (VMs), running each of them in an independent virtual world (secure and non-secure). It exploits TrustZone features in the platforms to provide memory segmentation, cache-level isolation, and device partitioning between the two VMs. LTZVisor dedicates timers to each VM that enables each one to have a distinctive notion of system time. Additionally, it provides an API for communication between the two VMs.

LTZVisor manages the secure and non-secure world interrupts in a way that meets the requirements of the hard real-time systems. All the implemented interrupts can be individually defined as secure and non-secure. If the secure VM is executing, all the secure interrupts are redirected to it without hypervisor interference. If a non-secure interrupt arises during secure VM execution, it will be queued and processed as soon as non-secure side becomes active. On the other hand, if the non-secure VM is executing and a secure interrupt arises, it will be immediately handled in the secure world. This design prevents a denial-of-service attack on the secure-side applications.

LTZVisor implements a scheduling policy that guarantees that the non-secure guest OS is only scheduled during the idle periods of the secure guest OS, and the secure guest OS can preempt the execution of the non-secure one. This scheduling policy resolves one of the well-known real-time scheduling problems in virtual environments known as hierarchical scheduling and makes LTZVisor an excellent choice to meet real-time requirements of the tasks in the secure VM. Besides, creators of LTZVisor show that the overhead of switching from secure VM to non-secure VM and vice versa is small and deterministic [28]. Thus, secure VM is ideal for running a real-time operating system (RTOS) whereas, non-secure VM can run general purpose operating systems like Linux.

5. In this paper, we have used TrustZone and LTZVisor. Nevertheless, other available Trusted Execution Environment (TEE) technologies such as Intel's Trusted Execution Technology (TXT) [24] can be employed to achieve the same goal.

5.2 TEE-enabled SEIs

In this design, to protect the SC and `find_safety_window` tasks, they execute in the secure VM, and everything else runs in the non-secure VM. The SC and `find_safety_window` are executed, and before they finish, they schedule their next execution time *i.e.,* the next SEI. Mission controller and any other component start running as soon as all the tasks in the secure VM have yielded. LTZVisor guarantees that the non-secure VM cannot interfere with the execution of the tasks in the secure VM.

Each task inside the secure VM, once executed, can choose to yield and set the future time when its status will change to ready again. In LTZVisor, the secure VM has a higher priority than the non-secure VM. Consequently, the non-secure VM tasks will execute only when there are no secure tasks that are ready to execute. Similarly, as soon as one of the secure VM tasks becomes ready, LTZVisor pauses the non-secure VM, stores the necessary registers and executes the secure task. The scheduling policy in each VM determines the priorities and execution details for the tasks of that VM.

The operation cycle of the system during the SEI is very much the same as described in Algorithm 2 except instead of setting the RoT and the restarting step, secure tasks schedule their next wake up time using the secure platform timer or the OS of the secure VM. SC and `find_safety_window` tasks execute in parallel. As soon as `find_safety_window` finds a valid safety window, both tasks set their next wake up time and yield the execution. At this point, LTZVisor resumes the execution of the non-secure VM until it is time for the SC and `find_safety_window` to wake up.

Note that, due to the isolation provided by TrustZone, non-secure VM cannot interfere with the execution of secure tasks when they are ready to execute. This protection eliminates the need for the RoT timer which was a necessary component to implement the restart-based SEI.

5.3 Optional Recovery Restart

The safety guarantees that the TEE-based implementation provides are precisely the same guarantees as restart-based SEI implementation. Nevertheless, there is a significant difference. When the system is being restarted in every cycle, if it gets compromised, the malicious components will only last until the following restart, and then the software will be restored. When using TrustZone, if the non-secure world gets compromised, it will remain compromised. Although the adversary cannot violate the safety of the

plant, it can seriously prevent the system from making any progress.

There are two possible mechanisms to mitigate this problem. One arrangement is to introduce rare, randomized restarts into the system[6]. Another mitigation is to monitor the platform, during the SEI, for potential intrusions and malicious activities and restart the platform after the malicious behavior is detected[7]. Note that with the optional recovery restarts described in this section, a well-behaving system that is not under attack will rarely restart. The platform will be restarted only after it is deemed malicious or when the random function requires it to do so. Whereas, with the restart-based implementation of SEIs, the platform has to be restarted before every SEI.

Deciding whether the platform needs to restart or not takes place at the beginning of the SEI – either based on a randomized policy or a detection mechanism. If it is decided to restart, the steps to perform the recovery are presented in Algorithm 3. One crucial point in restarting the system is the fact that the platform restart must take place only when the plant is in a state where it will sustain the safety throughout the restart and will end up in a recoverable state – according to Defintion 2 – after the restart has completed. This requirement is satisfied if the conditions of Equation 4 are met.

Under these steps, SC continues to push the plant towards the center of the safe region. In parallel, the `find_safety_window` function is executed in a loop and checks if the plant at its current state meets the conditions of safe restarting in Equation 4 for the length of platform restart time. Once the `find_safety_window` confirms the safety conditions for the current plant state, the recovery restart is initiated. In other words, the system is restarted when the plant has enough distance from the boundaries of the recoverable states and unrecoverable states.

Algorithm 3: Steps to perform a recovery restart.

1: SC starts and is periodically invoked in parallel to the next steps.
2: $\lambda_{\text{Recovery}} = T_{\text{restart}} + T_{\text{eq-4}} + \epsilon$
3: **repeat**
4: $x :=$ obtain the most recent state of the system from Sensors
5: **until** conditions of Equation (4) are true for $\lambda_{\text{Recovery}}$
6: (optional) Store sensor reading in the non-volatile storage
7: Restart the system
8: /*Following steps are executed after the restart*/
9: (optional) Load the pre-restart sensor data from storage into the memory

5.4 Carrying Sensor State Between Restarts

Some control applications might need the prior-to-restart sensor readings for improved performance or higher quality output. For instance, low-pass filters use the past sensor readings to remove noise from the sensors. TEE-assisted implementation can accommodate this requirement. In this design, restarts are always initiated within the secure VM and, the secure VM is always the first to execute after the restart. Immediately prior to the restart, the secure VM can store any data on the non-volatile storage, and load it back into the memory after the restart. Note that the non-secure VM is not able to interfere with this process at all.

It is worth mentioning that the above procedure can be used to carry any values, including the variables or states in the non-secure VM, and make them available after the restart. However, we strongly advise avoiding a design where the CPS relies on the prior-to-restart state of the non-secure VM to carry out its essential mission mainly because the platform is restarted only when the non-secure VM is deemed compromised. At this point, all the states in the non-secure VM must be assumed corrupted. Passing the corrupted values across restarts can propagate the adversarial effect across the restarts and defeat the purpose of recovery restarts.

6 Evaluation and Feasibility Study

In this section, we evaluate the protections provided by our approach and measure the feasibility of implementing it on real-world CPSs. We choose two physical plants for this study: a 3-degree of freedom helicopter [29] and a warehouse temperature management system [39]. For both plants, the controller is implemented using both restart-based and TEE-assisted approaches on a ZedBoard [8] embedded system.

6.1 Test-Bed Description

6.1.1 *Warehouse Temperature Management System:*

This system consists of a warehouse room with a direct conditioner (heater and cooler) to the room and another conditioner in the floor [39]. The safety goal for this plant is to keep the room temperature, T_R, within the range of $[20°C, 30°C]$. Following equations describe the heat transfer between the heater and the floor, the floor and the room, and the room and outside space. The model assumes constant mass and volume of air and heat transfer only through conduction.

$$\dot{T}_F = -\frac{U_{F/R}A_{F/R}}{m_F Cp_F}(T_F - T_R) + \frac{u_{H/F}}{m_F Cp_F}$$

$$\dot{T}_R = -\frac{U_{R/O}A_{R/O}}{m_R Cp_R}(T_R - T_O) + \frac{U_{F/R}A_{F/R}}{m_R Cp_R}(T_F - T_R) + \frac{u_{H/R}}{m_R Cp_R}$$

Here, T_F, T_R, and T_O are the temperature of the floor, room and outside. m_F and m_R are the mass of floor and the air in the room. $u_{H/F}$ is the heat transferred from the floor heater to the floor and $u_{H/R}$ is the heat transferred from the room heater to the room both of which are controlled by the controller. Cp_F and Cp_R are the specific heat capacity of floor (in this case concrete) and air. $U_{F/R}$ and $U_{R/O}$ represent the overall heat transfer coefficient between the floor and room, and room and outside.

For this experiment, the walls are assumed to consist of three layers; the inner and outer walls are made of oak and isolated with rock wool in the middle. The floor is assumed to be quadratic and consists of wood and concrete. The

6. The rationale behind randomized system restarts – also known in the literature as software rejuvenation – is that there are no perfect intrusion detection mechanisms. Also, there will always exist malicious activities that will remain undetected. In our previous work [4], we have analyzed the impact of restart-based recovery on the availability of a system under attack.

7. In this paper, we do not propose any particular intrusion detection algorithm. There is a variety of such techniques that the system architects can choose from.

parameters used are as following[8]: $U_{R/O} = 539.61$ J/hm^{2}K, $U_{F/R} = 49920$ J/hm^{2}K, $m_R = 69.96$ kg, $m_F = 6000$ kg, floor area $A_{F/R} = 25$ m^2, wall and ceiling area $A_{R/O} = 48$ m^2, thickness of rock wool, oak and concrete in the wall and floor respectively 0.25 m, 0.15 m and 0.1 m. Maximum heat generation capacity of the room and floor conditioner is respectively 800 J/s and 115 J/s. And, the maximum cooling capacity of the room and the floor cooler is -800 J/s and -115 J/s.

6.1.2 3-Degree of Freedom Helicopter:

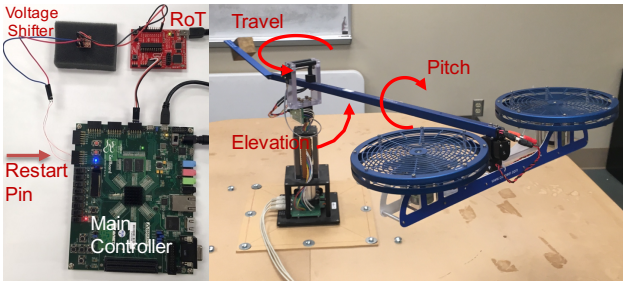


Fig. 2: 3DOF helicopter and the ZedBoard controller.

3DOF helicopter (displayed in figure 2) is a simplified helicopter model, ideally suited to test intermediate to advanced control concepts and theories relevant to real-world applications of flight dynamics and control in tandem rotor helicopters, or any device with similar dynamics [29]. It is equipped with two motors that can generate force in the upward and downward direction, according to the given actuation voltage. It also has three sensors to measure elevation, pitch, and travel angle as shown in Figure 2. We use the linear model of this system obtained from the manufacturer manual [29] for constructing the safety controller and calculating the reachable set in run-time. Due to the lack of space, the details of the model are included in our technical report [2].

For the 3DOF helicopter, the safety region is defined in such a way that the helicopter fans do not hit the surface underneath, as shown in Figure 2. The linear inequalities describing the safety region are $-\epsilon + |\rho|/3 \leq 0.3$, $\epsilon \leq 0.4$, and $|\rho| \leq \pi/4$. Here, variables ϵ, ρ, and λ are the elevation, pitch, and travel angles of the helicopter. Limitations on the motor voltages of the helicopter are $|v_l| \leq 4V$ and $|v_r| \leq 4V$ where v_l and v_r are the voltage for controlling left and right motors.

6.2 Restart-Based Implementation of SEI

In this section, we discuss the implementation of the controllers of the 3DOF platform and the temperature management system using the restart-based SEI approach (Section 4). In our technical report [2], more details are provided about the hardware and software implementation of the controller. Due to the limited access to a real warehouse, the controller interacts with a simulated model of the physical plant running on a PC (Hardware-in-the-loop simulation).

8. For the details of calculation of $U_{F/R}$ and $U_{R/O}$ and the values of the parameters refer to Chapter 2 and 3 of [39].

RoT Module:

The RoT module is implemented using a low-cost MSP430G2452 micro-controller on a MSP-EXP430G2 LaunchPad board [38]. To enable restarting, pin P2.0 of the micro-controller is connected to the restart input of the main controller. Internal Timer A of the micro-controller is used for implementing the restart timer. It is a 16-bit timer configured to run at a clock rate of 1 MHz (*i.e.*, 1μs per timer count) using the internal, digitally controlled, oscillator. A counter inside the interrupt handler of Timer A is used to extend the timer with an adjustment factor, in order to enable the restart timer to count up to the required range based on the application's needs.

The I^2C interface is adopted for the main controller to set the restart time on the RoT module. After each restart, during the SEI, the RoT acts as an I^2C slave waiting for the value of the restart time. As soon as the main controller sends the restart time, RoT disables the I^2C interface and activates the internal timer. Upon expiration of the timer, an active signal is set on the restart pin to trigger the restart event and the I^2C interface is activated again for accepting the next restart time.

Main Controller:

The controller is implemented on a Zedboard [8] which is a development board for Xilinx's Zynq-7000 series all programmable SoC. It contains an XC7Z020 SoC, 512 MB DDR3 memory, and an onboard 256 MB QSPI Flash. The XC7Z020 SoC consists of a processing system (PS) with dual ARM Cortex-A9 cores and 7-series programmable logic (PL). The processing system runs at 667MHz. In our experiments, only one of the ARM cores is used, and the idle cores are not activated. The I^2C and *UART* interfaces are used for connecting to the RoT module and the actuators of the plant. Specifically, two multiplexed I/Os, MIO14 and MIO15, are configured as SCL and SDA for I^2C respectively. We use UART1 (MIO48 and MIO49 for UART TX and RX) as the main UART interface.

The reset pin of Zedboard is connected to RoT module's reset output pin via a BSS138 chip, an N-channel voltage shifter. It is because the output pin on RoT module operates at 3.3 volts while the reset pin on Zedboard accepts 1.8 volts. The entire system (both PS and PL) on Zedboard is restarted when the reset pin is pulled to the low state. The boot process starts when the reset pin is released (returning to the high state). A boot-loader is first loaded from the onboard QSPI Flash. The image for PL is then loaded by the boot-loader to program the PL which is necessary for PS to operate correctly. Once PL is ready, the image for PS is loaded, and the operating system will take over the control of the system.

The platform runs *FreeRTOS* [1], a preemptive real-time operating system. Immediately after the reboot, `safety_controller` and `find_safety_window` tasks are created and executed. `safety_controller` is a periodic task with the period of 20 ms (50 Hz) and the execution time of 100 μs and has the highest priority in the system. Safety controller itself is designed using the method described in Section 3.1. Each invocation of this tasks obtains the values of sensors and sends the control

commands to the actuators. `find_safety_window` executes a loop and only breaks out when a valid safety window is calculated. It executes at all times except when it is preempted by `safety_controller`. When `find_safety_window` computes a valid safety window, it sends the value minus the elapsed time (Algorithm 2) to the RoT module via the I^2C interface, sets a global variable in the system, and terminates. Based on this global variable, `safety_controller` task terminates, and the mission controller task is launched. `find_safety_window` is implemented based on the Pseudo-code described in Algorithm 1. Execution time of each cycle of the loop in this function is capped at 50 ms (*i.e.*, $T_{\text{search}} := 50$ ms). In `find_safety_window`, to calculate the reachability of the plant from a given state, we used the implementation of our real-time reachability tool [11]. All the code for the implementation can be found in the GitHub repository [2].

3DOF Helicopter Controller: ZedBoard platform interfaces with the 3DOF helicopter through a PCIe-based Q8 data acquisition unit [30] and an intermediate Linux-based machine. The PC communicates with the Zedboard through the UART interface. Mission controller is a PID controller whose goal is to navigate the 3DOF to follow a sequence of set points. Control task has a period of 20 ms (50 Hz), and at every control cycle, the control task receives the sensor readings (elevation, pitch, and travel angles) from PC and sends the next set of voltage control commands for the motors. The PC uses a custom Linux driver to communicate with the 3DOF sensors and motors. In our implementation, the restart time of the ZedBoard with FreeRTOS is upper-bounded at $390ms$.

Warehouse Temperature Controller: Due to the lack of access to the real warehouse, we used a hardware-in-the-loop approach to perform the experiments related to this plant. Here, the PC simulates the temperature based on the heat transfer model Described in Section (6.1.1). The mission controller is a PID that adjusts the environment temperature according to the time of the day. The controller is implemented on the ZedBoard with the same components and configurations as the 3DOF controller – RoT, serial port connection, I^{2}C interface, $50Hz$ frequency, and the same restart time. Control commands are sent to the PC, applied to the simulated plant model and the state is reported back to the platform.

6.3 TrustZone-Assisted SEI implementation

Our prototype implementation uses LTZVisor on the ZedBoard which provides two isolated execution environments, secure VM and non-secure VM. LTZVisor can only use one of the ZedBoard cores, and the other cores are not activated. Similar to the previous section, ZedBoard is connected to the physical plant sensors and actuators through *UART* interface. The configuration of the *UART* pins and PL are the same as the previous section.

`safety_controller` and `find_safety_window` are compiled as one bare metal application and executed in the secure VM[9]. The functionality of these components is identical to what was described in the previous section. Using the platform timer, we ensure that the `safety_controller` function is called and executed every 20 ms while, `find_safety_window` is being executed for the rest of the time. Once the state of the plant reaches a state where a safety window is available, `find_safety_window` returns the results, the application yields the processor and sets the next invocation point to the current time plus computed safety window minus the computation time – Section 5.2. At this point, LTZVisor restores the execution of the mission controller application in the non-secure VM until the secure VM application is invoked again. We use the `YIELD` function, provided by the LTZVisor on the secure VM, which suspends the execution of the application and invokes it after the specified interval of time.

In our prototype, recovery restarts are initiated based on a randomized scheme. We use a pseudo-random number generator function that returns a value between 0 and 1. if the values are less than $1/1000$, we restart the platform – the probability of 0.1 percent. Otherwise, the execution proceeds to the normal SEI. The mechanism to trigger the restarts is through system-level watchdog timer. This is an internal 24-bit counter that on timeout outputs a system reset to the Processing System (all the cores and system registers) and Program Logic (the FPGA fabric in the ZedBoard). To trigger a restart, the timer is enabled and set to expire on the shortest time allowed by the resolution. The timer expires immediately and restarts the platform.

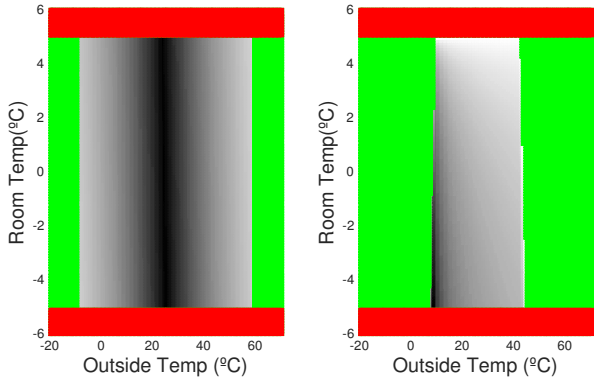
6.4 Safety Window of the Physical Plants

At the end of each SEI, the triggering point of the next SEI needs to be computed and scheduled. Two main factors determine the distance between consecutive SEIs; *(i)* how stable the dynamics of the plant is and *(ii)* the proximity of the current state of the plant to the boundaries of the inadmissible states. In figures 3 and 4, the absolute maximum safety window of the physical plant is plotted from various states for the plants under consideration. These values are computed using Algorithm 1 except for clarification, the lower end of the search in this algorithm, `RangeStart`, is set to 0. In these plots, the red region represents the inadmissible states, and the plant must never reach those states. If the plant is in a state that is marked green, it is still undamaged. However, at some future time, it will reach an inadmissible state, and the safety controller may not be able to prevent it from coming to harm. The reason is that actuators have a limited physical range. In the green states, even actuators operating with the maximum capacity, may not be able to cancel the momentum and prevent the plant from reaching unsafe states. The gray and yellow highlighted regions are the operational region of the plant – states where the safety window of the plant is larger than zero and mission controller can execute. In the gray area, the darkness of the color is the indicator of the length of the safety window in that state. Darker points indicate a larger value for the safety window.

Figures 3(a) and 3(a), plot the calculated safety windows for the warehouse temperature management system. For this system, when the outside temperature is too high or too

9. LTZVisor also provides support for FreeRTOS on the secure VM and Linux on the non-secure VM. However, at the time of this writing, the code enabling these features is not publicly released yet. That is why these components are implemented as bare-metal applications.

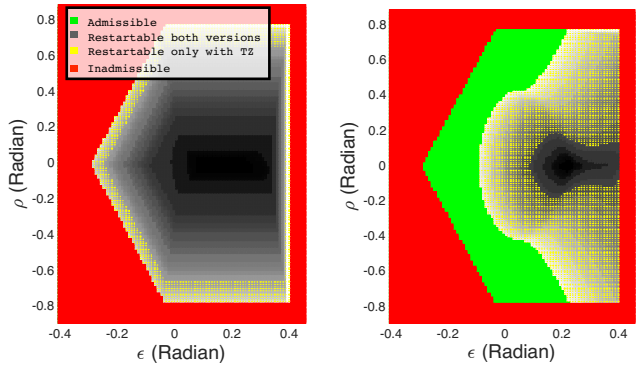
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2018.2889866, IEEE Internet of Things Journal



(a) Projection into $T_F = 25C$ (b) Projection into $T_F = 29C$

Fig. 3: Safety window values for the warehouse temperature. Largest value of the safety window – the darkest region – is $6235s$.

low, the attacker requires less time to take the temperature beyond or below the safety range. Note that if an adversary take over the platform at $T_F = 25C$, $T_R = 40C$, and $T_O = 26C$ – top part of Figures 3(a) – and runs the heaters at their maximum capacity, plant will remain safe for $6235s$. Intuitively, due to high conductivity between the floor and the room as well the high heat capacity of the floor, the rate of heat transfer from room to the floor is larger than the transfer rate from the heater to the room. Due to the same reason, when the floor temperature is $T_F = 29C$, the safety window of the plant is almost zero near the boundary of the $T_R = 40C$ – top part of Figure 3(a).



(a) Projection of the state space into the plane $\dot{\epsilon} = 0$, $\dot{\rho} = 0$, $\lambda = 0$, and $\dot{\lambda} = 0.3\text{Radian}/s$

(b) Projection of the state space into the plane $\dot{\epsilon} = -0.3\text{Radian}/s$, $\dot{\rho} = 0$, $\lambda = 0$, and $\dot{\lambda} = 0.3\text{Radian}/s$

Fig. 4: Safety window values for the 3DOF helicopter. Largest value of the safety window – the darkest point – is $1.23s$.

In Figure 4, the safety window for the 3DOF helicopter are plotted – projection into the 2D plane. The darkest point, have the largest safety window which is $1.23s$. As seen in this figure, safety window is largest in the center where it is farthest away from the boundaries of the unsafe states. In Figure 4(b), the angular velocity of 3DOF elevation is $\dot{\epsilon} = -0.3\text{Radian}/s$ which means that the helicopter is heading towards the bottom surface at a rate of 0.3 Radian per second. As seen in the figure, with this downward velocity, the plant cannot be stabilized from the lower elevation levels (*i.e.*, the green region). It can also be seen that in the states with elevation less than 0.1 Radians, the safety window is shorter in Figure 4(b) compared to Figure 4(a). Intuitively, for the adversary, crashing the 3DOF helicopter is easier when the plant is already heading downward.

As we mentioned earlier, the temperature management system has higher inertia and slower dynamics than the 3DOF helicopter. The above figures reflect this effect, very clearly. As the computed safety windows for the former plant are orders of magnitudes larger than the latter – 6235 s is the largest safety window for warehouse temperature versus 1.23 s for the 3DOF helicopter. In this system, the rate of the change of the temperature even when the heater/coolers run at their maximum capacity is slow, and adversary needs more time to force the state into unsafe states.

Now, we will discuss the difference between the gray and yellow regions. The mission controller can operate in the yellow states only with the TEE-assisted implementation of the SEIs and not with the restart-based implementation of the SEIs. This is due to the following reason. In run-time, computed safety windows are used to set the triggering point of the next platform SEI. However, the next SEI can be scheduled only if the safety window is larger than the switching time of the platform, T_s, as presented in Algorithm 2. With the restart-based implementation of the SEIs, the switching time is equal to the restart time of the platform (390 ms for RTOS on the ZedBoard) whereas, with the TEE-assisted implementation, switching time is the timing overhead of the context switching from secure VM to non-secure VMs and vice versa (less than 12 μs for ZedBoard at 667 MHz as presented in [28]). States marked with the yellow color are those that the computed safety window is shorter than the platform restart time. At these states, with the restart-based SEI, the mission controller cannot be activated.

As a result of using TrustZone-assisted implementation, we measured a 234 percent increase in the size of the operational region of the 3DOF plant – the yellow vs. the gray area – across the 6-dimensional state space. However, note that this measurement is very specific to this particular platform and this specific plant. The expected improvement highly depends on the platform restart time and the speed of the plant dynamics. Not every CPS can be expected to gain significant benefits from adopting TrustZone for implementing the SEIs. For instance, if the restart time of the platform were shorter, the size of the gray area in Figure 4 would have been larger, and the overall improvement of the operable states – as a result of using TrustZone – would have been smaller. Comparison between the size of the yellow region for the 3DOF vs. the temperature management system is another clear implication of this point. The platform restart time compared to the length of the safety windows of the warehouse plant is almost negligible. That is why implementing the SEIs using TrustZone does not yield any noticeable improvements and the yellow region in Figure 3 is non-visible.

6.5 Impact on Controller Availability

Every CPS has a mission that is the primary goal of the system to accomplish. The main component that drives the system towards this goal is the mission controller. Therefore, every process that interrupts the execution of the mission controller results in the slow progress of the CPS mission. Thus, one of the consequences of our design is that the SEIs and the platform restarts stop the execution of the mission controller and reduce its availability. In this section, we measure the impact of each one of the two implementations of our design, on the average availability of the mission controller.

The exact "availability" of the mission controller is the ratio of time that the mission controller is executing (all the times that the system is not in the SEI and is not going through a restart) to the total time of the operation. In every restart cycle, availability is defined as $\delta_{\text{mc}}/(\delta_{\text{mc}} + T_{\text{SEI}} + T_s)$. Here, δ_{mc} is the duration of mission controller execution, T_{SEI} is the length of SEI, and T_s is the switching time. With the restart-based implementation of the SEIs, T_s is equal to the restart time of the platform, whereas, for the TrustZone-assisted SEI implementation, T_s is the upper bound of the time required for switching from non-secure VM to secure VM and vice versa. The exact availability of the mission controller is specific to the particular trajectory that the plant takes. To get a better sense of this metric, for each implementation, we compute the average availability of the mission controller across all the states where the safety window is longer than the switching time, T_s, which is 390 ms for restart-based SEI and 12 μs for the TrustZone-assisted SEI implementation.

For the 3DOF system, with the restart-based implementation, the calculated average availability of the mission controller is $\%51.2$. As seen in the Figure 4, safety windows of the 3DOF plant are in the range of 0 s to 1.23 s. The platform has a restart time of 390 ms which is significant relative to the values of safety windows and it notably reduces the availability of the mission controller. On the other hand, with the TrustZone-assisted SEIs, the average availability of the mission controller is $\%85.1$. When TrustZone is utilized, T_s is negligible – $12\mu s$ which explains the $\%35$ improvement in the availability. It can be seen that despite the negligible switching switching overhead, the mission controller does not reach $\%100$ availability. This is because of the time required to evaluate the safety conditions and execute `find_safety_window` in the loop inside Algorithm 2. In the states near the unsafe/safe state boundary, the platform might need to execute the loop cycle more than once – longer SEI allows the safety controller to create enough distance from the unrecoverable states.

For the temperature management system, the average availability of the mission controller is $\%99.9$ with both restart-based and TrustZone-assisted implementations of the SEIs. Due to the slow dynamics of this plant, safety windows are much longer than the T_s and T_{SEI} under both implementations – as illustrated in Figure 3. Hence, the mission controller is almost always available. Due to the same reason, reduced switching time that is achieved when the controller is implemented using TrustZone instead of the restarts does not notably improve the average availability of the mission controller.

The above results show that the impact of our approach on the temperature management system is negligible under both implementation schemes. In fact, the restart-based implementation is the most suitable choice for this plant and many other high-inertia plants. On the other hand, integrating our design into the controller of the 3DOf helicopter comes with a considerable impact on the availability of the helicopter controller. Even though the TrustZone considerably reduces the overhead and improves the availability, but still the control performance will noticeably suffer. Note that, the helicopter system is among the most unstable systems and therefore, one of the most challenging ones to provide *guaranteed* protection. As a result, the calculated results for the helicopter system can be considered as an approximate upper bound on the impact of our approach on the controller availability. In the next section, we demonstrate that, despite the reduced availability, the helicopter and warehouse temperature remain safe and the plants make progress. Reduced availability of the controller is the cost to pay to achieve guaranteed safety and can be measured ahead of time by designers to evaluate the trade-offs.

6.6 Attacks on the Embedded System

To evaluate the effectiveness of our proposed design, we perform three attacks on the controllers of the 3DOF helicopter (with the actual plant) and one attack on the hardware-in-the-loop implementation of the temperature management system. All the attacks are performed on both versions of the controller implementation. In these experiments, our focus is on the actions of the attacker after the breach into the system has taken place. Hence, the breaching mechanism and exploitation of the vulnerabilities are not a concern of these experiments. An attacker may use any number of exploits to get into the controller device.

In the first experiment, the mission controller of the temperature management system was attacked in the following way. The outside temperature was set to 45° C, and initial room temperature was set to 25° C. Immediately after the SEI was finished, the malicious controller forced both of heaters to increase the temperature with their maximum capacity. Under the restart-based SEI, we observed that the platform was restarted before the temperature reached 30° C and after the restart, SC was able to lower the temperature. Similar behavior was observed with the TrustZone-assisted implementation. A switch to the secure VM was triggered before the temperature reached an unrecoverable value, the SC was able to lower the temperature.

Second attack experiment was performed on the 3DOF helicopter. Here, the attacker, once activated, killed/disabled the mission controller. Under the restart-based SEIs, in every operation cycle, the restart action reloads the software and revives the mission controller. Therefore, the attack was activated at a random time after the end of the SEI in each cycle. Under the TrustZone-assisted SEI implementation, once the mission controller is killed, it will only be recovered

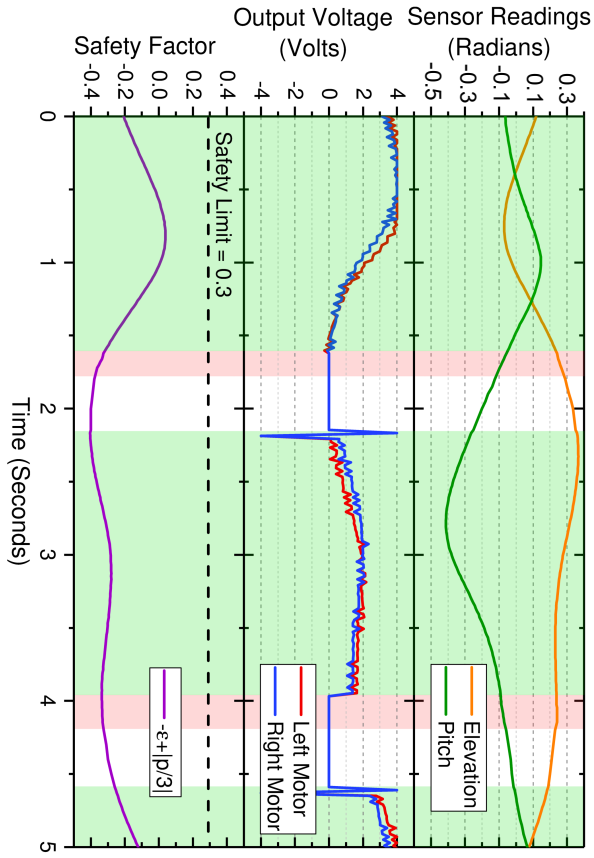


Fig. 5: 3DOF Helicopter trace under restart-based implementation during two cycles when the system is under worst-case attack (where attacker is active immediately after the SEI). Green: SEI, red: mission controller (in this case attacker), white: system reboot.

when a randomized recovery restart is performed[10]. We used a random value to activate the attack at a random operation cycle – with a probability of 1 percent. After the recovery restart, mission controller was revived and controlled the plant until the next attack was triggered. During these experiments, we observed that the 3DOF helicopter did not hit the surface *i.e.*, it always remained within the admissible set of states.

In the third experiment, the attacker corrupts the sensor readings and feeds the corrupted values in the mission controller logic. To evaluate the safety under an extreme case, the attack is activated immediately after the end of SEI. In both implementations of the controller, the attack is active during all the non-SEI and non-restart times of the system. Similar to the first attack experiment, it was observed that the 3DOF helicopter remained safe throughout the attack.

In the last attack experiment, we investigate the effectiveness of our design against an attacker that is active immediately after the SEI, replaces the original controller with a malicious process that turns off the motors/fans of the helicopter, and forces the plant to hit the surface. During the operation of the malicious controller, the elevation of the helicopter was reduced. However, in every cycle, before a crash, the safety controller will take over, push the helicopter and increase the elevation. Throughout this experiment, we observed that the plant tolerated the adversarial behavior and did not hit the surface.

A trace of the states of 3DOF helicopter during two consequent restart cycles, with the restart-based implementation of SEIs, is plotted in Figure 5. This trace is recorded from the sensor readings of the real physical plant when the plant is under the last attack experiment. The figure depicts elevation, pitch, actuator control inputs (voltages of the motor), and the safety factor. The safety factor is obtained from the safety conditions for the 3DOF as described in Section 6.1. From the figure, it can be seen the controller spends most of the time in SEI (green region) and reboot (white region) state. This is because this extreme-case attack is activated immediately after each SEI and destabilizes the helicopter. By the time that the reboot completes (end of the white region), the system is close to unsafe states. Hence, SEI becomes longer so that the SC can stabilize the helicopter. Under this very extreme attack model, the system did not make any progress towards its designated path, yet it remained safe which is the primary goal in this situation.

7 Related Work

There is a considerable number of techniques in the area of fault-tolerant CPS design that focuses on protecting the physical components in the presence of faults[11]. Although similar, there are fundamental differences between protecting against faults vs. protecting against an intelligent adversary. In what follows we review some of the papers and elaborate the differences and similarities.

The Simplex architecture [35] is a well known fault-tolerant design for CPS. It deploys two controllers: *(i)* a high-performance (yet unverifiable) controller and *(ii)* a high-assurance, formally verified, safety controller. A decision module (formally verifiable) is used to take over the control in the case that the high-performance controller is pushing the physical system beyond a precomputed safety envelope. A few variants of Simplex design exist. Some use a varying switching logic [11], [12] while others utilize a different safety controller [6], [46]. Nevertheless, all these designs assume that only a subset of the software misbehaves (for instance, they assume that switching unit cannot misbehave), which is invalid when the systems are under attack, and no other mechanism – such as restarts or TEE features are employed. In contrast, our work assumes that the adversary can corrupt "all" layers of the software.

Another variant of the Simplex architecture is System-Level Simplex [10] where the safety controller and the decision module run on dedicated hardware to isolate them from any fault or malicious activities on the complex controller (*i.e.*, the high-performance controller). Techniques based on this architecture [4], [5], [10], [47] guarantee the safety of the physical plant even when the complex

10. Note that in our prototype implementation, we did not implement a detection mechanism. However, one could deploy the logic to monitor the mission controller and restart the platform as soon as the controller is disabled.

11. Where the safety invariants of the physical plant must be preserved despite the possible implementation and logical errors in the software. Here, 'faults' refer to bugs in the software implementations. Another definition for faults exists that includes physical problems (*e.g.*, broken sensors/actuators/etc) – we do not consider them in this paper.

controller is under attack. However, implementing the System-Level Simplex design on most COTS platforms is challenging since most commercial multicore platforms are not designed to support strong inter-core isolation (due to the high degree of hardware resource sharing). For instance, an adversary residing in the high-privileged core may compromise power and clock configurations of the entire system. Hence, full isolation can only be achieved by utilizing two separate boards. On the other hand, our design provides formal safety guarantees using only one computing unit.

Trusted hardware features are commonly employed in the literature to achieve security goals. Some works have deployed the Trusted Platform Module (TPM) to build trusted computing environments on servers and hypervisors [27], [31], [32]. ARM TrustZone has been utilized in recent literature [20], [44], [45] to implement security monitors in the secure world. Authors in [9], leverage TrustZone and propose TZ-RKP to protect the integrity of the operating system kernel running in the normal, non-secure world. The use of trusted hardware features to create trusted execution environments is somewhat equivalent to the SEI concept as presented in our paper. The analytical framework proposed in this paper could be combined into these techniques to develop a diverse set of CPS platforms that can provide physical safety guarantees.

Restart-based recovery is previously explored in some of the aforementioned Simplex-based works [4], [5]. Specifically, these works restart the isolated, dedicated complex controller unit – equivalent to the mission controller. Restarting the complex controller while a safety controller running on separate hardware maintains the safety during the restart is more straightforward than restarting the entire platform. Another Simplex-based work in which the authors use a single hardware unit implements full-system restarts [6]. Nevertheless, this work assumes that the safety controller and the decision module may not be compromised and are always correct. Again, this assumption is invalid in the security context, and the physical safety cannot be guaranteed when the system is under attack.

A recent work studies frequent restarts and diversification for embedded controllers to increase the difficulty of attacks [7]. In spite of the conceptual similarity, our works mainly differ in the calculation of restart times. By dynamically calculating the next restart time using real-time reachability in each cycle, we can *guarantee* the system safety. Whereas, the authors in [7] empirically choose the restart times without any formal analysis.

The idea of restarting (either the entire system or a part of the components) at run-time is not novel and has been studied in earlier research to handle the problem of *software aging* in two forms: *(i) revival* (*i.e.,* reactively restarting a failed component) and *(ii) rejuvenation* (*i.e.,* proactively restarting functioning components). Some research [19], [23], [40] have tried to model failures and faults for client-server applications and tried to find an optimal rejuvenation strategy with the aim to reduce the system downtime. Some have introduced recursively restartable systems for fault-recovery and increased availability for Internet services [13]. The concept of microreboot (*i.e.,* systems consist of fine-grain rebootable components) is explored in [14], [15], [16]. In spite of entirely different purposes, these works assert the effectiveness of restarting as a recovery technique. In this context, some rejuvenation schemes [21] tackle software aging problems related to arithmetic issues such as the accumulation of numerical errors in controllers of safety-critical plants. Nevertheless, the rejuvenation techniques for safety-critical systems are very limited. A survey displays that, in this research area, only 6 percent of the published papers have considered safety-critical applications [18].

The philosophy of our work is similar to that of the works in a trend in systems dependability that applies the concepts and mechanisms of fault tolerance in the security domain, intrusion tolerance (or Byzantine fault tolerance) [17], [42]. These works advocate for designing intrusion-tolerant systems rather than implementing prevention against intrusion. Many works in intrusion-tolerant systems have targeted distributed services in which replication and redundancy are feasible. Their goals are mainly to ensure the availability of the system service even if some of its nodes are compromised. Another work proposes to proactively restore the system code from a secure source to eliminate any potential transformations carried out by an attacker [17]. With proactive recovery, the system can tolerate up to f faults/intrusions, as long as no more than f faults occur in between rejuvenations. In [41], the authors propose a general hybrid model for distributed asynchronous systems with partially synchronous components, named *wormholes*. In [37], the authors take wormholes as a trusted secure component (similar to our root of trust timer) which proactively recovers the primary function of the system. The authors suggest that such a component can be implemented as a separate, tamper-proof hardware module in which the separation is physical; or it can be implemented on the same hardware with virtual separation and shielding enforced by software. A proactive-reactive recovery approach is introduced in [36] (built on top of [37]) that allows correct replicas to force the recovery of a faulty replica. While these techniques are useful for some safety-critical applications such as supervisory control and data acquisition (SCADA), they may not be directly applicable to safety-critical CPS. Potentially, a modified version of these solutions might be utilized to design a cluster of replicated embedded controllers in charge of a physical plant.

8 Discussion

Some limitations need to be considered before deploying this design to a physical plant or platform. The restart-based implementation is most suitable for CPSs where the platform restart time is much smaller than the speed of the plant dynamics. Many embedded systems have reboot times that range from tens of milliseconds [26] to tens of seconds which are considered non-significant for many applications such as temperature/humidity management in storage/transportation industries, process control in chemical plants, pressure control in water distribution systems, and oxygen level management in patient bodies. The main advantage of the restart-based implementation of SEIs is that it can be deployed on the cheapest, off-the-shelf micro-controllers that are still widely used in many industrial applications. Also, the deployed application must

be designed to operate within the system's safety boundary. Otherwise operation of the system is trivially unsafe and the safety controller is unusable.

On the other hand, using the restart-based design on the physical plants with high-speed dynamics will require very frequent restarts and will significantly reduce the control performance and the progress of the system. Frequent reboots may also pose implementation challenges. For instance, the control device may need time to re-establish a connection over the Internet or to authenticate with the ground control. Such actions might not be possible if the device has to restart frequently. These types of applications will significantly benefit from the TrustZone-assisted implementation that eliminates the overhead associated with restarting. As a future direction, we are exploring a multicore implementation of TrustZone-assisted design where the SEI runs in parallel to the mission controller and has minimal impact on the mission controller's performance.

While a restart clears an instance of an attack it does not mean that the adversary is eliminated. It is possible that the adversary attempts to compromise and damage the system after each restart. However, even attack states cannot be carried across multiple attack instances due to the restarts. Each attack instance is contained by the proposed approach since the system restarts before it reaches the unsafe region. As a result, safety of the entire system is guaranteed.

One question that may arise is why not implement all the controllers using TrustZone? Platforms equipped with TrustZone or other TEEs are more expensive. Many control applications are deployed on very low-cost microcontrollers where only restart-based approach is feasible. Furthermore, many high-inertia physical plants will not gain any notable benefit if they are implemented via TrustZone – as shown for temperature management system in the evaluation section. In those cases, the TrustZone-based implementation only unnecessarily complicates the design and implementation of the CPS.

It should be noted that restart-based SEI is only suitable for stateless controllers (*e.g.*, mission controller) where the control command is generated based on the *current* state of the plant and environment. Such a design is useful for some applications but cannot be utilized with stateful controllers. In fact, for the very same reason, we introduce the TEE-based SEI in this paper. One question that comes into mind is about the compatibility of a stateful controller with TEE-based SEI implementation and recovery restarts? Note that with TEE-based SEI approach, the system is restarted only when it is detected to be compromised. Under the assumptions of our threat model, an adversary can maliciously modify all the state on the memory and disk (except read-only storage). In other words, even before the restart, the actual state of the system is already lost, and the stateful mission controller cannot continue to operate. Restarting the system at this point only loses the untrustworthy and hence unusable state.

Another important point to mention is that, under both restart-based and TEE-based implementations of SEI, the safety controller has to be a stateless controller so that it can safely stabilize the plant without the knowledge of its past states. This is the main reason that even with the TEE-based SEI design approach, only mission-controller, which is not critical for the safety, can be stateful. In this case, due to the loss of states after the compromise, system will inevitably suffer a performance loss, but the safety will not be violated. This can be another limiting factor on the type of systems or the kind of safety constraints imposed on it that needs to be considered when using our approach.

9 Conclusion

In this paper, we present an attack-tolerant design for embedded control devices that protects the safety of physical plants in the presence of adversaries. Due to the physical inertia, pushing a physical plant from a given (potentially safe) state to an unsafe state – even with complete adversarial control – is not instantaneous and often takes finite (even considerable) time. We leverage this property to calculate a *safe operational window* and combine it with the effectiveness of *system-wide restarts* or Trusted Execution Environments such as TrustZone to protect the safety of the physical system. We evaluate our approach on realistic systems and demonstrate its feasibility.

Acknowledgments

The authors would like to thank Sandro Pinto for providing support and being patient with all our questions regarding LTZVizor. The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grant numbers CNS-1646383 and SaTC-1718952. Marco Caccamo was also supported by an Alexander von Humboldt Professorship endowed by the German Federal Ministry of Education and Research. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] FreeRTOS . http://www.freertos.org, 2016. Accessed: Sep. 2016.
[2] https://github.com/emsoft2017restart/restart-based-framework-demo, 2017.
[3] F. Abdi, C.-Y. Chen, M. Hasan, S. Liu, S. Mohan, and M. Caccamo. Guaranteed physical security with restart-based design for cyber-physical systems. In *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*, ICCPS '18, pages 10–21, Piscataway, NJ, USA, 2018. IEEE Press.
[4] F. Abdi, M. Hasan, S. Mohan, D. Agarwal, and M. Caccamo. ReSecure: A restart-based security protocol for tightly actuated hard real-time systems. In *IEEE CERTS*, pages 47–54, 2016.
[5] F. Abdi, R. Mancuso, S. Bak, O. Dantsker, and M. Caccamo. Reset-based recovery for real-time cyber-physical systems with temporal safety constraints. In *IEEE 21st Conference on Emerging Technologies Factory Automation (ETFA 2016)*, 2016.
[6] F. Abdi, R. Tabish, M. Rungger, M. Zamani, and M. Caccamo. Application and system-level software fault tolerance through full system restarts. In *Proceedings of the 8th International Conference on Cyber-Physical Systems*, ICCPS '17, pages 197–206, New York, NY, USA, 2017. ACM.
[7] M. Arroyo, H. Kobayashi, S. Sethumadhavan, and J. Yang. FIRED: frequent inertial resets with diversification for emerging commodity cyber-physical systems. *CoRR*, abs/1702.06595, 2017.
[8] AVNET. Zedboard hardware users guide. http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf. Accessed: Apr. 2017.
[9] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 90–102. ACM, 2014.

[10] S. Bak, D. K. Chivukula, O. Adekunle, M. Sun, M. Caccamo, and L. Sha. The system-level simplex architecture for improved real-time embedded system safety. In *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*, pages 99–107. IEEE, 2009.

[11] S. Bak, T. T. Johnson, M. Caccamo, and L. Sha. Real-time reachability for verified simplex design. In *Real-Time Systems Symposium (RTSS), 2014 IEEE*, pages 138–148. IEEE, 2014.

[12] S. Bak, K. Manamcheri, S. Mitra, and M. Caccamo. Sandboxing controllers for cyber-physical systems. In *Proceedings of the 2011 IEEE/ACM Second International Conference on Cyber-Physical Systems*, ICCPS '11, pages 3–12, Washington, DC, USA, 2011. IEEE Computer Society.

[13] G. Candea and A. Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 125–130. IEEE, 2001.

[14] G. Candea and A. Fox. Crash-only software. In *HotOS IX: The 9th Workshop on Hot Topics in Operating Systems*, pages 67–72, 2003.

[15] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot- a technique for cheap recovery. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 31–44, 2004.

[16] G. Candea, E. Kiciman, S. Zhang, P. Keyani, and A. Fox. Jagr: An autonomous self-recovering application server. In *Autonomic Computing Workshop. 2003. Proceedings of the*, pages 168–177. IEEE, 2003.

[17] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, Nov. 2002.

[18] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo. A survey of software aging and rejuvenation studies. *J. Emerg. Technol. Comput. Syst.*, 10(1):8:1–8:34, Jan. 2014.

[19] S. Garg, A. Puliafito, M. Telek, and K. S. Trivedi. Analysis of software rejuvenation using markov regenerative stochastic petri net. In *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, pages 180–187. IEEE, 1995.

[20] X. Ge, H. Vijayakumar, and T. Jaeger. Sprobes: Enforcing kernel code integrity on the trustzone architecture. *arXiv preprint arXiv:1410.7747*, 2014.

[21] M. Grottke, R. Matias, and K. S. Trivedi. The fundamentals of software aging. In *2008 IEEE International Conference on Software Reliability Engineering Workshops (ISSRE Wksp)*, pages 1–6, Nov 2008.

[22] D. Halperin, T. S. Heydt-Benjamin, B. Ransford, S. S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, and W. H. Maisel. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 129–142, May 2008.

[23] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: Analysis, module and applications. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pages 381–390. IEEE, 1995.

[24] Intel Corp. Intel trusted execution technology. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/trusted-execution-technology-security-paper.pdf, 2018. Accessed: July 2018.

[25] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, et al. Experimental security analysis of a modern automobile. In *IEEE Symposium on Security and Privacy*, pages 447–462. IEEE, 2010.

[26] Make Linux. Super fast boot of embedded linux. http://www.makelinux.com/emb/fastboot/omap, 2017. Accessed: June 2017.

[27] R. Perez, R. Sailer, L. van Doorn, et al. vtpm: virtualizing the trusted platform module. In *Proc. 15th Conf. on USENIX Security Symposium*, pages 305–320, 2006.

[28] S. Pinto, J. Pereira, T. Gomes, A. Tavares, and J. Cabral. LTZVisor: TrustZone is the Key. In M. Bertogna, editor, *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:22, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[29] Quanser Inc. 3-DOF helicopter reference manual. Document Number 644, Revision 2.1.

[30] Quanser Inc. Q8 data acquisition board. http://www.quanser.com/products/q8, 2016. Accessed: September 2016.

[31] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. L. Griffin, and L. Van Doorn. Building a mac-based security architecture for the xen open-source hypervisor. In *null*, pages 276–285. IEEE, 2005.

[32] N. Santos, K. P. Gummadi, and R. Rodrigues. Towards trusted cloud computing. *HotCloud*, 9(9):3, 2009.

[33] D. Seto, E. Ferreira, and T. F. Marz. Case study: Development of a baseline controller for automatic landing of an f-16 aircraft using linear matrix inequalities (lmis). Technical report, DTIC Document, 2000.

[34] D. Seto and L. Sha. A case study on analytical analysis of the inverted pendulum real-time control system. Technical report, DTIC Document, 1999.

[35] L. Sha. Using simplicity to control complexity. *IEEE Software*, 18(4):20–28, Jul 2001.

[36] P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Verissimo. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):452–465, April 2010.

[37] P. Sousa, N. F. Neves, and P. Veríssimo. Proactive resilience through architectural hybridization. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC '06, pages 686–690, New York, NY, USA, 2006. ACM.

[38] Texas Instruments. Msp-exp430g2 launchpad development kit. http://www.ti.com/lit/ug/slau318g/slau318g.pdf, 2016. Accessed: April 2017.

[39] S. H. Trapnes. Optimal temperature control of rooms for minimum energy cost. Master's thesis, Institutt for kjemisk prosessteknologi, Norway, 2013.

[40] K. Vaidyanathan and K. S. Trivedi. A comprehensive model for software rejuvenation. *Dependable and Secure Computing, IEEE Transactions on*, 2(2):124–137, 2005.

[41] P. Veríssimo. Future directions in distributed computing. chapter Uncertainty and Predictability: Can They Be Reconciled?, pages 108–113. Springer-Verlag, Berlin, Heidelberg, 2003.

[42] P. E. Veríssimo, N. F. Neves, and M. P. Correia. Intrusion-tolerant architectures: Concepts and design. In *Architecting Dependable Systems*, pages 3–36. Springer Berlin Heidelberg, 2003.

[43] P. Wilson, A. Frey, T. Mihm, D. Kershaw, and T. Alves. Implementing embedded security on dual-virtual-cpu systems. *IEEE Design Test of Computers*, 24(6):582–591, Nov 2007.

[44] P. Wilson, A. Frey, T. Mihm, D. Kershaw, and T. Alves. Implementing embedded security on dual-virtual-cpu systems. *IEEE Design & Test of Computers*, 24(6), 2007.

[45] J. Winter. Trusted computing building blocks for embedded linux-based arm trustzone platforms. In *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pages 21–30. ACM, 2008.

[46] M.-K. Yoon, B. Liu, N. Hovakimyan, and L. Sha. Virtualdrone: virtual sensing, actuation, and communication for attack-resilient unmanned aerial systems. In *Proceedings of the 8th International Conference on Cyber-Physical Systems*, pages 143–154. ACM, 2017.

[47] M.-K. Yoon, S. Mohan, J. Choi, J.-E. Kim, and L. Sha. Securecore: A multicore-based intrusion detection architecture for real-time embedded systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 21–32. IEEE, 2013.