2018 9th ACM/IEEE International Conference on Cyber-Physical Systems

Guaranteed Physical Security with Restart-Based Design for Cyber-Physical Systems

Fardin Abdi*, Chien-Ying Chen*, Monowar Hasan*, Songran Liu†, Sibin Mohan*, and Marco Caccamo*
*Department of Computer Science, University of Illinois at Urbana-Champaign, USA
{abditag2, cchen140, mhasan11, sibin, mcaccamo}@illinois.edu
†School of Computer Science and Engineering, Northeastern University, China
liusongran@stumail.neu.edu.cn

***Abstract*—Physical plants that form the core of the Cyber-Physical Systems (CPS) often have stringent safety requirements. Recent attacks have shown that cyber intrusions can result in the safety of such plants being compromised – thus leading to physical damage. In this paper, we demonstrate how to ensure *safety* of the plant even when the system gets compromised. We leverage the fact that due to inertia, an adversary cannot destabilize the physical system (even with complete control of the software) in an instantaneous manner; in fact, it often takes finite (even considerable time). This property, coupled with *system-wide restarts* is used to enforce a secure (and safe) operational window for the system. A hardware root-of-trust, further decreases the ability for attackers to compromise our mechanisms. We demonstrate our approach using two realistic systems – a 3 degree of freedom (3-DoF) helicopter and a simulated warehouse temperature control unit. We also show that our system is robust against multiple emulated attacks – essentially the attackers are not able to compromise the safety of the CPS.**

I. Introduction

Some of the recent attacks on cyber-physical systems (CPS) are focused on causing physical damage to the plants or physical subsystems. Such intruders make their way into the system using cyber exploits but then initiate actions that can destabilize and even break the underlying (physical) systems. This can be particularly problematic since several critical systems fall into this category – automobiles, avionics, power grid substations, industrial controllers, manufacturing systems and medical devices, to name just a few. Attacks on cardiac defibrillators and pacemakers which put patients' lives at risk [23] and intrusions into automotive controllers with the goal of damaging the vehicles or hurting the passengers [25] are few examples among many others. Hence, any damage to such 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 security problems in such devices.

While some of the security threats can be mitigated using solutions aimed at traditional computing systems, there is still no guarantee that an attacker cannot bypass such mechanisms and gain administrative access to the software that controls the system. Once an attacker gains such access, then all bets are off with regards to the safety of the physical system. For instance, the control program can be prevented from running, either completely or even in a timely manner (CPSs often have stringent timing requirements and any delays in sending out actuation commands can be detrimental), sensor readings can be intercepted (either they are 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 in 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 completely compromised*. The main idea of our paper is to carry out consecutive software restorations 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 restorations. In this paper, the interval between consecutive restorations is dynamically calculated in real time, based on the model of the physical plant and its current state. The key to provide such formal guarantees is to make sure that each restoration takes places before an attacker can cause any sort of physical damage.

To further clarify the approach, consider a simple drone example. The base-line safety for a drone is to not crash into the ground which can be translated into: altitude must be positive at all times. Using a physical model of the drone, in Section V-B, we show how to calculate the shortest time that an adversary with full control on all the actuators would need to take the drone into zero altitude (an unsafe state) from its current state (*i.e.,* velocity and altitude). The key is to schedule the next software restoration event such that it takes place before the calculated shortest time for reaching the ground. Under this condition, immediately after the software restoration, despite the potential hijack of the control software, the drone will still be above the ground (safe) and the software will be clean (due to the restoration). At this point, depending on whether the drone was actually compromised or not, it will be either stabilized – by a safety controller – or its normal operation will resume.

Providing formal safety guarantees, even in this simple example, is non-trivial and there are many issues that need to be addressed. For instance, restoration event must be scheduled such that not only the drone must not reach the ground, but also its velocity and level must be such that the controller is able to stabilize the drone, within the limits of drone motors, before it hits the ground. In addition, the software restoration event (in this paper, full system restart and reload) is not instantaneous and takes a non-zero amount of time which needs to be taken into account. Furthermore,

0-7695-6378-3/18/$31.00 ©2018 IEEE
DOI 10.1109/ICCPS.2018.00010

IEEE computer society

we need to ensure that the attacker may not prevent the restoration action under any circumstances. All these issues and a few others are discussed and addressed in this paper. One of the main contributions of this paper compared to many of the existing security approaches is to consider physical stability along with the safety of the system.

The software restoration action we choose in this paper is restarting the system and reloading the uncompromised image of the controller software from a read-only storage. Restarting the system enables us to *(i)* eliminate all the possible transformations carried out by the adversary during previous execution cycle and also *(ii)* provides a window for trusted computation in an untrusted environment which we use to compute the next restart/restoration time (Section V-A). Additionally, *(iii)* most embedded controllers which are the main target of this paper can be restarted very quickly [5]. We use a very simple external HW timer to trigger the system restart at the scheduled time. This simple design prevents the potential adversary from interfering with the scheduled restart event.

One of the design decisions we made in this work is to proactively restart the system after each execution cycle. We believe this is a necessary choice if we want to run the monitoring code in the same computing platform that is subject to security attacks. Otherwise, the monitor needs to run on a separate safe unit (*e.g.,* hardware monitor implemented on FPGA as System Simplex approach[8] would dictate.). An alternative approach could have been to utilize some sort of detection mechanism and restart the system only after a threat is detected. The problem is a perfect intrusion detection mechanism does not exist. With imperfect threat detection, an adversary could crash the plant during a cycle that the detection mechanism fails to identify the intrusion. Hence, our choice to restart proactively in every execution cycle is the result of achieving *guaranteed safety* rather than a *probabilistic* one within the bounds of the specified threat model.

To be specific, in this paper: *(a)* we use proactive system-wide restarts, *(b)* the (software) system state is "cleansed" and reloaded from a trusted copy, *(c)* the restart rate is adjusted, at run-time, based on the dynamics of the physical plant and its *current* state *such that physical safety is guaranteed*, *(d)* we utilize a *safety controller* that can maintain the plant within the required safety margins (more details in Section IV) and *(e)* an *external hardware timer as root-of-trust* to ensure that the restart/reloading process cannot be interfered with by an adversary. In summary, the contributions of the paper 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 cyber unit is under attack.
2) Our design enables trusted computation (Section V-A) in a untrusted environment using system restarts and common-off-the-shelf (COTS) components (timer IC and common embedded boards), without requiring chip customizations or specific hardware features.
3) We demonstrate the effectiveness of our approach against attacks through a prototype implementation for a realistic physical system and a hardware-in-the-loop simulation.

II. Related Work

The problem of preserving safety of physical components despite possible software faults is widely studied in the fault-tolerant CPS literature[1]. Despite the similarities, there are fundamental differences between protecting against faults vs. protecting against a determined attacker. In the following, we briefly review some of these works and explain the differences.

The Simplex architecture [33] is a well known fault-tolerant design for control systems where a high-performance (yet unverifiable) controller is in charge of a safety-critical system. This is achieved by the presence of a high-assurance, formally verified, *safety controller* and a decision module (again, formally verifiable) that can take control and maintain the safety of the plant if the high-performance controller (also referred to as the "complex controller") is in danger of pushing the physical plant beyond a precalculated safety envelope[2]. There are a few variants of Simplex Architecture; some use a varying switching logic [9], [10] and, others utilize a different safety controller [5]. All these papers, however, assume that misbehavior is possible only within a subset of the software system (for instance, they assume that switching unit cannot misbehave). Such an assumption is not valid for systems under attack. Attackers can actively corrupt the verified software components and force them to misbehave. In this paper, our safety guarantees hold assuming attackers can compromise "all" the software layers.

System-Level Simplex [8] is another variation of Simplex Architecture where authors propose to run the Safety Controller and Decision Module on a dedicated hardware to isolate them from fault/malicious activities on the complex unit. With the complete isolation of complex and safety control units and assuming that the safety subsystem cannot be compromised, techniques based on System-Level Simplex [8], [4], [3], [27] can protect the safety of the physical plant when the complex subsystem is under attack. Nevertheless, exercising System-Level Simplex design on most COTS multicore platforms is challenging. Majority of commercial multicore platforms are not designed to achieve strong inter-core isolation due to the high degree of hardware resource sharing. For instance, an attacker in the core with the highest privilege level may compromise power and clock configurations of the entire platform. To achieve full isolation and independence, one has to utilize two separate

[1] 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. There exists another definition for faults that includes physical problems (*e.g.,* broken sensors/actuators/etc) – we do not consider them in this paper.

[2] **Note:** See Section IV for more details about Simplex as well as the references.

boards. Our paper, on the other hand, provides formal safety guarantees with only one computing unit.

Among the above-mentioned Simplex-based works, [3] and [4] specifically propose restarting the complex unit as a means to mitigate the attacks and recover the system. Notice that, restarting the complex unit in these papers is trivial because safety controller that runs on a separate hardware unit is present during the restart and can keep the plant stable. Another Simplex-based work that uses full system-restarts is [5] in which authors only use a single hardware unit. Similar to some of the earlier mentioned works, this work assumes that faults may only occur within the complex controller and/or operating system and, safety controller and decision module are always correct. This assumption does not hold in the presence of malicious adversary and therefore this approach cannot provide physical safety when the system is under the attack.

A recent work [6] investigates frequent restarts and diversification for embedded controllers to increase the difficulty of launching attacks. Despite the conceptual similarity, our works mainly differ in the calculation of restart times. By dynamically calculating the next restart time in each cycle using real-time reachability, we can *guarantee* the safety of the system. Whereas, authors in [6] choose the restart times with an empirical approach without any formal analysis.

The idea of restarting the entire system or its components at run-time is not novel and has been explored in earlier research to battle the problem of *software aging* in two forms of *revival* (*i.e.,* reactively restarting a failed component) and *rejuvenation* (*i.e.,* proactively restarting functioning components). Authors have tried to model failure and faults for client-server type applications trying to find an optimal rejuvenation strategy for various systems with the aim of reducing the expected down time [38], [21], [24]. Researchers have introduced recursively restartable systems for fault-recovery and increased availability of Internet services [13]. The concept of microreboot that is having fine-grain rebootable components and trying to restart them from the smallest component to the biggest one is explored in [14], [16], [15]. In spite of being used for entirely different purposes, these works assert the effectiveness of restarting as a recovery technique. In this context, some rejuvenation schemes [22] tackle software aging problems related to arithmetic issues such as the accumulation of numerical errors in controllers of safety-critical plants. Nevertheless, rejuvenation techniques for safety-critical systems are very limited. A survey on software aging and rejuvenation shows that only 6 percent of the published papers in this area have considered safety-critical applications [19].

The philosophy of our work is similar to that of the works in a trend in systems dependability that applies fault tolerance concepts and mechanisms in the domain of security, intrusion tolerance (or Byzantine fault tolerance) [17], [40]. Authors in these works advocate for designing intrusion tolerant systems rather than implementing intrusion prevention. Many of the works in intrusion-tolerant systems have targeted distributed services where redundancy and replication are possible and the goal is to ensure that a system still delivers its service correctly even if some of its nodes are compromised. Authors in [17] propose proactively restoring the system code of the replicas from a secure source to eliminate potential transformations carried out by an adversary. Using proactive recovery, one can increase the resilience of any intrusion-tolerant replicated system able to tolerate up to f faults/intrusions, as long as no more than f faults occur between rejuvenations. In [39], authors introduce a general hybrid model for distributed asynchronous systems with partially synchronous components, called *wormholes*. In [35], authors propose the use of wormholes as a trusted secure component (similar to our root of trust timer), which proactively recovers the main functionality of the system. Authors suggest that such a component can be achieved using a separate, tamper-proof hardware module (e.g., PC board) where the separation is physical; or it can be implemented on the native hardware, with a virtual separation and shielding implemented in software, between the former and the OS processes. A proactive-reactive recovery approach is proposed in [34] that builds on top of [35] and allows correct replicas to force the recovery of a replica that is detected or suspected of being faulty. Whilst these techniques are useful for some safety-critical applications such as Supervisory control and data acquisition (SCADA), they are not directly applicable to safety-critical control devices. Potentially, a modified version of these solutions might be utilized for designing a cluster of replicated embedded controllers in charge of a physical plant.

III. Applications, Threats and Adversaries

Our technique is designed for end-point control devices that drive a physical system. Other components of CPS that are not directly controlling a physical plant such as monitoring nodes are not the focus of this paper. Our target applications are safety-critical CPS *i.e.,* controlled physical plant with *stringent safety requirements* that need to be respected at *all* times. Such safety requirements for the physical system can be described by defining a connected subset of the state space as an admissible region. The physical plant is always required to operate within those states. If the plant state reaches outside the admissible set, it might result in damage to the physical components or the surrounding environment.

Our proposed approach works for stateless controllers – in this paper, those that only use the current or very recent state of the plant such as PD controller. Upon restart of a stateless controller device, volatile and non-volatile memory (except for the read-only image of the system) are erased. In case of the stateful controllers, a small section of non-volatile memory will be preserved across restarts in which the controller tasks store the necessary state information for future cycles. An attacker might corrupt this values from time to time if they compromise the system. Therefore, a stateful controller must consider this possibility to avoid decreased control performance in cycles following a successful system compromise. In the rest of this paper, we focus on stateless controllers.

A. Adversary and Threat Model

Embedded CPS controllers 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 the ones whose target is to cause damage to the physical plant. In this paper, we assume attackers require an *external interface* such as the network, serial port or debugging interface to intrude into the system and launch the attacks. We assume that attacker does not have physical access to the plant. Once an attacker is in the system, it has full control (root access) over all the software, actuators, and peripherals. In the light of this assumption, immediately after a reboot, as long as the external interfaces of the device (*i.e.,* network and debugging interface) remain disabled, software running on the system is assumed to be uncorrupted.

In this paper we make the following assumptions about the system and capabilities of the adversary:

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 bugs or 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.,* E^{2}PROM). This content is not modifiable at runtime by anyone including adversary. There exists a secure means for updating this image such as requiring physical access for upgrade or using a dedicated interface for image upgrade.

iii) *Integrity of Root of Trust (RoT)*: RoT is an isolated hardware timer responsible for issuing the restart signal at designated times. As shown in Section V-A, 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 system data, hence, attacks that aim to steal secrets, monitor system activity pattern, or violate user privacy are not mitigated. Our approach does not protect from network attacks such as man-in-the-middle or DoS attacks that restrict network access. However, as we show, the physical system remains safe during such attacks.

IV. 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 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 that 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 is denoted by $\mathcal{R}$. $\mathcal{R}$ is a subset of $\mathcal{S}$ such that if the given SC starts operation from $x \in \mathcal{R}$, all future states will remain admissible.*

In other words, the physical plant is considered momentarily safe when its physical state is in $\mathcal{S}$. And, 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 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, we explain one possible way to design a safety controller which is based on solving linear matrix inequalities and has been used to design Simplex systems as complicated as automated landing maneuvers for an F-16 [31].

A. Safety Controller

According to this design approach [32], [31], 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 [32], [31], [33], [9], [5], [8].

Safety constraints, along with the linear dynamics for the system are the inputs to a convex optimization problem. These parameters produce both the control gain 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 $(x^T Px)$ with a negative-definite derivative. As a result, for the states where $x^T Px < 1$, the stability of the linear system is guaranteed using Lyapunov's direct or indirect methods. It follows that the states which satisfy $x^T Px < 1$ are a subset of the safety region. As long as the system's state is inside $x^T Px < 1$ and the $u = Kx$ is the controller, the physical plant will be driven toward the equilibrium point, *i.e.,* $x^T Px = 0$. Since the potential function is strictly decreasing over time, any trajectory starting inside the region $x^T Px < 1$ will remain there for an unbounded time window. As a result no inadmissible states will be reached. Hence, the linear-state feedback controller $u = Kx$ is the SC and $\mathcal{R} = \{x : x^T Px < 1\}$ is the recoverable region. Designing SC in such a way ensures that the physical system would remain always safe [33].

B. Real-Time Reachability

In this paper we use a tool for runtime computation of reachable states of a physical system within a future time as proposed in [9]. This efficient algorithm is specifically designed for *embedded systems with real-time constraints and low computation power*. This approach uses 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 $t_i^{\mathit{crossing}} = \mathit{width}(N[i])/d_i^{max}$ 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 $\mathit{face}_i + d_i^{max} \times t^a$. For further details on inward neighborhood versus outward neighborhoods, and the choosing of neighborhood widths and time steps refer to [9].

In this algorithm a parameter called `reach-time-step` is used to control neighborhood widths. This parameter allows us to tune the total number of steps used in the method, and therefore alter the *total runtime* to compute reachable set. As a result, the time required to compute reachable set can be capped. Moreover, authors have demonstrated that this algorithm is able to generate useful results within very short computation times. In the paper [9], acceptable results are reported that are achieved with computation times as short as $5ms$ on embedded platforms. Therefore, this real-time reachability approach is suitable to be used on real-time embedded systems.

V. 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 the system in any way. Under such setup, 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 set such that, due to physical inertia, the plant cannot reach an inadmissible state by the end of the following SEI. *(ii)* The state of the plant at the beginning of the following SEI will be such that the SC can stabilize the system. Under these conditions, the plant will be safe in between two SEIs (due to condition 1). If 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 system (condition 2).

Hence, we need an approach to enable trusted, secure execution intervals where we can trust the integrity of the code executing on the system (next section). Then, we present how to utilize the physical model of the plant and calculate separation between consecutive SEIs such that we can ensure the above conditions.

A. Secure Execution Intervals (SEI)

The need for SEIs arises from the fact that a mechanism is required to prevent any malicious interference with the tasks that are necessary for providing the safety guarantees. In this work, we choose *full system restarts* and *software reloads* as a mechanism to achieve this goal. This combination ensures that the uncorrupted version of the software is running on the system immediately after restart completes. Furthermore, through disabling all the external interfaces of the system (*i.e.,* those that may be used as an exploit point for external adversary *e.g.,* network), we can eliminate the possibility of adversarial interferences for as long as it is needed. SEIs refer to this isolated intervals of time after each restart.

During a SEI, we perform the tasks whose correct execution is *necessary* for the safety of the plant. Once the execution of these tasks is finished, external interfaces of the controller board are activated and the normal operation resumes.

In addition, we need another mechanism to ensure that *under any circumstances system will be restarted and adversaries cannot prevent it*. Hence, we utilize an isolated HW module in charge of triggering restarts. This module is called hardware root of trust (RoT). RoT is essentially a timer that can send a restart signal to the HW restart pin of the controller board at the scheduled times. It provides an interface that allows the main controller board to set the time of the next restart signal. We refer to this interface by `SetRestartTime`. The only difference of RoT with a regular timer is that it allows the processor to call the `SetRestartTime` interface only *once* after each restart and will ignore any additional calls to this interface. Once the RoT timer is set, adversaries cannot disable it until it has expired and the system has restarted. Figure 1 presents an example of system events.

ROT needs to be secure and incorruptible. Hence, we require hardware isolation (*e.g.,* a standalone timer) and independence from the processor with no connectivity except for the HW interface, `setRestartTime`. In our prototype implementation, RoT is implemented using a simple micro-controller (Section VI).

B. Finding a Safe Restart Time

Figure 1 presents an example of system operation. After each restart, during the SEI, two tasks execute in parallel: SC (to stabilize the physical plant if needed) and `FindRestartTime` (Section V-B) to calculate time of the next restart. If the attacker had been able to compromise the system during the previous execution cycle and tried to push the system towards inadmissible states, the SC will require a longer time to stabilize the plant and SEI will be longer (Second cycle in Figure 1). Otherwise, if no malicious activity had taken place, SEI finishes quickly and normal operation resumes. The *key* idea is that `FindRestartTime` calculates the restart times such that physical plant *cannot* reach an unsafe state until the restart

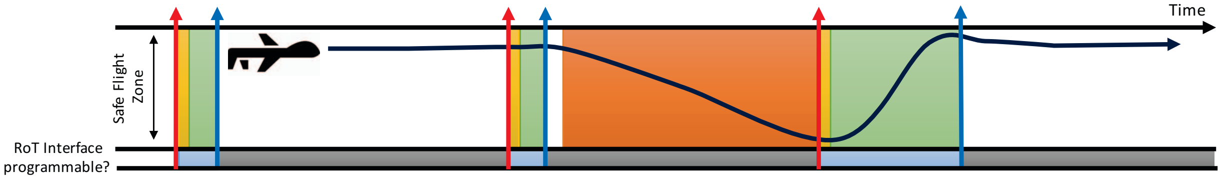


Figure 1: This figure shows an example period of system operation and the sequence of events. Meaning of the colors and signs in this figure are the following: White: Main controller in charge and system is not compromised. Yellow: system is undergoing restarting. Green: SEI is active, SC is running and the next restart time is being calculated. Orange: Adversary is in charge. Blue: RoT's interface is available. Gray: RoT's interface is not available. Red Arrow: RoT triggers a restart. Blue Arrow: SEI ends and the next restart time is scheduled in RoT.

takes place and, at the beginning of next SEI, state is still *recoverable* by the SC.

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 control of the controller. $\mathsf{Reach}_{\leq T}(x, C)$ can be defined as $\bigcup_{t=0}^{T} \mathsf{Reach}_{=t}(x, C)$ *i.e.,* set of all the states reachable within up to T time units. In addition, 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 system to reach state x_2, from state x_1. Using above notations, let us define the set of True Recoverable states for a given physical system.

Definition 3. *True Recoverable states are all the states from which the given SC can eventually push the system inside recoverable region. 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* $\mathcal{T}_\alpha \subseteq \mathcal{T}$.

Let's use T_r to refer to the length of one restart cycle of embedded platform[3]. 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

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

From this definition, 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

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

Equation 3 indicates that if it was possible to calculate $\gamma(x)$ in an SEI, we could have scheduled the consecutive restart to occur at time $\gamma - T_r - \epsilon$ (ϵ is a small positive value). This process would have ensured that by the time the following restart had completed, physical plant was *truly recoverable* (inside region $\mathcal{T}$) and admissible.

Intuitively, γ is a function of the dynamics of the system and limits of the actuators. Unfortunately, it is not usually possible to compute a closed-form representation for $\gamma(x)$. The main reason is computing a closed-form representation for the $\mathcal{T}$ of the given controller is not a trivial problem. Actuator limits is another factor that needs to be taken into the account in the calculation of $\mathcal{T}$. Therefore, in many cases, finding γ would require performing extensive simulations and/or solving numerical or differential equations.

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

$$\begin{aligned} \mathsf{Reach}_{\leq \lambda}(x(t), UC) \subseteq \mathcal{S} \\ \mathsf{Reach}_{=\lambda}(x(t), UC) \subseteq \mathcal{T}_\alpha \end{aligned} \tag{4}$$

Fortunately, it is possible to use real-time reachability to evaluate all the components of Equation (4). The reachable set can be computed up to λ time under an untrusted controller UC to check the first part of the equation (4). The reachable set calculated for $t = \lambda$ can be used as the initial set to perform another reachability computation for α time. In the implementation, we chose a value of α such that real-time reachability can terminate within the assigned deadline. This enables us to check the conditions specified in the definition of $\mathcal{T}_\alpha$ and evaluate the second part of Equation 4.

There are a few points that are worth discussing here. First, the real actions of the adversary are unknown ahead of time. As a result, in the conditions of Equation (4), the reachability of the plant under *all* possible control values needs to be calculated. Hence, 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*. Real-time reachability tool easily enables this computation due to using a box representation for the control inputs. Control inputs are treated as a dimension and are set to the full range of actuators. As a result, the computed reachable set is the states that might be achieved under any of the actuator values. Notice that this procedure does not

[3] T_r is the length of the interval from the restart triggering point until the reboot is completed, filters are initialized and control application is ready to control the system.

impact the time required for reachability computation.

Assuming t is the current time, if conditions of Equation 4 are true for a value of λ from current state $x(t)$, system can be restarted at $t + \lambda - T_r$. However, given more time, we can evaluate the conditions of the theorem for other larger values until we run out of time. Algorithm 1, demonstrates an increasing binary search that tries to find longer restart times within a fixed computation time.

Algorithm 1: Find Restart Time

```
FindRestartTime(x,$\lambda_{\text{candidate}}$)
startTime = currentTime()
RangeStart = 0;
RangeEnd = $\lambda_{\text{candidate}}$ /*Initialize range of binary search for $\lambda_r$*/
while currentTime() - startTime $< T_s$ do
    if conditions of Equation (4) are true for $\lambda_{\text{candidate}}$ then
        $\lambda_{\text{safe}} := \lambda_{\text{candidate}}$
        RangeStart = $\lambda_{\text{safe}}$ ; RangeEnd $= 2 * \lambda_{\text{safe}}$ /* increase the $\lambda_{candidate}$ */
    else
        RangeEnd $= \lambda_{\text{candidate}}$ /* decrease the $\lambda_{candidate}$ */
    end if
    $\lambda_{\text{candidate}} :=$ (RangeStart + RangeEnd)/2
end while
if $\lambda_{\text{safe}} >$ currentTime() - startTime then
    $\lambda_{\text{safe}} = \lambda_{\text{safe}} -$(startTime - currentTime())
    return True , $\lambda_{\text{safe}}$
end if
return False , –
```

When an intelligent adversary compromises the system, it will try to push the system towards inadmissible states as quickly as possible (Example in Figure 1). Although the plant remains safe and recoverable, at the end of the adversarial execution, state of the plant might be very close to the boundaries of the unsafe/inadmissible states. When operating in such proximity to the boundaries of the admissible and inadmissible region, there is a very narrow margin for misbehavior. If attackers take over again while the system is in such risky states, they can easily violate the physical safety. Therefore, in such states, safety controller needs to keep executing longer than usual and push the plant back into inner part of the stable region.

Deciding whether the SC needs to run longer is based on the output of the Function `FindRestartTime` in Algorithm 1. If none of the candidate values satisfy the conditions of Equation (4), then the algorithm will return false as the value. This is an indicator that the plant is too close to the boundary and SC needs to run for another cycle until the system is within its inner stability region. Algorithm 2 describes various stages of the system operation. In Algorithm 2, the number of iterations of the while loop depends on if an adversary was successful in compromising the system in the previous cycle or not.

In our implementation, λ_{init} is set to the middle point of maximum and minimum restart times that are observed during the offline analysis with the model of the physical system (Subsection VI-C of Evaluation). 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, based on the current state of the plant, the associated λ_{init} with the region can be used to initialize the Algorithm 1.

Algorithm 2: System operation in one cycle

```
1: Start Safety Controller.   /* SEI begins */
2: timeFound := False
3: $\lambda_{\text{safe}} = \lambda_{\text{init}}$ /*Initializing the restart time*/
4: while timeFound == False do
5:     $x$ := obtain the most recent state of the system from Sensors
6:     (timeFound, $\lambda_{\text{safe}}$) :=FindRestartTime($x$, $\lambda_{\text{safe}}$)
7: end while
8: Send $\lambda_{\text{safe}}$ to RoT.   /* Set the next restart time. */
9: Activate external interfaces.  /* SEI ends. */
10: Terminate SC and start the main controller.
11: When RoT sends the restart signal to hardware restart pin:
12:     Restart the system
13:     Repeats the procedure from beginning (from Line 1)
```

VI. Evaluation and Feasibility Study

In this section, we aim to evaluate the effectiveness of our solution in protecting from attacks and measure the feasibility of implementing it on real-world physical systems. We choose two target systems for this study; a 3-degree of freedom helicopter [28] and a warehouse temperature management system [37]. The former is a very unstable system with fast-moving dynamics and the latter is a relatively stable system with slower dynamics. We have implemented the controllers for both of these systems on a ZedBoard [7] embedded platform. We performed synthetic attacks on the embedded system and demonstrate that the both systems remain safe and recover from the attacks.

A. Test Bed

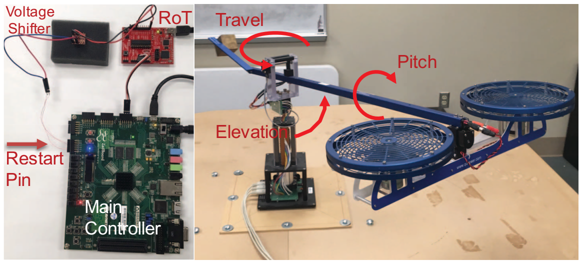


Figure 2: 3DOF helicopter used as the test-bed.

1) 3-Degree of Freedom Helicopter:: 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 [28]. 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 [28] 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]. This platform is repeatedly used by various researchers in the field of CPS [29].

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 $-\epsilon + |\rho|/3 \leq 0.3$ 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 $|v_l| \leq 4$ and $|v_r| \leq 4$ where v_l and v_r are the voltage for controlling left and right motors.

2) Warehouse Temperature System:: This system consists of a warehouse room with a direct conditioner system (heater and cooler) to the room and another conditioner in the floor [37]. The safety goal of this system is to keep the temperature of the room within the range of $[20°C, 30°C]$. Following equations describe the heat transfer between the heater and floor, floor and the room, and room and outside temperature. 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 from the floor heater to the floor and $u_{H/R}$ is the heat 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 parameters used are as following[4]: $U_{R/O} = 539.61 J/hm^2K$, $U_{F/R} = 49920 J/hm^2K$, $m_R = 69.96kg$, $m_F = 6000kg$, floor area $A_{F/R} = 25m^2$, wall and ceiling area $A_{R/O} = 48m^2$, thickness of rock wool, oak and concrete in the wall and floor respectively $0.25m$, $0.15m$ and $0.1m$. Maximum heat generation capacity of the room and floor conditioner is respectively $800J/s$ and $115J/s$. And, maximum cooling capacity of the room and the floor cooler is $-800J/s$ and $-115J/s$.

B. Implementations

In this section, the implementation of the restart-based protection approach for the controller system of the 3DOF helicopter platform and warehouse temperature management system is described. A more detailed description of the implementation is given in our technical report [2]. For the temperature management system, due to the limited access to a real warehouse, the controller interacts with a simulated model of the physical system running on a PC (Hardware-the-loop simulation).

RoT Module:: We implemented the RoT module using a minimal MSP430G2452 micro-controller on a MSP-EXP430G2 LaunchPad board [36]. 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 1MHz (*i.e.,* $1\mu s$ 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 SEI, the RoT acts as an I^2C slave waiting for the value of the restart time. This prevents a compromised system from reprogramming the RoT. 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:: We used a Zedboard [7] to implement the main controller of the 3DOF helicopter and warehouse temperature control system. Zedboard is a development board for Xilinx's Zynq-7000 series all programmable SoC. It contains a XC7Z020 SoC, 512MB DDR3 memory and an on-board 256MB QSPI Flash. The XC7Z020 SoC consists of a processing system (PS) with dual ARM Cortex-A9 cores and a 7-series programmable logic (PL). The processing system runs at 667MHz. In this evaluation, only one of the ARM cores is used while the idle one is not activated. The I^2C and *UART* interfaces are used for connecting to the RoT module and the 3DOF helicopter. 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 on-board 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 main controller runs *FreeRTOS* [1], a preemptive real-time operating system. Immediately after the reboot when the *FreeRTOS* starts, `SafetyController` and `FindRestartTime` tasks are created and executed. `SafetyController` is a periodic task with a

[4] 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 [37].

period of $20ms$ ($50Hz$) and execution time of $100\mu s$. `SafetyController` task has the highest priority. `FindRestartTime` is a single task that executes in a loop and only breaks out when the next restart time is found. It executes at all times except when SC task is running. As soon as `FindRestartTime` calculates the restart time, sends it to the RoT module via the I^2C interface and sets a flag in the system. Based on this flag, `SafetyController` and `FindRestartTime` tasks are terminated and the main control application tasks are launched.

Function `FindRestartTime` is implemented based on the Pseudo-code described in Algorithm 1. Execution of each loop of this function is limited to $50ms$ (*i.e.,* $T_s = 50ms$). To calculate reachability in function `FindRestartTime`, we used real-time reachability tool [9] with the dynamics of 3DOF and the warehouse system. The code and full implementation can be found in the GitHub repository [2].

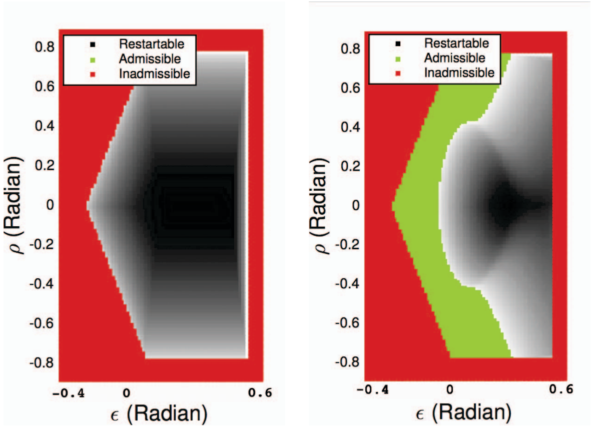
3DOF Helicopter Controller: The main controller unit interfaces with the 3DOF helicopter through a PCIe-based Q8 data acquisition unit[30] and an intermediate Linux-based PC. The PC communicates with the Zedboard through the UART interface. Main controller task is a PID controller whose goal is to navigate the 3DOF to follow a sequence of set points. Control task has a period of $20ms$ ($50Hz$) and at every control cycle, the control task on the controller communicates with the PC to receive the sensor readings (elevation, pitch, and travel angles) and send the next set of voltage values for the motors. The PC uses a custom Linux driver to send the voltages to the 3DOF helicopter motors and reads the sensor values. In our implementation, ZedBoard with FreeRTOS restarts and reboots within $390ms$.

Warehouse Temperature Controller: For this system, due to lack of access to the real physical plant (*i.e.,* a real warehouse) we used a hardware-in-the-loop approach. Here, the PC runs a simulation of the temperature based on the heat transfer model of Equation VI-A2 and VI-A2. The main controller is a PID that adjusts the environment temperature according to the time of the day and preset parameters. The controller is implemented on the Zedboard with the exact same configurations as the 3DOF controller (RoT, serial port connection, I^{2}C interface, $50Hz$ frequency, and restart time). Control commands are sent to the PC, applied to the simulated plant model and the state is reported back to the controller.

C. Safe Restart Time for the Physical Systems

After each system restart, the next restart point needs to be computed and scheduled. Two main factors determine the distance between consecutive restarts; *(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 safe distance between consecutive restarts are calculated and plotted from various states for both of the plants. 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. This is because physical actuators have a limited range and the maximum capacity of the actuators is not enough to cancel the momentum and prevent the plant from reaching the unsafe states. The gray/black area is the operation region (projection of actual operational region into the 2D plane). In this region, the darkness of the color indicates the time between two consecutive restarts if the system is in that specific state. The black points indicate the maximum time and the white points are an indicator of zero time.



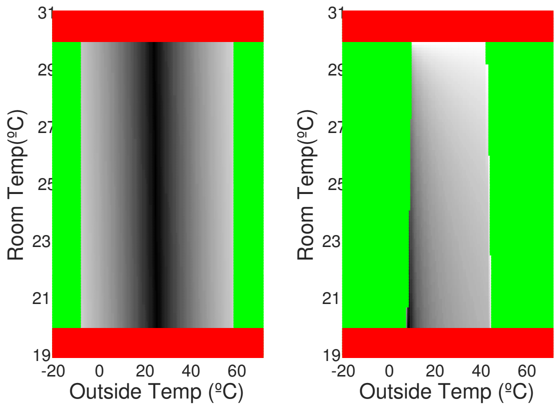
(a) Projection of the state space into the plane $\dot{\epsilon} = 0$, $\dot{\rho} = 0$, $\lambda = 0$, and $\dot{\lambda} = 0.3\text{Radian}/s$

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

Figure 3: Calculated safe restarting time for the 3DOF helicopter system from various states. Darkest black points represent a possible restart time of 1.23 seconds.

In Figure 3, for 3DOF helicopter, the maximum calculated safe restart time (*i.e.,* the darkest points in the gray region) is 1.23 seconds. As seen in the Figure, restart time is maximum in the center where it is farthest away from the boundaries of unsafe states. In Figure 3(b), the angular elevation speed of the helicopter is $\dot{\epsilon} = -0.3\text{Radian}/s$. This indicates that the helicopter is heading towards the surface with the rate of 0.3 Radian per second. As a result, the plant cannot be stabilized from the lower elevations levels (*i.e.,* the green region). Moreover, values of the safe restart times are smaller in Figure 3(b) compared to the Figure 3(a). Because crashing the 3DOF helicopter with the initial downward speed requires less time than the case without any downward angular velocity.

Figures 4(a) and 4(a), plot the calculated safe restart times for the warehouse temperature system. For this system, when the outside temperature is too high or too low, the attacker requires less time to take the temperature beyond or below the safety range. One major difference of this system with the 3DOF helicopter is that due to the slower dynamics,



(a) Projection of states to $T_F = 25^oC$ (b) Projection of states to $T_F = 29^oC$

Figure 4: Safe restarting times calculated for the embedded controller system of warehouse temperature.

safety window has values up to $6235s$. In other words, if the attacker obtains root access when the inside temperature is $25°C$ and outside temperature is $25°C$, and runs the heaters at maximum capacity, it would take at least $6235s$ for the inside temperature to reach above safety range (which is $30°C$). In the warehouse system, the rate of the change of the temperature even when the heater/coolers run at their maximum capacity is slow and hence, the attacker needs more time to take the system to unsafe states.

If above systems are restarted at rates smaller or equal to the calculated safe operation windows, the physical plant is guaranteed to remain safe under an attacker who gains full control immediately after SEI ends[5]. Achieving this goal is very reasonable for systems such as the warehouse temperature control and many other similar systems with applications in Internet of Things (IoT), city infrastructure, and *etc.*. For systems with faster and more unstable dynamics such as the helicopter, the calculated restart times might become very short. The fast restart rate, even though theoretically possible, in practice may create implementation challenges. For instance, initializing a connection with the remote user over the Internet may require more time than the calculated safe restart time. For our particular 3DOF controller implementation, we use the serial port for communication which has a very small initialization overhead. Despite short restart times, we demonstrate that the system remains stable and functional.

D. Impact on Availability and Control

During the restart, the embedded system is unavailable and actuators keep executing the last command that was received before the restart occurred. Results from Section V-B guarantee that such unavailability does not compromise safety and stability of the physical plant. However, low availability may reduce the control performance and slow down the progress of the system towards its goal. In this section, we measure the average ratio of the time that the embedded system is available when restart-based protection is implemented.

The ”availability” of the controller is the ratio of the time that the main controller is running (not rebooting and not in SEI) to the total operation time of the system. In every restart cycle, availability is $(\delta_r)/(\delta_r + T_s + T_r)$ where δ_r is the safe restart window (main controller is active during δ_r), T_s is the length of SEI, and T_r is the reboot time of the embedded platform. Value of δ_r is not fixed and changes according to the state of the plant. To measure availability, we divide the state space into three sub-regions based on how often the physical plant is expected to operate in those regions. The availability values computed for the states in the most common region are assigned a weight of *1*, second most common region a weight of *0.5* and the least common region a weight of *0.3*. These weights are calculated by the operating system in normal condition and recording the trace of the system. Then we measured the number of the data points in each region and assign weights according to this number. We calculated the expected availability of the controller for all the states in the operational range of the system. At the end, the weighted average of the availability was calculated according to the aforementioned weights. For the value of the T_r, we used $390ms$ for the reboot time of the 3DOF embedded controller (measured on the ZedBoard [7] used in the implementation of the controller in next section) and $10s$ for reboot time of the temperature controller embedded system (we assumed platform with embedded Linux OS). The ranges for the regions and obtained availability results are presented in Table I.

From these results, the impact of our approach on the temperature control system is negligible. However, there is a considerable impact on the availability of the helicopter controller due to frequent restarts. Notice 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 approximated upper bound on the impact of our approach on controller availability among all the systems. In the next section, we demonstrate that, despite the reduced the availability, the helicopter and temperature plants remain safe and make progress. Reduced availability of the controller is the cost to pay for safety and can be measured ahead of time by designers to evaluate the trade-offs.

E. Attacks on the Embedded System

To evaluate the effectiveness of the proposed approach, we tested three types of attacks on the implemented controllers of the 3DOF helicopter (with the actual plant) and one attack on the hardware-in-the-loop implementation of the warehouse temperature control system. In these experiments, our focus is on the actions of the attacker after the breach into the system has taken place. Hence, the breaching approach and exploitation of the vulnerabilities are not a

[5] Recall that the system is assumed to remain trusted during the SEI.

	Regions (From most common to least Common)	Avail.
Temperature Control System	$15<T_O<40$	%99.9
	$0<T_O<15$ or $40<T_O<60$	
	$T_O<0$ or $60<T_O$	
3DOF Helicopter	$-\epsilon+\lvert\rho\rvert<0.1$ & $\epsilon<0.2$ & $\lvert\rho\rvert<\pi/8$	%64.3
	$0.2< -\epsilon+\lvert\rho\rvert<0.1$ & $0.2<\epsilon<0.3$ & $\pi/8<\lvert\rho\rvert<\pi/6$	
	$-\epsilon+\lvert\rho\rvert<0.2$ & $0.3<\epsilon$ & $\pi/6<\lvert\rho\rvert$	

Table I: Weighted average availability of the embedded system.

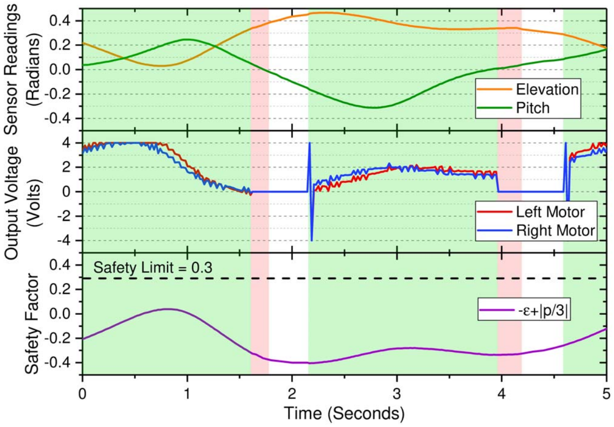


Figure 5: Trace of 3DOF Helicopter during two restart cycles when the system is under worst-case attack that is active immediately after SEI. Green: SEI, red: normal operation (In this case attacker), white: system reboot

concern for these experiments. An attacker may use any number of exploits to get into the controller device.

In the first attack experiment, we evaluate the protection provided by our approach in presence of an attacker who, once activated, is capable of killing the main controller task. The attack is activated at a random time after the end of SEI. We observed that under this attack, the 3DOF helicopter did not hit the surface (*i.e.,* it always remained within the set of admissible states).

In the second attack experiment, attacker replaces the sensor readings of the system with corrupted values with the aim of destabilizing the plant and reducing its progress. The activation time of this attack, similar to the previous one, is dictated by a random variable. Similar to the first attack experiment, the system remained safe throughout the attacks. As expected, reducing the activation time of the attack, the stability of the system and progress towards the set points was also reduced.

In the third attack experiment, we investigate the impact of an attacker who becomes active immediately after the SEI and replaces the original controller with a malicious process that turns off the fans of the helicopter forcing it to hit the surface. This is the worst-case scenario attack because the attacker is active immediately after the SEI. We observed that the system was able to tolerate this attack and did not hit the surface. The trace of the system during a time interval of activity is plotted in Figure 5. In this figure, elevation and pitch along with the control input (voltages of the motor) are depicted. The safety factor in the third graph is obtained from the safety condition for the 3DOF as described in Section VI-A.

From the figure, it can be seen the controller spends most of the time in SEI (red region) or in reboot (white region). This is due to the fact that this extreme-case attack is activated immediately after each SEI and destabilizes the helicopter. By the time that the reboot is complete (end of the white region), system is close to unsafe states. Hence, SEI becomes longer as SC is stabilizing the system. 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.

In the last experiment, an attack is performed on the embedded controller of the warehouse temperature. In this experiment, the outside temperature is set to $45°C$ and initial temperature of the room is set to $25°C$. The attacker takes full control of the system and immediately after the SEI activates both heaters to increase the temperature. We observed that system restarts before the room temperature reached above $30°C$ and after the restart, SC drove the temperature towards the safe range.

VII. Discussion

Suitable Applications: Restart-based protection provides very strong safety guarantees for the CPS. However, it is not necessarily applicable to every system. Applying restart-based protection to very unstable physical systems may require very frequent restarts which will reduce the controller's available time and essentially prevent the system from making any progress. It may also create implementation challenges; a system may require some time to establish a connection over the Internet or to authenticate that may not be possible if the system has to restart very frequently.

Our proposed approach is most suitable for two categories of systems; *first,* CPS for which the control device restart time is very small relative to the speed of the physical system dynamics. Many embedded systems have reboot times that range from tens of milliseconds [26] to tens of seconds which are considered negligible for many applications such as temperature/humidity control in storage/transportation industries, process control in chemical plants, pressure control in water distribution systems, and oxygen level control in patient bodies. *Second,* CPSs in which damage to the plant has more severe consequences than the reduced control performance due to proactive restarts. For instance, a delivery drone carrying a very expensive/sensitive shipment can be considered as examples of this category. Many applications fit in above categories and will benefit from our approach.

Software Diversification After Restarts: Despite the effectiveness of restarting and software reloading in removing the malicious components, restarting does not fix the vulnerabilities of the system that provided the opportunity for the intruder to breach into the system. An attacker may be able to re-launch the same attack after the restart and gain control of the system. Diversifying the software after each reboot can effectively complement our approach to increase the difficulty of launching attacks by canceling the effect of previously learned information. There is a comprehensive body of work on diverse replicas including relocation and/or padding the run-time stack by random amounts [20], rearranging basic blocks and code within basic blocks [20], randomly changing the names of system calls [18], or instruction opcodes [11] , randomizing the heap memory allocator [12] , and randomizing schedule of system [41].

VIII. Conclusion

In this paper, we present an attack-tolerant design for embedded control devices that protect the safety of physical plants in the presence of adversaries. Because of 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* to protect the safety of the physical system. We evaluate our approach on realistic systems and demonstrate its feasibility.

Acknowledgements

The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grant numbers CNS-1646383, SaTC-1718952, and CNS-1423334. We would also like to thank Disha Agarwal for her help in performing some of the experiments. 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, 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.

[4] 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.

[5] 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.

[6] 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.

[7] AVNET. Zedboard hardware user's guide. http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf. Accessed: Apr. 2017.

[8] 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.

[9] 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.

[10] 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.

[11] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanović. Randomized instruction set emulation. *ACM Trans. Inf. Syst. Secur.*, 8(1):3–40, Feb. 2005.

[12] E. D. Berger and B. G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 158–168, New York, NY, USA, 2006. ACM.

[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] M. Chew and D. Song. Mitigating buffer overflows by operating system randomization. Technical report, School of Computer Science, Carnegie Mellon University, 2002.

[19] 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.

[20] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No.97TB100133)*, pages 67–72, May 1997.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[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] S. Mohan, S. Bak, E. Betti, H. Yun, L. Sha, and M. Caccamo. S3a: Secure system simplex architecture for enhanced security and robustness of cyber-physical systems. In *Proceedings of the 2nd ACM international conference on High confidence networked systems*, pages 65–74. ACM, 2013.

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

[29] Quanser Inc. Research papers related to quanser inc. http://www.quanser.com/research_papers/?Sort=Year-DESC. Accessed: June 2017.

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

[31] 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.

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

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

[34] 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.

[35] 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.

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

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

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

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

[40] 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.

[41] M.-K. Yoon, S. Mohan, C.-Y. Chen, and L. Sha. TaskShuffler: A schedule randomization protocol for obfuscation against timing inference attacks in real-time systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12. IEEE, 2016.