Runtime System Support for CPS Software Rejuvenation

Raffaele Romagnoli, *Member, IEEE*, Bruce H. Krogh, *Life Fellow, IEEE*, Dionisio de Niz, *Senior Member, IEEE*, Anton D. Hristozov, *Member, IEEE*, and Bruno Sinopoli, *Fellow, IEEE*

***Abstract*—Software rejuvenation, which was originally introduced to deal with performance degradation due to software aging, has recently been proposed as a mechanism to provide protection against run-time cyber attacks in cyber-physical systems (CPSs). Experiments have demonstrated that CPSs can be protected from attacks that corrupt run-time code and data by periodically restoring the run-time system with an uncorrupted image. Control theoretic and empirical methods have been developed to determine the timing and mode-switching conditions for CPS software rejuvenation (CPS SR) that will guarantee system safety. This article presents the requirements that need to be met by the run-time system to support CPS SR. It also presents an implementation and demonstration of the run-time system for the PX4 autopilot system for autonomous vehicles.**

***Index Terms*—Cyber physical systems, resilience, safety, security, software rejuvenation, UAV.**

I. Introduction

CYBER-PHYSICAL systems (CPSs) rely on the integration of computation, communication, sensing, and control strategies to operate effectively and safely. Cyber security has become a major concern because of the increasing role of CPSs in energy systems, healthcare, transportation, and many other critical applications [1]. Standard methods are typically employed to guarantee security against malware and cyber attacks that can be detected. Run-time cyber attacks that exploit vulnerabilities in communication channels may change the system code or data in ways that make the attack undetectable. Recently, software rejuvenation has been investigated as a method for guaranteeing the safety of CPSs in the presence of such attacks. By periodically refreshing the run-time code and data with an uncorrupted image, malicious changes to the system can be eliminated before they have a disastrous effect on CPS safety and performance. We call this application of software rejuvenation CPS SR.

There two aspects to the dynamic behavior of a CPS: the *physical dynamics*, which characterize the response of the physical system to the control inputs and physical disturbances; and the *cyber dynamics*, by which we mean the dynamic timeing characteristics of the computing and communication systems that host the computer programs implementing the control algorithms.

In this article, we focus on CPSs with controllers that use sensor data to generate the control inputs to guide the behavior of the physical system. Our notion of physical dynamics refers to the physical response of the real system under closed-loop control, rather than just the dynamic models used for control system design. Models of the physical system used for control system analysis and design (often called the *plant*) typically do not represent the details of the cyber dynamics, for example, by assuming ideal synchronous fixed sampling rates for all of the signals of interest and neglecting the times required for computations and communication [2], [3]. These simplified models can usually be justified because of the time-scale separation between the dynamics of inertial physical elements and the high execution rates of cyber components. There can be significant interactions between the cyber dynamics and physical dynamics of a CPS, however, when switching conditions in the cyber system influence, and are influenced by, conditions in the physical system. This is the case with CPS SR, which makes it necessary to coordinate the design of the control system and the design of the hosting run-time system to provide security, timing, and memory management. Studying cyber dynamics is not only fundamental for control but also can be related to those disciplines that study how to develop software as in software cybernetics [4], [5].

The main effect of the cyber dynamics generated by the CPS SR on the control system is management of time in the control algorithm implementation. The CPS SR cycle restarts the controller from a point in time in the past while the physical system keeps moving in the future. This reset of the control software introduces a discrepancy between the control action and the current state of the physical system. We have developed a control theoretic framework for the design of CPS SR [6]–[8]. Further extensions have been proposed for the discrete-time case, the presence of environmental constraints, and persistent

Manuscript received 17 March 2022; revised 10 January 2023; accepted 13 April 2023. Date of publication 21 April 2023; date of current version 6 September 2023. This work was supported in part by the Department of Defense under Grant FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. *(Corresponding author: Raffaele Romagnoli.)*

Raffaele Romagnoli is with the Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15235 USA (e-mail: rromagno@andrew.cmu.edu).

Bruce H. Krogh, Dionisio de Niz, and Anton D. Hristozov are with the Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15235 USA (e-mail: krogh@cmu.edu; dionisio@sei.cmu.edu; adhristozov@sei.cmu.edu).

Bruno Sinopoli is with the Department of Electrical & Systems Engineering, Washington University in St. Louis, St. Louis, MO 63130 USA (e-mail: bsinopoli@wustl.edu).

Digital Object Identifier 10.1109/TETC.2023.3267899

attacks [9], [10]. These articles focus on the theoretical aspect of determining the software rejuvenation timing that ensures safety and control performance for the physical system.

We recently submitted a article that proposes empirical methods to determine CPS SR parameters [11] and illustrates their application to design the CPS SR scheme for a quadrotor using the PX4 flight control software [12]. This article concerns the requirements that must be satisfied by the run-time system to support CPS SR. Specifically, the software rejuvenation is implemented through *micro-reboots*, which consist of two basic operations: *checkpoint* and *rollback*. During the checkpoint the contents of the run-time system memory is saved into a secure memory location, and during a rollback the saved system image is reloaded into the run-time system memory. In previous CPS SR implementations, one checkpoint is taken at the beginning of the mission when it is guaranteed that the software stack is not corrupted by a possible attacker. This is not a viable approach for a controller that makes use of state estimation to compute the control law because, after each software refresh, the gap between the true state of the physical system and the state estimation saved in the initial checkpoint can diverge. To keep this gap bounded, we make use of *multiple checkpoints* where a clean copy of the software stack is saved after each software refresh. In [11], the multi-checkpoint approach has been studied from the control theory side. This article focuses on the run-time system aspects that need to be considered to support multi-checkpoint CPS SR. Specifically, the contributions of this article are:

- run-time system management of multiple checkpoints;
- run-time system requirements for data and software module protection against attack and against software refresh; and
- interactions between the secure kernel and the application-level software.

The following section reviews previous work on CPS SR. Section III describes the requirements that need to be addressed by the run-time system to support CPS SR. As a context for the run-time system implementation, Sect. IV provides an overview of the PX4 autopilot system for autonomous vehicles. Section V presents the implementation of the run-time system features that support the implementation of CPS SR for the PX4 autopilot system, and Sect. VI presents experimental results for a 6-DOF quadrotor controlled by PX4 autopilot system that implements CPS SR. For completeness, subsection VI-D presents the parameter design and performance results from [11]. The concluding section summarizes the contributions of this article and identifies directions for further work.

II. Previous Work

The concept of software rejuvenation was introduced in 1995 to address the problem of so-called software aging, that is, failures that occur when a running program encounters a system state that was not anticipated when the software was designed [13]. The basic idea is to restart the software intermittently at a "clean" state, either through a complete system reboot or by returning to a recent checkpoint. Since the introduction of the concept, there has been considerable research into the development of software rejuvenation strategies [14], and it has become a practical tool for enhancing the robustness of computing systems in a number of applications [15], [16].

Software rejuvenation has been used to mitigate the effects of cyber attacks such as in [17] where mechanisms of attack detection have been studied to activate software rejuvenation for nodes in tactical mobile ad hoc networks (MANETS) in order to eliminate and mitigate the spread of software worms throughout the network. In [18], a dynamical model of the spread of the attack through the network is proposed to better use software rejuvenation to mitigate the attack's effects. In [19], software rejuvenation and aging models have been extended to include load-changing attacks in microgrids. In those techniques, detection and/or models are used to obtain some information about possible ongoing attacks. Predicting or detecting attacks requires reliable software and run-time information to determine if an attack is occurring. These methods are not viable in the presence of run-time cyber attacks that can compromise the code and data. This is the motivation for implementing CPS SR using timing rather than detection to trigger the refresh mechanism.

Run-time cyber-attacks can change the controller code, data, and control flow. Such attacks can change control inputs to the physical system directly or can indirectly change the controls by compromising other modules that support the controls, such as the sensor readings and the state estimation. For these kinds of attacks, software rejuvenation is a very appealing solution since it completely removes the corrupted software and data. In contrast to software aging where mean-time to failure can be the basis for timing software refresh, the frequency of software refresh for CPS must be determined by the length of time a refreshed system can remain viable once it has become vulnerable to attacks.

Arroyo et al. proposed CPS SR based on the idea that a cyber attack requires a minimum time to be effective [20]–[22]. They present experimental results for a simple quadrotor controller and an automotive engine controller to illustrate how control performance and stability are influenced by the frequency of periodic rebooting. Full state measurements are assumed in both examples and only constant setpoint control is considered.

Abidi et al. extended the development of CPS SR by introducing three concepts [23]–[25]. First, the *hardware root of trust* is a secure onboard module that must be available without compromise to implement software rejuvenation. Second, the *secure execution interval* (SEI) is a period during which all external communication is disabled so that no cyber attacks can occur as the software is refreshed. Third, the *safety controller*, which executes immediately following a software refresh and during the SEI, drives the CPS to a safe operating state before restoring communication and returning the system to mission control with vulnerability to cyber attacks. They use a simple, conservative real-time reachability algorithm from [26] to determine the time that can be allowed before the next software refresh and illustrate their approach for a simulated warehouse temperature controller and a bench-top 3-DOF helicopter. Again, only constant setpoint control with full state measurements is considered in these examples.

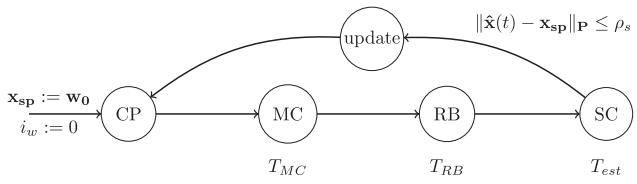


Figure 1. CPS SR mode-transition graph. Unlabeled transitions occur immediately after the operations for the preceding mode are completed, or when the time indicated for the mode has elapsed.

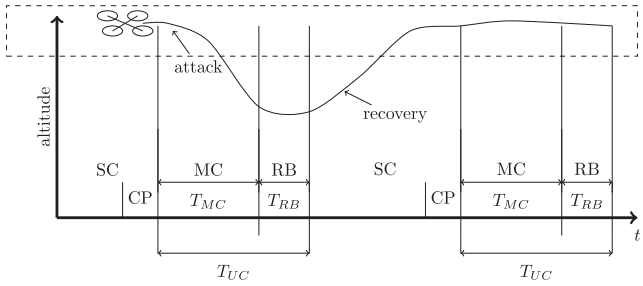


Figure 2. CPS SR timeline.

In [6], the overall approach to software rejuvenation from [24] is adopted, but the timing strategy for software rejuvenation for constant setpoint control is based on analysis of the control problem using Lyapunov functions, invariant sets, and off-line reachability analysis. The control-theoretic approach of [6] is extended to systems that track reference signals in [7]. In [8], the theory is extended to more realistic control problems with bounded disturbances, bounded model uncertainties and general output measurements with bounded noised requiring state estimation. The theoretical results are illustrated with simulation experiments for control of the nonlinear dynamics of a 6-DOF quadrotor.

The CPS SR scheme developed in the above articles is illustrated in Figure 1 and 2, where the modes are defined as follows:

MC *mission control* is the mode where the CPS is executing the code designed to carry out its intended mission under normal operating conditions. During this mode the CPS can exchange information over the network, making it vulnerable to cyber-attacks. Hence, it is allowed to stay in this mode for only a finite period of time, T_{MC}, which is the amount of time the system can remain safe even if there is an (undetectable) attack on the run-time code and data.

RB *rollback* is the mode where the control software is rolled back to the last checkpoint. During this mode, the control inputs are kept constant and equal to the last value set before the rollback, which may be the result of an attack. T_{RB} is the maximum time needed to rollback the control software. Network communication is shut off before the rollback is executed to avoid corruption of the restored run-time image.

SC *safety control* is the mode where the CPS is controlled back to the region where the mission controller can execute safely. It is executed in protected mode, i.e., no external communication is allowed, to guarantee that the information used for safety control is not compromised. The duration of the SC mode is for a duration of at T_{est}, the time required to obtain a sufficiently accurate state estimate, plus any additional time required for the state estimate to be within a range of its current setpoint that is sufficiently close for MC to be implemented (this is the condition indicated on the transition from SC in Figure 1).

CP *checkpoint* represents the mode where a new non-compromised image of the run-time code and data is saved. The system remains in secure execution (no communication) during the CP mode and the control output during CP is the most recent control from the safety controller.

As shown in Figure 1, the CPS-SR cycle is in mode MC for a finite time interval T_{MC}. Then, the control software is rolled back to the previous checkpoint in mode RB. After that, mode SC is active for a duration of at least T_{est}, to guarantee the state estimate is sufficiently good for control, and the transition from mode SC also does not occur until the estimated state is sufficiently close to the current setpoint. (This condition is discussed in Sec. IV.) Before transitioning to mode CP for another checkpoint, the setpoint for tracking control is updated by the *update* function shown as an additional mode in Figure 1 between modes SC and CP.

Figure 2 illustrates the timing of the mode transitions. During T_{MC} and T_{RB} the control input may be compromised by an attack, hence the total time that the CPS can be under the effect of an attack is $T_{UC} \triangleq T_{MC} + T_{RB}$. This time represents the time when the CPS is under uncertain control and it has to be computed. In the case where $T_{UC} \leq T_{RB}$ the problem is not feasible. During RB and SC, the CPS has to be protected, and this time is called *secure execution interval* SEI. The system is vulnerable to attack only during the period T_{MC} when it is in MC mode.

The implementation and demonstration of CPS SR with run-time system support using software-in-the-loop (SITL) simulation for the 6DOF quadrotor is presented in [11], with a focus on the empirical determination of the timing and mode-switching parameters. The results presented in that article are possible thanks to a novel implementation of the software rejuvenation task which considers *multiple checkpoints*. This new implementation introduces new challenges for the run-time system which are not (or just marginally) dealt with in [11]. In this new article, the focus is to present these new challenges and the associated solutions that make software rejuvenation viable to make CPS resilient to run-time cyber attacks.

III. Problem Statement: Run-Time System Requirements

To successfully implement the proposed CPS SR scheme there are several requirements that must be satisfied by the run-time system.

Protection of the timing mechanism: To guarantee safety, RB must be executed after MC has been executed for the period

T_{MC}. The mechanisms that activate the timer and trigger the rejuvenation which runs during MC cannot be compromised by any attack and needs to be protected.

Secure Execution Interval: When the system is in a safety control interval (SEI) there must be no possibility for an attack to change the software. This means that there is no communication through the network while the system is in the RB, SC, and CP modes.

Limits on Control Uncertainty: To guarantee CPS SR feasibility ($T_{UC} > T_{SR}$) the control inputs that can be generated during MC have to be limited. This means that an attacker can use only a limited subset $\bar{U} \subset U$ of all possible control inputs. The control limitations have to be active during MC and have to be implemented at the driver level. It is fundamental that the attacker cannot compromise those limits, so they have to be protected. It is also important to note that these limitations have implications on the actual controller running during MC during normal operations since this controller has to generate control inputs within $\bar{U}$.

Multiple Checkpoints: We consider the case where only the state estimation, denoted by $\hat{\mathbf{x}}(t)$, is available to make decisions in the CPS SR scheme (Figure 1). The run-time system must take a checkpoint before switching to MC and all functions and threads running at that point in time will be running when that checkpoint is restored in the RB mode. The run-time system needs to manage the threads at the time of restoration to assure there are not multiple instances of the threads propagated forward as the images are restored.

Persistent Data: There are data and information that are essential for the control and liveness of the mission, such as the current setpoint $\mathbf{x_{sp}}$ and current mission waypoint $\mathbf{w_i}$. The run-time system must manage this persistent information. It needs to be obtained, protected, and provided correctly to the restored system.

Section V describes how these requirements are met by a run-time system implemented in the context of the PX4 autopilot described in the following section.

IV. Application: PX4 Autopilot System

In general, a CPS controller is implemented by several modules that support the computation of the control action. An example is the PX4 open-source autopilot system for autonomous vehicles, which has a complex modern architecture [12]. We use this widely-used CPS control system as a context to illustrate and demonstrate a run-time system to support CPS SR that meets the requirements described in the previous section. The PX4 software is composed of several modules that implement different parts of the system (e.g., State Machine, Autonomous Flights, Position Control, Sensor Hub, GPS, etc..).[1]

The PX4 software is divided into modules. Each of these modules is run either in separate threads or in thread pools called working queues. The working queues are basically threaded pools where several modules are sharing a single thread. These architectural decisions are favorable for developers and maintainers but pose challenges when rejuvenation is considered. One of the challenges is the state of each module and how it can affect the operation of the autopilot. Another challenge is the queuing in the publish-subscribe μORB module, described below. Finally, due to the threading model and implementation of the modules, there are no timing guarantees.

A. State of Modules Versus State of the System

PX4 has a central message broker module called μORB. Its function is to store and dispatch messages between modules. Each module can publish messages and can subscribe to different messages. This approach is asynchronous and does not depend on time. This flexibility means that the μORB module has to queue messages and store them until all subscribers have received them. One issue that exists with this model is that the software rejuvenation checkpoint and refresh operations can happen at any time while a message is queued. The queued messages, therefore, act as some form of a global state. There is a probability that some messages can be lost while being queued depending on the timing of the checkpoint. Many of the messages are periodic, however, and even if one message is skipped the next messages can compensate by delivering fresh data to the subscribed modules.

Individual states of modules and their effect on the system is a real concern, especially in a loosely coupled architecture like PX4. Each module can have a substantial state in the form of many variables that get updated constantly. Once the system is refreshed the modules are recovered to their state at the moment of the checkpoint. While the state of the software can be rebooted, the state of the physical system $\mathbf{x}(t)$ keeps moving and after RB, so there is a discrepancy between the state of the physical system represented in the restored software and the actual state of the physical system.

The (physical) state of the system can also deviate from its desired value due to software rejuvenation. PX4 has a module `ekf2` that implements an Extended Kalman Filter that provides an estimation $\hat{\mathbf{x}}(\mathbf{t})$ of the actual state of the system by fusing all the sensors measurements [12]. After each RB there is a step change in the estimation error since the value of the state estimation is resumed to the one saved during the last CP. This can affect not only the control performance in terms of stability but may compromise the safety of the CPS SR scheme since $\hat{\mathbf{x}}(t)$ is the only available information used for the CPS SR transitions, and if it is not accurate it may put the system in a dangerous situation.

For example, taking only one checkpoint at the beginning of the mission has negative effects on the estimation error $\mathbf{e}(t) \triangleq \hat{\mathbf{x}}(t) - \mathbf{x}(t)$ during trajectory tracking missions because the actual state $\mathbf{x}(t)$ may move far from the point of the first checkpoint since the physical system is tracking a trajectory. This means that the estimation error can grow after each refresh during the mission and reach values where the switching condition

$$\|\hat{\mathbf{x}}(t) - \mathbf{x_{sp}}\|_{\mathbf{P}} \leq \rho_s \tag{1}$$

[1]See https://docs.px4.io/master/en/concept/architecture.html for a detailed description of the PX4 autopilot architecture.

cannot be guaranteed any more. (1) states that the state estimate is inside an ellipsoid $\mathcal{E}(\rho_s, \mathbf{x_{sp}}) = \{x \in \mathbb{R}^n | (\hat{\mathbf{x}}(t) - \mathbf{x_{sp}})^T \mathbf{P} (\hat{\mathbf{x}}(t) - \mathbf{x_{sp}}) \leq \rho_s^2\}$ centered in $\mathbf{x_{sp}}$ with $\mathbf{P} > 0$ a symmetric definite positive matrix. This condition ensures safety against attack during MC around a new setpoint $\mathbf{x_{sp}}$ [7]. Taking a checkpoint before switching to MC and after having updated the equilibrium point helps to prevent the uncontrolled growth of $\mathbf{e}(t)$.

An additional parameter, ρ_M, is important for the safety and setpoint update. It defines an ellipsoid $\mathcal{E}(\rho_M, \mathbf{x_{sp}})$ that contains $\mathcal{E}(\rho_s, \mathbf{x_{sp}})$ which satisfies the following condition: if $\hat{\mathbf{x}}(t) \in \mathcal{E}(\rho_M, \mathbf{x_{sp}})$ then the system can tolerate attacks for T_{UC}. Hence, if $\hat{\mathbf{x}}(t) \in \mathcal{E}(\rho_s, \mathbf{x_{sp}})$ then we can move $\mathbf{x_{sp}}$ to $\mathbf{x_{sp}}'$ such that $\hat{\mathbf{x}}(t) \in \mathcal{E}(\rho_M, \mathbf{x_{sp}}')$ [7], [11].

Taking multiple checkpoints is necessary, but not a sufficient, to eliminate the growth of the state estimation error. The estimation error can still continue to increase through several CPS SR cycles. As previously mentioned, the state estimator needs time to provide an acceptable state estimation, this means that just right after RB the state estimation may satisfy switching condition (1) to switch to MC while the true state is still far from the true value. This can cause an additional growth in the estimation error after each RB even in the case of no attack. This problem is addressed by introducing a required period T_{est} to spend in SC to recover the state estimate before checking condition (1). The adoption of multiple checkpoints and T_{est} provide bounds on $\mathbf{e}(t)$ that guarantee the existence of a feasible T_{UC} that satisfies (1) [8].

B. Flight Control Modules

We keep the structure of the PX4 architecture, but we use a different position and attitude controller. In version `v1.11.xx` of the PX4 software, the module `mc_att_control` generates the angular rates as reference signals to module `mc_rate_control`, which publishes the actuation controls. We have replaced these modules with `lqr_control`, which implements a linear state feedback algorighm based on linear-quadratic-regulator theory [2], and the setpoint/waypoint update is performed by the module `gen_set_point`, as shown in Figure 3.

At the beginning of the operations, we upload the mission represented by an ordered sequence of waypoints $\mathcal{W}$. This process can be done through the PX4 `commander` routine, which also starts a new thread `gen_set_point start`. At this point, we take the first checkpoint from the command line `checkpoint_p save` and set the timer for the rollback `checkpoint_p do_timer()`. Those functions are in the Checkpoint/Rollback module (described further below), which implements the scheme of Figure 1. After each rollback, this module evaluates the state estimation $\hat{\mathbf{x}}(t)$ provided by the module `ekf2`, which also provides it to the controller `lqr_controller`. The state estimation is also used to update the current setpoint and/or waypoint.

Finally, the Output Driver transforms the control inputs $\mathbf{u}(t)$ into four command signals, one for each propeller. This transformation is inverse of the following *mixer matrix*, which relates the signals $\{U_1, \ldots, U_4\}$ provided to each propeller (values between 0 and 1) to the force/torque control input vector for the model of the physical dynamics, $\mathbf{u}(t) = [F \quad \tau_\phi \quad \tau_\theta \quad \tau_\psi]'$

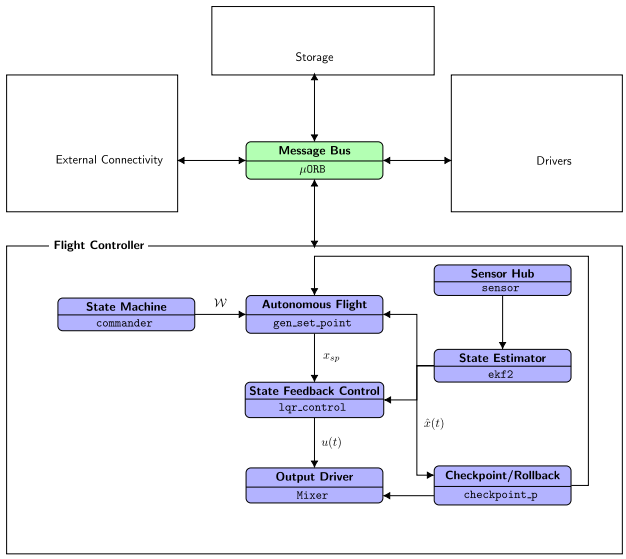


Figure 3. PX4 architecture for the implementation of CPS SR.

$$\begin{bmatrix} F \\ \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} k_1 & k_1 & k_1 & k_1 \\ 0 & -L \cdot k_1 & 0 & L \cdot k_1 \\ L \cdot k_1 & 0 & -L \cdot k_1 & 0 \\ -k_2 & k_2 & -k_2 & k_2 \end{bmatrix} \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix}$$

The constants k_1 and k_2 are the maximum thrust and torque for each propeller, and L is the dimension of the quadrotor. Further details concerning quadrotor design and the controller implementation can be found in [6], [27] and the references in these articles.

C. Data Protection

There are some parts of the PX4 software architecture that should not be compromised by an attack. One of these is the output driver that converts the actuator controls into the signals for each propeller. This conversion is made by the inverse of the mixer matrix shown above. Changing any entry of that matrix is highly dangerous since the drone can irreversibly lose control. In order to guarantee resiliency against attacks, we need to limit the potential impact of an attack by protecting the mixer matrix and limiting the maximum thrust and torque of each propeller. When there is the mode transition to MC and before making the entire system vulnerable, the module Checkpoint/Rollback sends a message that can be only read by the Output Driver module which sets the restrictive limits to the maximum thrust and torques for each propeller. When the system is running in MC, the calls to any function defined in Checkpoint/Rollback module are disabled to assure an attacker cannot exceed the limits on the actuator commands.

D. Secure Execution Interval

During the SEI (which spans the RB, SC, and CP modes) we must guarantee that the controller is working properly. To do so

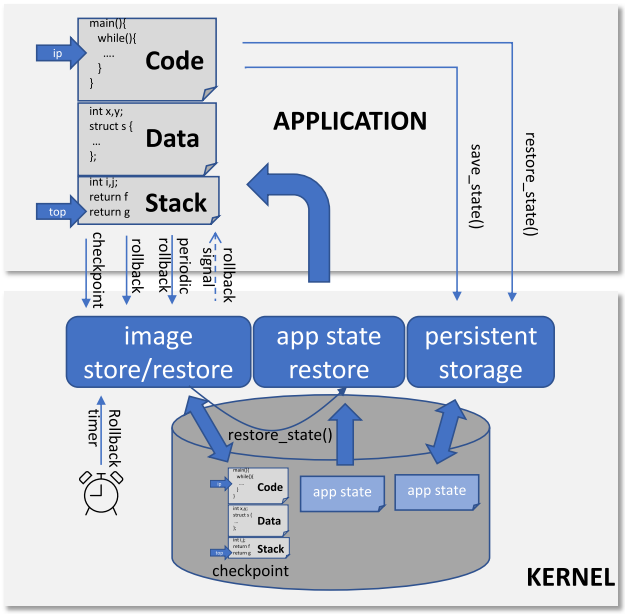


Figure 4. Micro-reboot block diagram.

we need to guarantee that the attacker cannot have the possibility to corrupt the control software. This condition is guaranteed by closing all network communications. The network communication is managed by the External Connectivity modules that can interact with other modules via the publish-subscribe mechanism.

PX4 manages the `data link loss` by implementing a fail-safe mode that forces the quadrotor to move to and land at a "home" location. For the CPS SR implementation, the fail-safe mode is modified in order to guarantee that during SC the system is disconnected from the network and not forced to land.

We assume that the mission starts when the system reaches the first waypoint and ends when it reaches the final waypoint. If we need to protect the takeoff and the return to the home we need to design two specific operation modes that operate while we are disconnected from the network.

V. Run-Time System Implementation

The run-time system we have implemented to support CPS SR hosts the PX4 autopilot control system, which we use to control a quadrotor. Figure 4 shows the micro-reboot scheme used in [11]. From the application level the function `create_checkpoint()` is called to create the image of the code, data, stack segments, and processor registers of the running process. This operation is executed at the kernel level by the *image store/restore* module. The snapshot of the running process is saved in the kernel secure storage and it is restored when the `rollback()` function is invoked. Note that at the application level we can directly invoke this function or set a timer (periodic rollback) at the kernel level to invoke it to restore the image saved during the checkpoint. The restore process occurs in two steps: (i) the *image store/restore* kernel module performs the image restore; and (ii) the *app state restore* kernel module containing code developed by the application programmer is called to restore the application state that needs to be restored before the control is handed over to the application. For the PX4, this code creates a new timestamp correction value inside PX4 that allows it to calculate time intervals correctly after rollbacks.

Once the application has been restored, the kernel generates a `rollback_signal()` which activates a function called `rollback_handler()` in one of the existing threads. This function is useful to handle the code that needs to be executed right after the application has been restored.

A. Periodic and Non-Periodic Rollback

From the application level, it is possible to set a timer for the rollback and make it periodic by calling `do_timer(time, periodic)`. This function sets a timer where, if `periodic==0` the rollback is called only once after the period `time`, otherwise if `periodic==1` it is called periodically every `time`.

The periodic solution offers an advantage in terms of security because the periodic rollback is executed at the kernel level. Thus, the reboot is always guaranteed no matter what happens at the application level. The only issue from the security perspective is to make sure that `create_checkpoint()` and `do_timer(time,1)` cannot be called by an attacker when the system is vulnerable. To avoid this problem these calls are disabled during MC.

The safety of the physical system is guaranteed by the CPS SR scheme presented in Figure 1. This scheme suggests that the periodic rollback cannot be implemented since there is an unknown but finite period of time where the system must stay in SC mode. A periodic rollback can be implemented only in a situation where it is assumed that an attack needs some time before affecting the physical system [22]. In this way, it is possible to set a period that ensures that the physical system will never need to be recovered from the action of an attack. The aperiodic SC mode also offers robustness against trajectory deviations caused by unmodelled dynamics and external disturbances.

In the constant setpoint case, where the initial checkpoint is taken once before the mission, the periodic rollback can be managed entirely at the kernel level because no run-time data needs to be saved and restored. In contrast, the proposed non-periodic CPS SR scheme with updated checkpoints needs run-time information from the application level to update the setpoint information. Hence, the CPS SR scheme is implemented inside the rollback handler, which runs at the application level while the system is disconnected from network communication.

B. Rollback Handler

The software rejuvenation scheme of Figure 1 is implemented by the function `rollback_handler()` presented in Figure 5. The first action is to switch to SC mode. The function `set_SC_mode()` is defined in the module `checkpoint_p` and disconnects the system from the network to protect the system from attacks. The SC mode also enables the Output Driver module to read messages that change the control limits. Once in SC mode, the rollback handler waits for T_{est} before

```
void rollback_handler(){
  set_SC_mode();
  sleep(T_est);
  bool flag=true;
  while(flag){
    x̂:=get_state();
    x_sp:=get_setpoint();
    if (‖x̂ − x_sp‖_P < ρ_s){
      publish_update_sp();
      sleep(tmin);
      flag=false;
      recreate_checkpoint());
      do_timer(T_MC,0);
      set_MC_mode();
  }
}
```

Figure 5. Rollback handler function.

checking condition (1). This allows the `ekf2` module to update the state estimation with new measurements in order to reduce the estimation error. From this point the rollback handler starts to check (1) at each iteration. Since the `while` loop is much faster than the state estimation update, we impose that the check happens after every 2 ms. If condition (1) is true, the function `publish_update_sp()` sends a message to the Autonomous Flight module where `gen_set_point()` is listening to see if the setpoint needs to be updated. The next step is to take a new checkpoint. Waiting for `tmin` after the setpoint update guarantees that the new setpoint is active before `recreate_checkpoint()`. This way, after each rollback the current setpoint is always up to date and we do not need to save the current setpoint and the waypoints in the persistent memory. Before switching to MC, `do_timer()` sets at the kernel level the timer that triggers the next rollback when the time interval T_{MC} is elapsed. Finally, the function `set_MC_mode()` sets the new control limits, disables the calls of functions from `checkpoint_p` module, and restores the network communication.

C. Multiple Checkpoints

The function `recreate_checkpoint()`, which is called inside the rollback handler, introduces a further implementation issue. After each rollback, a new instance of `rollback_handler()` is called while the whole program is resumed. This means that the previous rollback handler resumes starting from `do_timer`. This way, `do_timer` is called recursively, making the time when the next rollback happens unpredictable. In this situation safety cannot be guaranteed by the CPS SR scheme of Figure 1. A modified version of the rollback handler function is presented in Figure 6 to solve this problem.

We introduce a new variable `Semaphore` which is used to decide whether or not the program is executing the code in the current or in the previous instance of the `rollback_handler()`. If we are in the current rollback

```
void rollback_handler(){
  set_SC_mode();
  sleep(T_est);
  bool flag=true;
  bool Semaphore=true;
  while(flag){
    x̂:=get_state();
    x_sp:=get_setpoint();
    if (‖x̂ − x_sp‖_P < ρ_s){
      publish_update_sp();
      sleep(tmin);
      flag=false;
      write_Semaphore(Semaphore);
      recreate_checkpoint());
      Semaphore:=read_Semaphore();
      if (Semaphore==true){
          Semaphore:=false;
          write_Semaphore(Semaphore);
          set_MC_mode();
          do_timer(T_MC,0);
      }
  }
}
```

Figure 6. Rollback handler function.

handler call, the saved `Semaphore` is `true` and `do_timer` can be executed. It is important to note that to implement this mechanism the variable `Semaphore` cannot be rejuvenated, this means that it has to be saved in the persistent memory.

VI. Experimental Results

A. Software in the Loop

Since our lab access was restricted during the COVID pandemic, we used the PX4 Software In The Loop (SITL) platform for our experiments. The experiments were carried out on Intel Core i7-8550 U CPU @ 1.80 GHz, Linux 5.4.0-72-generic, x86_64. The SITL approach allows for the instance of PX4 to be run as one process and the simulator jMAVSim to simulate the quadrotor dynamics as another process with communication between them through sockets. Even when PX4 runs on the simulation computer it still emulates controlling a quadrotor with standard characteristics. Through the class `SimpleSensors.java`, the simulator jMAVSim also implements simulations of the gyro, accelerometer, magnetometer, pressure sensor, and GPS. For each type of sensor, it is possible to define the noise, accuracy, delays, and other parameters that recreate conditions similar to a physical implementation.

For Px4 in SITL mode, the modules in Output Driver are not compiled and the normalized forces and torques are sent directly to jMAVSim. In this situation, we impose the limits directly on the force and torques computed by the controller.

B. Timing Diagram

PX4 publishes new control inputs with a variable sampling time that in general is about 2 ms. When the CPS SR approach

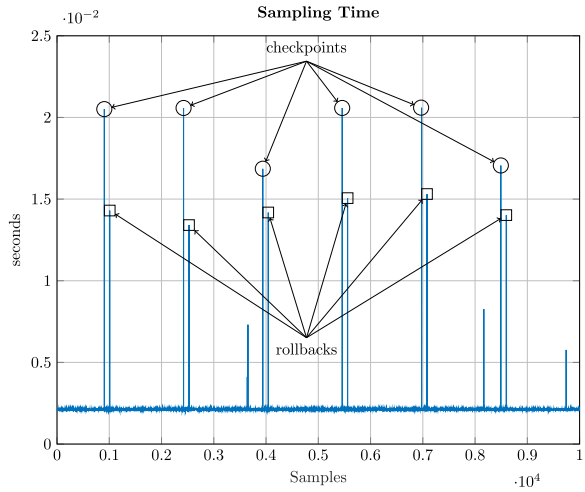


Figure 7. Control inputs updating time.

```
->:rollback signal captured rollback
timestamp: 27766 s, 70180234 ns checkpoint
timestamp: 27768 s, 592161986 ns
->: Semaphore: 0 Exit from rollback
handler!
->: Semaphore: 0 Exit from rollback
handler!
->: Semaphore: 0 Exit from rollback
handler!
->: Semaphore: 0 Exit from rollback
handler!
->: Semaphore: 0 Exit from rollback
handler!
->:Starting rollback_handler...
->:WARN [lqr_control] Recovery check
->:INFO [lq_control] RECOVERY control
ACTIVE
->: ....
->: Do timer ...
->:INFO [checkpoint_p] checkpoint: setting
timer
```

Figure 8. Multiple checkpoint management.

is applied we can observe the duration of the checkpoint and rollback operations.

Figure 7 shows the effective time between two consecutive control input updates during a mission that implements CPS SR. Taking new checkpoints requires more time (≈ 20 ms) than rolling back the software (≈ 15 ms). During this time the control inputs are kept constant, and this may have implications for the control performance. For example, during tracking control, a naive implementation of the checkpoint and rollback procedures can destabilize the physical system. The CPS SR scheme in Figure 1 prevents this possible situation [7].

The constant control inputs during the CP mode are computed by the controller in SC mode, so they are known to be good control values, in contrast to the constant control inputs during the RB mode which may be compromised by an attack. Therefore, the effects of possibly unknown controls during the period T_{RB} need to be considered in the computation of the time interval when the CPS can be vulnerable to attacks T_{UC}, whereas the effects of the constant controls during mode CP will be negligible.

Figure 8 shows the portion of the output of the rollback handler function of Figure 6. Once the rollback signal is captured, the output shows that the control software is still in one of the previous rollback handler calls. Without the introduction of the variable `Semaphore`, `do_timer()` is called several times within the same software rejuvenation cycle. Instead, the function of Figure 6 forces the program to exit from all the previous rollback handler calls (`->: Semaphore: 0 Exit from rollback handler!`) and sets the timer only during the current call of the rollback handler (`->: Starting rollback_handler...`).

C. Experiment Results

Figures 9 and 10 show results from an SITL experiment that implements the CPS SR scheme. The mission is to track a square that has a side length of 1 meter in the $x-y$ plane at an altitude of 3 meters. For this experiment, we use $T_{MC} = 0.3$ s, and

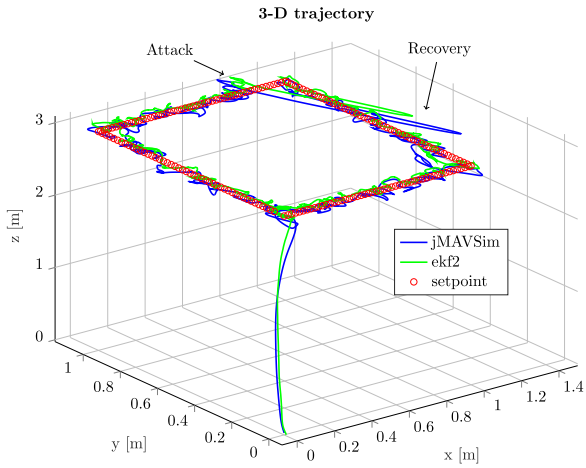


Figure 9. 3D behavior of the system implementing the proposed CPS SR scheme. The blue trajectory represents the actual position of the quadrotor provided by the simulator jMAVSim, and the green one is provided by the PX4 `ekf2`. The red circles are the setpoints computed by the tracking algorithm.

$\rho_M = 0.1$, $\rho_s = 0.0894$, and $T_{est} = 3$ s. Both figures show that the quadrotor is able to accomplish the mission and recover from attacks. Figure 10 shows the trajectory with respect to time. The performance of the control system is affected by the accuracy of the state estimation. Rejuvenating the `ekf2` contributes to increasing the estimation error. For this reason, the introduction of T_{est} is fundamental to guarantee resiliency against attacks and stability of the physical system. T_{est} increases the total time of the mission and makes the quadrotor hover around each setpoint for more time than necessary, generating more oscillations.

In the next subsection, the parameter design and control performance are briefly presented based on the results obtained

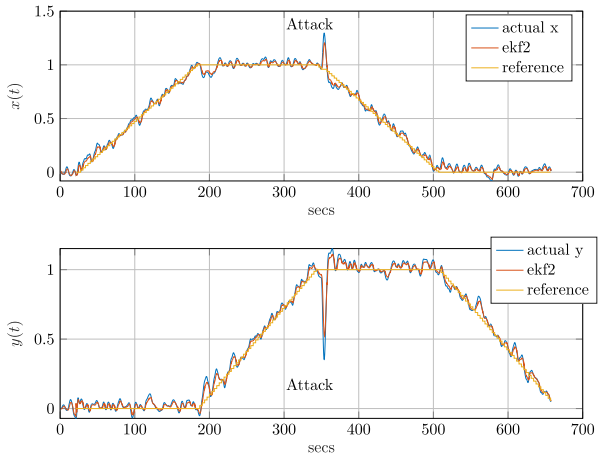


Figure 10. Quadrotor's behavior along the x and y components with respect to time. The blue line represents the actual position provided by jMAVSim, the red line is the position provided by the PX4 module `ekf2`, and the yellow line is the setpoint.

in [11]. The presented values are the same used to design the experiment discussed in this section.

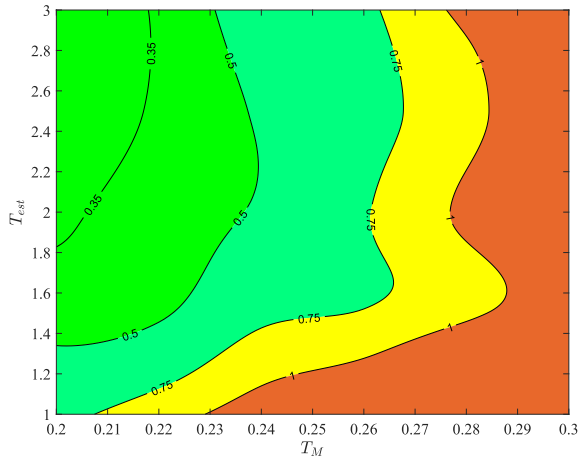
D. Parameter Design and Control Performances

The CPS SR design procedure developed in [11] allows us to find T_{MC} and T_{est} by building a look-up table that contains the maximum value of $\max \|\mathbf{x}(t) - \mathbf{x_{sp}}\|^2_{\mathbf{P}}$ for each pair of T_{MC} and T_{est} values, where $\mathbf{x}(t)$ is the actual state of the quadrotor. In each experiment, the quadrotor is subject to 45 consecutive attacks. The attacks are implemented as minimum time optimal control actions that push the system outside the safety set as fast as possible. Perpetrating consecutive attacks explores the region where an attack can start [11].

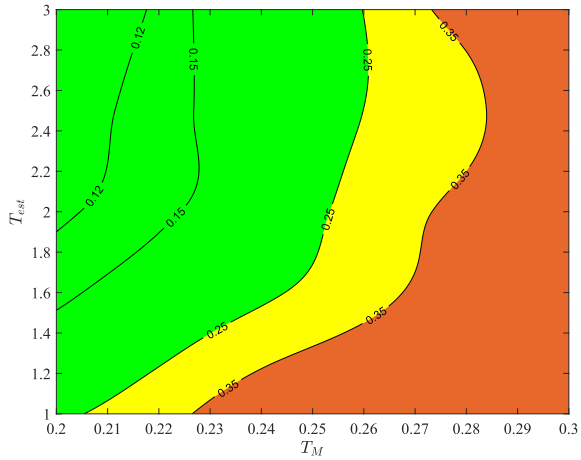
By interpolating the recorded values of $\max \|\mathbf{x}(t) - \mathbf{x}_{sp}\|^2_{\mathbf{P}}$, it is possible to plot the contour regions based on the values of T_{MC} and T_{est} that represent how close the true state of the system $\mathbf{x}(t)$ is to the boundary of the safety region $\|\mathbf{x}(t) - \mathbf{x_{sp}}\|^2_P \leq 1$ (Figure 11(a)). Similarly, it is possible to build the same contour plot with the maximum estimation error $\|\mathbf{x}(t) - \hat{\mathbf{x}}(t)\|^2_{\mathbf{P}}$ (Figure 11(b)).

The results in Figure 11 show that safety is guaranteed for values of T_{est} that go to 3 s, and for values of T_{MC} that go to 0.2 s. These results show how the effects of the attacks in terms of safety are mitigated by reducing T_M and increasing T_{est}. The time needed to refresh the software T_{RB} is already implicitly considered in the experiments. Figure 11(b) shows that the contour of the maximum estimation error follows a similar profile to Figure 11(a), showing that the estimation error has an impact on the safety of the system. Finally, the designer can use Figure 11 to choose T_{est} and T_{MC} to ensure safety and a bounded estimation error. Two additional parameters need to be set, ρ_M and ρ_s, which are kept fixed to $\rho_M = 0.1$ $\rho_s = 0.0894$ for the experiments run to create Figure 11.

The experimental results used for the design of T_{est} and T_{MC} show that safety against attacks can be achieved with a certain



(a) Maximum of $\|\mathbf{x}(t) - x_{sp}\|^2_{\mathbf{P}}$ on the domain $0.2s \leq T_{MC} \leq 0.3s$, $1s \leq T_{est} \leq 3s$.



(b) Maximum of $\|\mathbf{x}(t) - \hat{\mathbf{x}}(t)\|^2_{\mathbf{P}}$ on the domain $0.2s \leq T_{MC} \leq 0.3s$, $1s \leq T_{est} \leq 3s$.

Figure 11. The contour plots show the behavior of (a) the maximum control error and (b) the corresponding estimation error with respect to T_{MC} and T_{est}.

TABLE I
COMPARISON BETWEEN TRAJECTORY TRACKING WITH AND WITHOUT REJUVENATION

Case	T_{MC} [s]	T_{est} [s]	$\bar{d}$ [cm]	max d [cm]	$\bar{T}$ [s]
state estimation no rejuvenation	-	-	6.98	17.73	86.63
state estimation with rejuvenation	0.2	3.0	10.51	27.88	484.25

level of robustness (e.g., an attack is less effective for decreasing values of T_{MC} and increasing values of T_{est}). As pointed out in the previous section, the choice of those parameters affects the overall tracking performance. Table I shows the tracking performance of the system with and without software rejuvenation in absence of attack [11]. Based on the results of Figure 11, the extreme values that provide more robustness against attacks have been chosen as $T_{est} = 0.2$ s and $T_{est} = 3$ s, and both cases

of Table I implement the same tracking algorithm. The mission is the same as described in the previous subsection. The value $\bar{d}$ represents the average distance of the actual drone position to the desired trajectory, $\max \bar{d}$ is the maximum distance, and $\bar{T}$ is the time to complete the mission. In the presence of attacks, CPS SR can guarantee safety during the mission in Figure 9. Still, in the case of no attacks, the tracking performance is lower than the case with no rejuvenation and the total time to complete the mission is longer.

CPS SR is a time-triggered mechanism that does not rely on attack detection because it assumes that any possible information that might be used to detect attacks could be corrupted, making an attack undetectable. When it is possible to implement attack detection, such as the situations considered in [17], [18], the number of software rollbacks will be reduced and the overall tracking performance when there are no attacks can be improved.

VII. Conclusion

This article presents a run-time system to support software rejuvenation to protect CPSs from run-time cyber attacks. The PX4 autopilot example demonstrates that the run-time system can be implemented for complex CPS architectures. The frequency of the reboots and the kernel implementation of the rejuvenation make the solution resistant to a large class of undetectable attacks [28]. The implementation protects against memory and code corruption due to malicious activity. The article presents a specific implementation for the PX4 autopilot, but it can be applied to a large class of systems such as Ardupilot and ROS-based robotic architectures [29]. The viability of the approach has been demonstrated by extensive STIL experiments with a variety of simulated cyber-attacks.

There are several directions for future research and development. Currently, CPS SR relies on a fixed period for mission control, which is determined by worst-case analysis of the reachable physical states for any possible attack beginning immediately when the system becomes vulnerable. Consequently, the system performance is degraded by the checkpoint and software refresh operations even when there are no attacks. Methods for detecting the presence of a possible attack and initiating software rejuvenation only when an attack is detected would make CPS SR much less conservative. This would require secure mechanisms for monitoring the system's behavior. Methods for dealing with the state estimation error due to the time lapse between the checkpoint and image restoration would also be of value to reduce the amount of time required for safety control (SC) before the system can be returned to mission control (MC). Finally, the experience developing the prototype run-time system in this research could guide the development of standard run-time system features to provide commercial support for CPS SR.

Acknowledgment

Carnegie Mellon was registered in the U.S. Patent and Trademark Office by Carnegie Mellon University. Copyright 2022 IEEE. DM22-0241

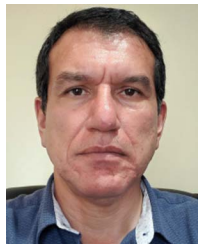
References

[1] A. Platzer, *Logical Foundations of Cyber-Physical Systems*. Berlin, Germany: Springer, 2018.
[2] K. Ogata, *Modern Control Engineering*, 5th Edition. Englewood Cliffs, NJ, USA: Prentice-Hall, 2009.
[3] G. F. Franklin, J. D. Powell, and A. Emami-Naeini, *Feedback Control of Dynamic Systems*, 8th Edition. London, U.K.: Pearson, 2020.
[4] H. Yang, F. Chen, and S. Aliyu, "Modern software cybernetics: New trends," *J. Syst. Softw.*, vol. 124, pp. 169–186, 2017.
[5] K.-Y. Cai, K. S. Trivedi, and B. Yin, "S-ada: Software as an autonomous, dependable and affordable system," in *Proc. IEEE/IFIP 51st Annu. Int. Conf. Dependable Syst. Netw.-Supplemental Volume*, 2021, pp. 17–18.
[6] R. Romagnoli, B. H. Krogh, and B. Sinopoli, "Design of software rejuvenation for CPS security using invariant sets," in *Proc. IEEE Amer. Control Conf.*, 2019, pp. 3740–3745.
[7] R. Romagnoli, B. H. Krogh, and B. Sinopoli, "Safety and liveness of software rejuvenation for secure tracking control," in *Proc. IEEE 18th Eur. Control Conf.*, 2019, pp. 2215–2220.
[8] R. Romagnoli, B. H. Krogh, and B. Sinopoli, "Robust software rejuvenation for CPS with state estimation and disturbances," in *Proc. Amer. Control Conf.*, 2020, pp. 1241–1246.
[9] T. Arauz, J. M. Maestre, R. Romagnoli, B. Sinopoli, and E. F. Camacho, "A linear programming approach to computing safe sets for software rejuvenation," *IEEE Contr. Syst. Lett.*, vol. 6, pp. 1214–1219, 2022.
[10] R. Romagnoli, P. Griffioen, B. H. Krogh, and B. Sinopoli, "Software rejuvenation under persistent attacks in constrained environments," *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 4088–4094, 2020.
[11] R. Romagnoli, B. H. Krogh, D. N. Dionisio, H. AntonD, and B. Sinopoli, "Software rejuvenation for safe operation of cyber-physical systems in the presence of run-time cyber attacks," *IEEE Trans. Control Syst. Technol.*, early access, Jan. 24, 2022, doi: 10.1109/TCST.2023.3236470.
[12] L. Meier, D. Honegger, and M. Pollefeys, "Px4: A node-based multithreaded open source robotics framework for deeply embedded platforms," in *Proc. IEEE Int. Conf. Robot. Automat.*, 2015, pp. 6235–6240.
[13] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton, "Software rejuvenation: Analysis, module and applications," in *Proc. IEEE 25th Int. Symp. Fault Tolerant Comput.*, 1995, pp. 381–390.
[14] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "A survey of software aging and rejuvenation studies," *J. Emerg. Technol. Comput. Syst.*, vol. 10, pp. 8:1–8:34, Jan. 2014.
[15] J. Alonso, A. Bovenzi, J. Li, Y. Wang, S. Russo, and K. Trivedi, "Software rejuvenation: Do IT & telco industries use it?," in *Proc. IEEE 23rd Int. Symp. Softw. Rel. Eng. Workshops*, 2012, pp. 299–304.
[16] M. Grottke, A. P. Nikora, and K. S. Trivedi, "An empirical investigation of fault types in space mission system software," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2010, pp. 447–456.
[17] A. Avritzer, R. G. Cole, and E. J. Weyuker, "Using performance signatures and software rejuvenation for worm mitigation in tactical MANETs," in *Proc. 6th Int. Workshop Softw. Perform.*, 2007, pp. 172–180.
[18] E. Altman, A. Avritzer, R. El-Azouzi, D. S. Menasche, and L. P. de Aguiar, "Rejuvenation and the spread of epidemics in general topologies," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops*, 2014, pp. 414–419.
[19] R. M. Czekster, A. Avritzer, and D. S. Menasché, "Aging and rejuvenation models of load changing attacks in micro-grids," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops*, 2021, pp. 17–24.
[20] M. A. Arroyo, L. Sethumadhavan, and J. Weisz, "Secured cyber-physical systems," U. S. Patent US20170357808A1, Sep. 17, 2019.
[21] M. Arroyo, H. Kobayashi, S. Sethumadhavan, and J. Yang, "FIRED: Frequent inertial resets with diversification for emerging commodity cyber-physical systems," 2017, *arXiv: 1702.06595*.
[22] M. A. Arroyo, M. T. I. Ziad, H. Kobayashi, J. Yang, and S. Sethumadhavan, "YOLO: Frequently resetting cyber-physical systems for security," in *Proc. Auton. Syst.: Sensors, Process., Secur. Veh. Infrastructure*, 2019, pp. 197–206.
[23] F. Abdi, R. Tabish, M. Rungger, M. Zamani, and M. Caccamo, "Application and system-level software fault tolerance through full system restarts," in *Proc. IEEE/ACM 8th Int. Conf. Cyber- Phys. Syst.*, 2017, pp. 197–206.
[24] 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 *Proc. IEEE/ACM 9th Int. Conf. Cyber- Phys. Syst.*, 2018, pp. 10–21.
[25] F. Abdi, C.-Y. Chen, M. Hasan, S. Liu, S. Mohan, and M. Caccamo, "Preserving physical safety under cyber attacks," *IEEE Internet Things J.*, vol. 6, no. 4, pp. 6285–6300, Aug. 2019.

[26] S. Bak, T. T. Johnson, M. Caccamo, and L. Sha, "Real-time reachability for verified simplex design," in *Proc. IEEE Real-Time Syst. Symp.*, 2014, pp. 138–148.
[27] X. Dai, Q. Quan, and K.-Y. Cai, "Design automation and optimization methodology for electric multicopter unmanned aerial robots," *IEEE Trans. Automat. Sci. Eng.*, vol. 19, no. 3, pp. 2354–2368, Jul. 2022.
[28] P. Dash, M. Karimibiuki, and K. Pattabiraman, "Stealthy attacks against robotic vehicles protected by control-based intrusion detection techniques," *Digit. Threats: Res. Pract.*, vol. 2, pp. 1–25, 2021.
[29] M. Lauer, M. Amy, J. Fabre, M. Roy, W. Excoffon, and M. Stoicescu, "Resilient computing on ROS using adaptive fault tolerance," *J. Softw.: Evol. Process*, vol. 30, 2018, Art. no. e1917.

Raffaele Romagnoli (Member, IEEE) received the PhD degree in control system and automation specialized in optimal and robust control system theory from the Universitá Politecnica delle Marche (UNIVPM), Ancona, Italy, in 2015. From 2015 to 2017, he was a postdoctoral researcher with the Department of Control Engineering and System Analysis (SAAS), Université Libre de Bruxelles (ULB), Brussels, Belgium. After being a postdoctoral researcher with the Department of Electrical and Computer Engineering, Carnegie Mellon University (CMU), Pittsburgh, PA, USA, he is now a research scientist with the same institution working on safe and secure control of AI robotics applications and edge computing. He is still currently collaborating with the Software Engineering Institute, (CMU) where he has been working on secure control for cyber-physical systems. His other research interests are nonlinear control, control of biological systems, energy storage (Li-ion batteries), space applications in microgravity conditions, and model inversion.

Bruce H. Krogh (Life Fellow, IEEE) is a professor emeritus of electrical and computer engineering with Carnegie Mellon University, Pittsburgh, PA, USA, and a member of the technical staff of Carnegie Mellon's Software Engineering Institute. He was founding director of Carnegie Mellon University-Africa in Kigali, Rwanda. He is chair of the board of the Kigali Collaborative Research Centre (KCRC) in Rwanda and co-lead of the IEEE Continued initiative to develop IEEE's continuing education resources for technical professionals in Africa. professor Krogh's research is on the theory and application of control systems, with a current focus on methods for guaranteeing safety and security of cyber-physical systems. He was founding editor-in-chief of the *IEEE Transactions on Control Systems Technology*. He is a distinguished member of the IEEE Control Systems Society.



Dionisio de Niz (Senior Member, IEEE) received the master's degree of science in information networking from the Information Networking Institute, Carnegie Mellon University, and the PhD degree in electrical and computer engineering from Carnegie Mellon University. He is a principal researcher and the technical director of the Assuring Cyber-Physical Systems directorate, Software Engineering Institute at Carnegie Mellon University. His research interest includes cyber-physical systems, real-time systems, model-based engineering, and security of CPS. In the Real-time arena he has recently focused on multicore processors and mixed-criticality scheduling and more recently in real-time mixed-trust computing. He co-edited and co-authored the book "Cyber-Physical Systems" where the authors discuss different application areas of CPS and the different foundational domains including real-time scheduling, logical verification, and CPS security.

Anton D. Hristozov (Member, IEEE) received the electrical engineering degree from the Technical University of Sofia, Bulgaria, and the master's degree in telecommunications and information science from the University of Pittsburgh. He is currently working toward the doctoral degree in technology with Purdue University. He holds a position as a research engineer with the Software Engineering Institute Carnegie Mellon University where he deals with scientific experiments for safety assurance of real-time systems. His research interests involve embedded systems, Internet of Things, smart sensors and real time systems. He is a Linux enthusiast and enjoys working with cyber physical systems which use sensors and convert energy in different forms. He has worked on different types of UAVs and UGVs whith focus on mission critical and fault-tolerant software. Anton is generally inderested in runtime techniques for improving security, relaibility and adaptataion of cyber physical systems.

Bruno Sinopoli (Fellow, IEEE) received the PhD degree in electrical engineering from the University of California, Berkeley, in 2005. After a postdoctoral position with Stanford University, he was the faculty with Carnegie Mellon University from 2007 to 2019. In 2019, he joined Washington University in Saint Louis, where he is the chair of the Electrical and Systems Engineering department. He was awarded the 2006 Eli Jury Award for outstanding research achievement in the areas of systems, communications, control and signal processing , U.C. Berkeley, the 2010 George Tallman Ladd Research Award from Carnegie Mellon University and the NSF Career award in 2010. His research interests include the modeling, analysis and design of secure by design cyber–physical systems with applications to energy systems, interdependent infrastructures and Internet of Things.