SchedGuard++: Protecting against Schedule Leaks Using Linux Containers on Multi-Core Processors

JIYANG CHEN, Technische Universität München, Munich, Germany
TOMASZ KLODA, LAAS-CNRS, Université de Toulouse, INSA, Toulouse, France
ROHAN TABISH, AYOOSH BANSAL, CHIEN-YING CHEN, BO LIU, and
SIBIN MOHAN, University of Illinois at Urbana Champaign, Urbana Champaign, Illinois
MARCO CACCAMO, Technische Universität München, Munich, Germany
LUI SHA, University of Illinois at Urbana Champaign, Urbana Champaign, Illinois

Timing correctness is crucial in a multi-criticality real-time system, such as an autonomous driving system. It has been recently shown that these systems can be vulnerable to timing inference attacks, mainly due to their predictable behavioral patterns. Existing solutions like schedule randomization cannot protect against such attacks, often limited by the system's real-time nature. This article presents "*SchedGuard++*": a temporal protection framework for Linux-based real-time systems that protects against posterior schedule-based attacks by preventing untrusted tasks from executing during specific time intervals. *SchedGuard++* supports multi-core platforms and is implemented using Linux containers and a customized Linux kernel real-time scheduler. We provide schedulability analysis assuming the Logical Execution Time (LET) paradigm, which enforces I/O predictability. The proposed response time analysis takes into account the interference from trusted and untrusted tasks and the impact of the protection mechanism. We demonstrate the effectiveness of our system using a realistic radio-controlled rover platform. Not only is "*SchedGuard++*" able to protect against the posterior schedule-based attacks, but it also ensures that the real-time tasks/containers meet their temporal requirements.

CCS Concepts: • **Computer systems organization → Embedded software**; *Real-time operating systems*; Robotic control;

Additional Key Words and Phrases: Response time analysis, Linux containers, Logical Execution Time, security

The material presented in this article is based upon work supported by the Office of Naval Research (ONR) under grant number N00014-17-1-2783 and by the National Science Foundation (NSF) under grant numbers CNS 1646383, CNS 1932529, CNS 1815891, and SaTC 1718952. M. Caccamo was also supported by an Alexander von Humboldt Professorship endowed by the German Federal Ministry of Education and Research. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsors.
Authors' addresses: J. Chen and M. Caccamo, Technische Universität München, München, Germany; emails: jiyang.chen@tum.edu, mcaccamo@tum.de; T. Kloda, LAAS-CNRS, Université de Toulouse, INSA, Toulouse, France; email: tkloda@laas.fr; R. Tabish, A. Bansal, C.-Y. Chen, B. Liu, S. Mohan, and L. Sha, University of Illinois at Urbana Champaign, Urbana, Illinois, USA; emails: {rtabis, ayooshb2, cchen140, boliu1, sibin, lrs}@illinois.edu.
Current address: S. Mohan, George Washington University; email: sibin.mohan@gwu.edu.

2378-962X/2023/02-ART6 $15.00
https://doi.org/10.1145/3565974

ACM Reference format:
Jiyang Chen, Tomasz Kloda, Rohan Tabish, Ayoosh Bansal, Chien-Ying Chen, Bo Liu, Sibin Mohan, Marco Caccamo, and Lui Sha. 2023. SchedGuard++: Protecting against Schedule Leaks Using Linux Containers on Multi-Core Processors. *ACM Trans. Cyber-Phys. Syst.* 7, 1, Article 6 (February 2023), 25 pages.
https://doi.org/10.1145/3565974

1 INTRODUCTION

In the race to achieve full autonomy, the software complexity of vehicles is increasing. Current state-of-the-art vehicles are equipped with driver assistance technology such as adaptive cruise control, lane departure warning, and emergency brake assistance, which all require real-time sensing and actuating. On the other hand, non-safety-critical tasks such as infotainment systems are also essential. As a result, it is common for automobile manufacturers to apply a mixed-criticality design [11, 55] to run safety-critical tasks simultaneously with other non-safety-critical tasks. It is vital to ensure the timing isolation of safety-critical tasks and protect all safety-critical components against failure propagation and unintended use.

It has been shown in the literature that multimedia and connected services featured in vehicles can be exploited as entry points and open different attack surfaces for an intruder to get control of the safety-critical components [1, 31, 32]. Exploiting these attack surfaces gives the intruder a way to get into the system. There have been works in the real-time security community that demonstrated how an intruder once inside the system can launch schedule-based attacks to compromise system security by running along with other trusted and useful tasks [6]. This kind of attack targets the exact time the victim finishes execution or interacts with the outside world through I/O channels. Examples include bias-injection attacks [51], zero-dynamics attacks [21–23, 39, 51, 52], and replay attacks [33]. These attacks steal or compromise the victim task's data integrity by scheduling themselves right after completing the victim task where important crypto-related information might still be available in the shared caches or DRAM. In order to defend against such attacks, two orthogonal approaches, cache-flush-based defense mechanisms [35, 40] and schedule randomization-based defenses, have been proposed [7, 60, 61]. However, recently Nasri et al. [36] suggested that randomization-based approaches sometimes fail to defend against schedule-based attacks as they are incompatible with isolation-based defenses and might result in inflated tasks' worst-case execution time (WCET). Also, cache-flush-based approaches might fall short, especially if the I/O channels are targeted.

We noticed that for schedule-based attacks to be effective, they have to be deployed/executed within a specific time window relative to the execution of the victim task (before, after, in the middle). In this article, we define this time for the attacker task as the *attack effective window (AEW)*. If the attack is launched outside the *AEW*, it is ineffective. An example of the *AEW* has been successfully demonstrated in *ScheduLeak* [6], in which the authors determined the *AEW* for a control output overwrite attack to be 8.3 ms after actuation.

In our previous work [9], we proposed a new systematic approach called *SchedGuard* (schedule guard) that blocks untrusted tasks from running right after the victim task. It is based on the idea of *AEW* and is implemented using cgroup, one of the main techniques used for enabling Linux containers. However, one drawback is that it only works for single-core with one victim task. In this work, we extended the model to cover multi-core systems and modified the implementation to support multiple victims. The main contributions of this work include:

- We provide worst-case response time analysis for trusted and untrusted tasks with multiple victims under *SchedGuard++* on the multi-core platform.

- We implement our proposed *SchedGuard++* approach in the Linux scheduler and demonstrate its effectiveness on commercial-off-the-shelf (COTS) RC cars with multi-core support.
- We evaluate the extended implementation with experiments on a commercial-off-the-shelf multi-core platform.

2 BACKGROUND

2.1 Trusted and Untrusted Tasks

We consider a mixed-criticality system where tasks of different criticalities, such as safety-critical and non-safety-critical, both exist. For example, in an autonomous car system, engine and brake control are considered safety-critical tasks, while infotainment, navigation, logging are seen as non-safety-critical ones. Experience shows that a low-criticality task, such as an entertainment system, is often the entry point to attack safety-critical system components, such as transmission and braking [32]. Mixed-criticality scheduling can prevent high-critical task overruns but disregards security issues. In our approach, we divide the tasks into *trusted* and *untrusted.*

The system developer generally analyzes tasks with high-criticality to ensure that their timing and functional correctness always hold. Under such an assumption, we consider a security model where all high-criticality tasks are considered as *trusted* tasks as they have gone through rigorous timing checks during the development and deployment phase. All other tasks are considered as *untrusted* tasks since they have not been examined thoroughly and can be a potential attacker. Each task is assigned the minimum set of required capabilities following the principle of least privilege.

2.2 Schedule-Based Attacks

An attacker able to determine the precise schedule of system activities (e.g., I/O updates) can launch targeted attacks. Based on the timing relationship between the attacker's execution and the victim task, the schedule-based attack has been categorized into different categories [36]: (a) *Posterior attack model.* Attack launched after the victim has completed its execution; (b) *Anterior attack model.* Attack mounted before the execution of the victim task; (c) *Pincer attack model.* Attack runs before and after the victim task where the attacker analyzes the victim task at load time and monitors its behavior after the victim task has completed execution; (d) *Concurrent attack model.* Attack performed while the victim is running and can be mounted by executing between the execution window of the victim task's job. In this article, we only consider defense against the posterior attack model.

2.3 Attack Effective Window

We assume that an attack must be launched within a specific time interval to be successful. For instance, based on the small UAV case study, the control output overwrite attack [6] can make the control loop unstable if the attack is executed within a time interval of 8.3ms after the actuator's update. Otherwise, if the attack occurs after that time, the system performance can be degraded, but the system stability will be preserved. Here we formally define the *attack effective window* (*AEW*).

Definition 2.1. The attack effective window is the time interval of duration $\Omega_v > 0$ during which schedule-based attacks are effective for task τ_v and ineffective otherwise.

An example of *AEW* is shown in Figure 1. The window is associated with victim task τ_v and is marked in green. τ_h is a higher priority trusted task, and τ_u is a lower priority untrusted task that might be a potential attacker. We define a window as *covered* when trusted tasks utilize all their time slots. In this case, a large part of *AEW* is not covered and leaves a place for the untrusted task to execute. This is considered unsafe.

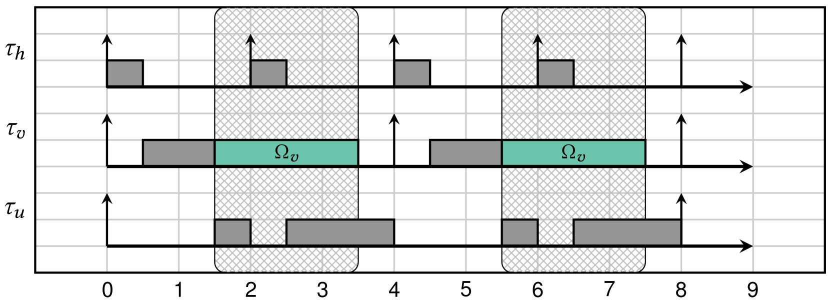

Fig. 1. Attack effective window of τ_v. Task parameters: $\tau_h = (0.5, 2)$, and $\tau_u = (4, 8)$, and $\tau_v = (1, 4)$ with attack effective window of length $\Omega_v = 2$.

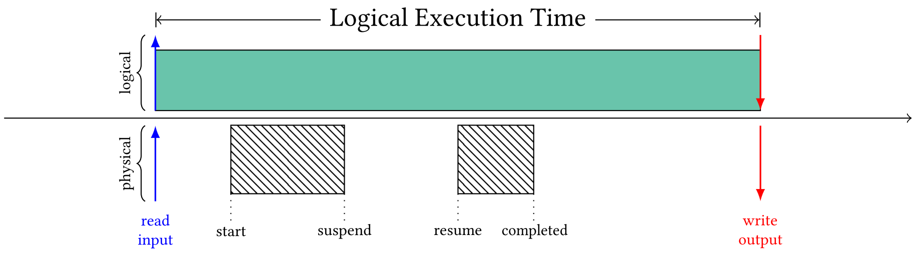

Fig. 2. Logical execution time.

The following subsection identifies the time instants at which the safety-critical tasks update their outputs. An *AEW* starting at such instants can effectively protect the system from the posterior output overwrite attacks.

2.4 Logical Execution Time

The safety-critical tasks deployed in the current generation of vehicles require time-deterministic I/O behavior. Typically, these tasks follow the *Logical Execution Time* (*LET*) model [17]. The *LET* abstraction is a part of the *AUTOSAR* timing extensions [54], has been successfully adopted in the automotive domain [16, 44, 57, 63], and is receiving increasing attention from the research community [12]. The *LET* model enforces a deterministic input/output timing behavior by performing the read and write operations at fixed time instants.

The *LET* is a fixed time interval from task input sensor read to task output actuation update, as shown in Figure 2. Both are static time-triggered events and do not depend on the physical execution of the task. At release time, the task copies its inputs into its local memory and then becomes ready for execution, during which it can work only on the local data stored in the local memory. The task outputs are made available (i.e., copied from the task local to global memory or written to the actuators) at the end of the *LET* interval regardless of how quickly the task is completed. Although the results are not published immediately upon the task completion, the *LET* model brings time- and value-determinism to the task execution as it eliminates jitter and avoids data race conditions. By decoupling the application design from the hardware aspects, the software development process is facilitated throughout the phases of implementation (separation of functionality and timing), verification (time- and value-determinism), and maintenance (platform independence). These are some of the reasons that have proved the *LET* concept to be very attractive for the automotive industry to cope with the increasing complexity of embedded software [12, 14].

Table 1. Notation used in this article

Notation		Description
C_i		task τ_i worst-case execution time
T_i		task τ_i period or minimum inter-arrival time
D_i		task τ_i relative deadline
R_i		task τ_i worst-case response time
$d_{i,j}$		absolute deadline of the j-th job of task τ_i
Ω_v	victim	task τ_v attack effective window length
$hp(i)$		tasks with higher priority than τ_i
$thp(i)$	trusted	tasks with higher priority than τ_i
$uhp(i)$	untrusted	tasks with higher priority than τ_i
V		set of all victim tasks
$deadlines_v(t_1, t_2)$	victim	task τ_v absolute deadlines within $[t_1, t_2]$
$aew_v(t_1, t_2)$	victim	task τ_v trusted execution time within $[t_1, t_2]$
$aew(t_1, t_2)$		trusted execution time within $[t_1, t_2]$ of all victim tasks
$\alpha(\Delta)$		minimal trusted execution time within any interval of length Δ
$\beta(\Delta)$		maximal trusted execution time within any interval of length Δ
AEW		Attack Effective Window
LET		Logical Execution Time

3 SYSTEM MODEL

We consider an electronic control unit (ECU) containing a multi-core processor running a finite set of real-time tasks. Each task is mapped to a single processor and cannot migrate (i.e., partitioned scheduling). Tasks are individually scheduled on each processor by a fixed-priority preemptive scheduling algorithm as applied in AUTOSAR [53] and OSEK/VDX operating system [38]. The task can be periodic or sporadic, and task priorities are unique. Priorities can be assigned using any static priority assignment rule. Table 1 summarizes the task sets notations relative to the task τ_i's priority. Each task τ_i is characterized by a tuple (C_i, T_i, D_i) where C_i is its worst-case execution time (WCET), T_i is its period or the minimum inter-arrival time between releases of its jobs, and $D_i \leq T_i$ its relative deadline that can be less than or equal to the period (i.e., constrained deadlines). All the above parameters are positive integers. The scheduling and context switch overheads are assumed to be included in the task worst-case execution times. Each task gives rise to an infinite sequence of identical jobs (instances) with the first job arriving at time instant 0 (i.e., synchronous arrival) and successive job arrivals separated by, respectively, exactly T_i time units for *periodic* and at least T_i time units for *sporadic* activation pattern model. The worst-case response time R_i for task τ_i is the longest time between the release of a job of the task τ_i until its completion. The task is schedulable if its worst-case response time is less than or equal to its deadline ($R_i \leq D_i$). Each task is either *trusted* or *untrusted*. We do not assume any particular priority order between trusted and untrusted tasks, and their priorities can interleave arbitrarily. Moreover, tasks from different trust levels can be mapped on the same processor, and tasks from the same trust level can span other processors. The task set may contain one or more victim tasks that are prone to the posterior I/O attacks (e.g., control tasks that write the data to the actuators or external devices). We denote a set of all victim tasks by V. All victim tasks fall into the category of trusted tasks. We

Table 2. Linux Capability Attributes

Capability	***Description***
CAP_SYS_NICE	Controls a process permission to change parameters such as scheduling policy, period, and priority.
CAP_SYS_RAWIO	Controls a process permission to read/write to an I/O device

assume that all victim tasks are periodic and follow the *LET* model, which is particularly suitable for control applications [17] (e.g., no jitter and constant sampling period). Each victim task $\tau_v \in V$ is characterized by a constant *AEW* duration of $0 \leq \Omega_v \leq T_v$. The outputs are always updated at the task deadline. An attack is considered successful if the attacker can overwrite the task output (e.g., actuators) within the *AEW* starting from the last task output update: $[d_{v,j}, d_{v,j} + \Omega_v]$ where $d_{v,j} = T_v \cdot (j-1) + D_v$ is the j-th absolute deadline of task τ_v. The tasks that do not fall in the victim task subset can be periodic or sporadic.

4 THREAT MODEL

In our threat model, we assume the attacker's goal is to perform a posterior attack successfully. We categorize the attacker's capabilities into technical ones and operational ones. Technical capabilities rely on the assumptions that the target platform and binary victim applications are available to the attacker. Using such capabilities, the attacker can extract the execution time of the victim task and period on the given platform. One way to extract such information is to run as one of the tasks in the system and rely on techniques such as [6]. Operational capabilities refer to the attacker's ability to implement the attack by exploiting the vulnerable attack surfaces offered as a feature on the current cyber-physical systems. Examples of such features include wireless networks and configurations. It has been demonstrated in the literature that attackers can exploit these communications protocols remotely to install malware and launch the attack [58]. Tencent's recent attack on Tesla demonstrated to hack a Tesla through legacy browser software remotely. The attack was possible because the web browser used an old version of QtWebkit with many vulnerabilities. Exploiting such vulnerability allowed the injection of arbitrary code execution in the center display console of the Tesla.

This article considers the attacks where the attacker only has remote access and exploits the attack surfaces to get into the system. Following such a model, we assume a capability-based security system where a program requires specific "capabilities" to achieve the desired goal. Capabilities are a known concept in Linux-based operating systems. For example, starting with Linux 2.2, superuser privileges are divided into distinct units known as capabilities, and they can be independently enabled and disabled for each process. Only superusers can assign capabilities to other processes. Table 2 summarizes the two crucial capabilities that Linux offers to control a process's behavior in terms of its execution time, period, scheduling, and I/O access. We argue that it is not uncommon for communication modules (such as radio) to have access to hardware I/O, but unusual for them to have the capability to change scheduling parameters. The attacker could gain device I/O access by remotely exploiting the communication module but cannot achieve the capability to modify the scheduling system. Note that although we assume the attacker can access I/O, we do not consider Denial-of-Service (DoS) attack on I/O in this article. The DoS attack can be mitigated, for example, by rate-limiting some system-critical resources [8].

We assume that the attacker cannot exploit kernel vulnerabilities and gain root privilege inside the target system. Although in the aforementioned attack, the security team was able to achieve privilege escalation to gain root access in the system, they attribute their success to the fact that the system used an old version of the Linux kernel (2.6.36) which does not have many exploit mitigation applied. They also commended Tesla's response that patched all known kernel vulnerabilities in the old kernel and introduced new kernels (4.4.35) in newer models. With security concerns

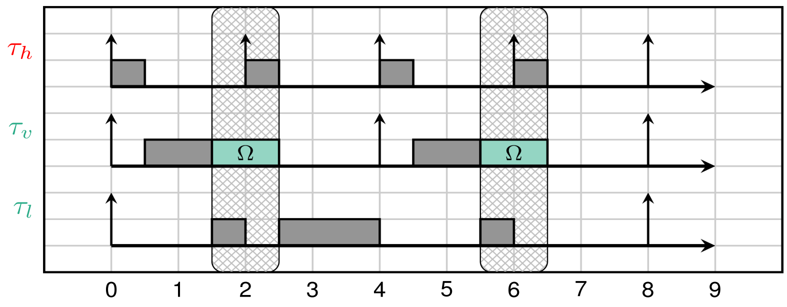

Fig. 3. Schedule of three tasks without any protection.

rising for the Cyber-physical system (CPS), it will become more difficult for attackers to achieve privilege escalation in the newer generation of CPS. However, these do not stop the attacker from getting into the system and launch attacks that do not require root privileges.

The timing of the *AEW* depends on the type of associated schedule-based attack. For the anterior attack, the *AEW* will exist before the execution of the victim task, while for the posterior attack, the *AEW* is after the execution of the victim task. To successfully carry out the attack, the attacker needs to execute during the *AEW* following the execution of the victim task such that the victim's secret can be stolen, corrupted, or overwritten.

As described earlier, the attacker considered in this article can only penetrate the system through remote code execution on the target platform and gain device I/O access but can neither gain scheduling capabilities nor kernel privileges. Hence, we assume the system kernel (including the scheduler) is secure from manipulating an attacker. The attacker aims to successfully initiate a posterior schedule-based attack which means the attacker needs to execute during the *AEW* for the chosen attack.

5 DEFENSE APPROACHES

5.1 Philosophy

The successful execution of an attacker task during *AEW* is crucial to the attack's success. Hence, our defense focuses on using scheduling techniques to block all untrusted tasks from executing during *AEW*. To this end, we define two approaches: (a) paranoid approach and (b) trusted execution approach.

5.2 *Paranoid* Approach

A simple, brute-force approach would be to block all tasks from execution during *AEW*, using the system idle task to occupy this window. This would be equivalent to introducing the Flush task approach to prevent information leakage used by Mohan et al. [35] and Pellizzoni et al. [40]. This can fulfill our defense goal but at the cost of reducing the schedulability of the system. The schedule for three tasks without any protection is shown in Figure 3 and the schedule of the proposed approach for the same three tasks is shown in Figure 4. We consider this the base approach and is the conservative but safe approach.

5.3 Trusted Execution Approach

Blocking all tasks from executing during the window wastes CPU cycles and reduces system utilization. We propose the trusted execution approach to blocking only untrusted tasks during *AEW*, since trusted tasks are considered safe. An example schedule is shown in Figure 5.

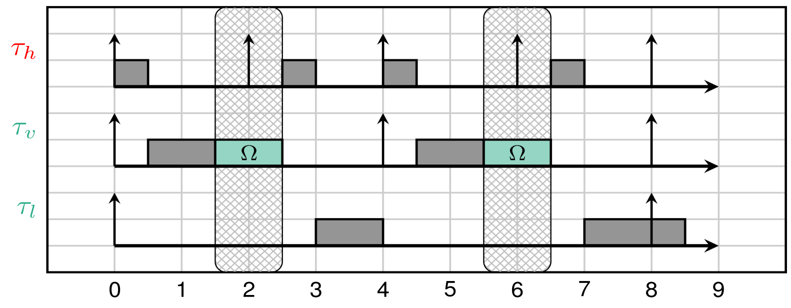

Fig. 4. Schedule of three tasks with with paranoid protection approach.

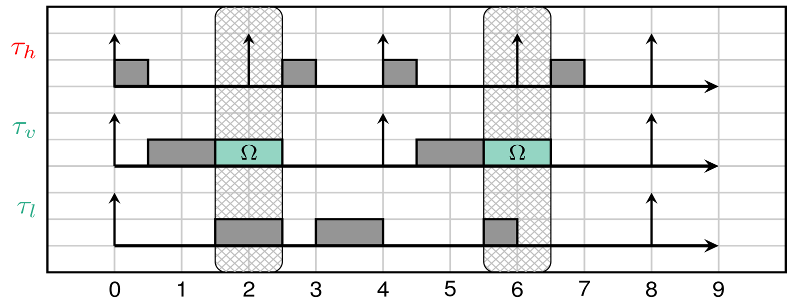

Fig. 5. Schedule of three tasks with trusted protection approach.

6 RESPONSE TIME ANALYSIS

This section provides response time analysis for both paranoid and trusted execution approaches under fixed-priority preemptive scheduling. As the *AEWs* can block the tasks, we first quantify the amount of blocking time (Section 6.1). In the response time analysis for the paranoid approach (Section 6.2), we add a blocking factor for the cumulative length of all *AEWs* during which all tasks must be suspended. For trusted execution approach (Section 6.3), given that only untrusted tasks are blocked, we derive two separate analyses for trusted and untrusted tasks (respectively, Sections 6.3.1 and 6.3.2).

6.1 Trusted Execution Time Estimation

During the *AEW*, the untrusted tasks are blocked, and the trusted tasks can execute without interference from the untrusted tasks. Therefore, the response time analysis for untrusted tasks requires estimating the amount of maximal blocking that these tasks can suffer due to the *AEW*. Likewise, the response time analysis for trusted tasks requires information about the minimal amount of *AEW* during which the trusted tasks can occupy the processor without interference from untrusted tasks.

We first characterize the amount of *AEWs* generated by a single victim task $\tau_v \in V$ where V is the set of all victim tasks. To do this, we introduce an auxiliary set $deadlines_v(t_1, t_2)$ that consists of all absolute deadlines of task τ_v within time interval $[t_1, t_2]$. By abuse of notation, for $t_1 < 0$, we allow the deadlines to be less than zero. Formally, $deadlines_v(t_1, t_2) = \{j \cdot T_v + D_v \text{ for } t_1 \leq j \cdot T_v + D_v < t_2 \text{ and } j \in \mathbb{Z}\}$. Using this definition, we denote a set of all time instants that belong to *AEW* of task τ_v within time interval $[t_1, t_2]$ as follows:

$$aew_v(t_1, t_2) = [t_1, t_2] \cap \{[d_v, d_v + \Omega_v] \text{ for } d_v \in deadlines_v(t_1 - T_v, t_2)\}. \qquad (1)$$

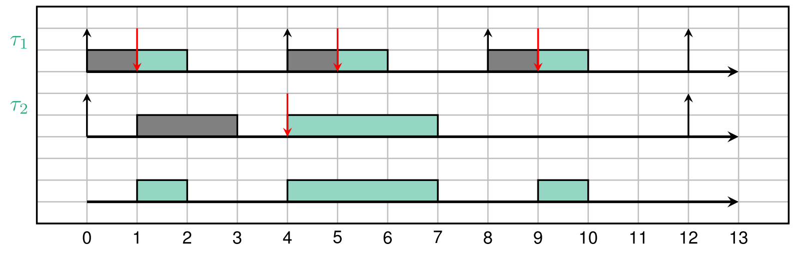

Fig. 6. Trusted attack effective windows overlapping.

In the above formula, we must consider the absolute deadline of the last τ_v instance released before time instant t_1 as its *AEW* might overlap with the beginning of the time interval $[t_1, t_2]$.

Consider example from Figure 6 with two victim tasks. The first victim task τ_1 is characterized by the worst-case execution time $C_1 = 1$, period $T_1 = 4$, deadline $D_1 = 1$ and *AEW* of length $\Omega_1 = 1$. Between time instants $t_1 = 0$ and $t_2 = 12$, there are three τ_1 instances each having its *AEW* within the time interval $[0, 12]$. By Equation (1), $aew_1(0, 12) = [0, 12] \cap \{[d_1, d_1 + 1]$ for $d_1 \in deadlines_1(0-4, 12)\}$ where $deadlines_1(-4, 12) = \{-3, 1, 5, 9\}$. By evaluating Formula (1), we obtain $aew_1(0, 12) = \{[1, 2], [5, 6], [9, 10]\}$ (time interval $[-3, -2]$ is discarded as $[0, 12] \cap [-3, -2] = \{\emptyset\}$; it would overlap only for *AEW* larger than 3; we consider that the remaining part of such *AEW* spanning two task instances is activated for the first time at time instant 0, so the same analysis can be applied for each task instance, arriving at the system start or later time). Please note that in this articler, we aim to protect from the posterior I/O attacks, and consequently, Equation (1) considers that each *AEW* starts at the victim's output update. However, it is possible to define other time-triggered *AEW* patterns and apply the following analysis without any further modifications.

The time reserved for trusted tasks comes from multiple *AEWs* of different victim tasks that can potentially overlap. We define $aew(t_1, t_2)$ as the set of trusted execution time intervals from all victim tasks over time interval $[t_1, t_2]$.

$$aew(t_1, t_2) = \bigcup_{\tau_v \in V} aew_v(t_1, t_2), \tag{2}$$

where V is the set of all victim tasks. Consider the second victim task τ_2 in example from Figure 6. Its worst-case execution time is $C_2 = 2$, deadline $D_2 = 4$ and $\Omega_2 = 3$. In the interval $[0, 12]$, we have $aew(0, 12) = aew_1(0, 12) \cup aew_2(0, 12) = \{[1, 2], [4, 7], [9, 10]\}$ as shown in the bottom of Figure 6.

We now quantify the total amount of trusted execution time. The cumulative length of the trusted execution time within time interval $[t_1, t_2]$ is defined by the following metric on a set $aew(t_1, t_2)$:

$$|aew(t_1, t_2)| \quad = \sum_{[s,f] \in aew(t_1,t_2)} f - s, \tag{3}$$

where s and $f > s$ are, respectively, the start and finish time of a contiguous interval with trusted execution $[s, f] \in aew(t_1, t_2)$. For instance, in the example from Figure 6, we have $|aew(0, 12)| = (2 - 1) + (7 - 4) + (10 - 9) = 5$. We can now use the above definition to get the maximum blocking time that an untrusted task can experience due to *AEWs* and the minimal trusted time during which a trusted task can run without interference. The total minimal amount (length) of trusted execution in any generic interval of length $\Delta > 0$ is represented by:

$$\alpha(\Delta) \;\; = \;\; \min_{t \geq 0} |aew(t, t + \Delta)| \tag{4}$$

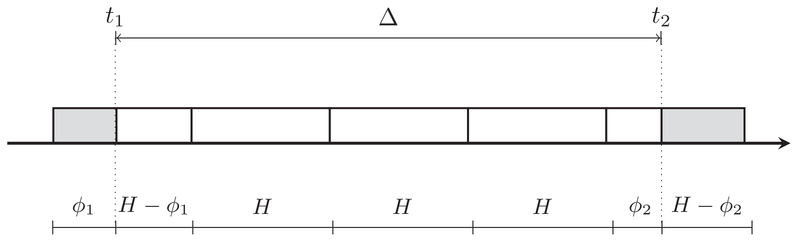

Fig. 7. Trusted execution calculation in an interval covering multiple hyperperiods.

and the total maximum amount of trusted execution in any generic interval of length $\Delta > 0$ by:

$$\beta(\Delta) \ = \ \max_{t \geq 0} |aew(t, t+\Delta)| \tag{5}$$

Coming back to our running example from Figure 6, we can see that $\alpha(4) = 1$ (e.g., from 0 to 4) or $\beta(4) = 3$ (e.g., from 4 to 8).

Finding the exact values of the above functions involves checking the entire hyperperiod (i.e., the least common multiplier of all periods) of victim tasks. For a large number of victims and high values of hyperperiod, it might be computationally intractable. Therefore, we will introduce the upper bounds with a lower computational complexity besides the exact method.

We start with the exact method. In the first stage, we look for the set $aew(0, H)$ where $H = lcm(\{T_v \mid \tau_v \in V\})$ is the least common multiplier of all victim task periods. Please recall that V is the set of all victim tasks in the system, including those running on different cores. By the property of periodic schedule [28], the pattern of $AEWs$ is the same in every hyperperiod H. For each victim task $\tau_v \in V$ we generate a set of non-overlapping intervals with τ_v's attack effective windows $aew_v(0, H)$ accordingly to Formula (1). Next, to obtain $aew(0, H)$, we need to merge the $AEWs$ from all victim tasks (see Formula (2)). This can be done as follows. We first sort all the $AEWs$ by their starting times. Then, we take the window with the earliest starting time and check whether it overlaps with the next window. If so, we merge these windows and repeat the same procedure with the next window given in the starting time non-decreasing order. Otherwise, if the windows do not overlap, we save the current window and check the overlapping for the next two windows. We obtain a list of merged windows by repeating this procedure for all remaining windows. Consider the example in Figure 6. We have $aew_1(0, 12) = \{(1, 2), (5, 6), (9, 10)\}$ and $aew_2(0, 12) = \{(4, 7)\}$. After sorting the windows by their starting times, we obtain the following list $\{(1, 2), (4, 7), (5, 6), (9, 10)\}$. The first and the second windows do not overlap. We save the first window and check whether the second and the third window can overlap. As they do, we merge these two windows. Then, we check the new window against the last one. We save both windows as they do not overlap. Finally, we obtain the set $aew(0, H) = \{(1, 2), (4, 7), (9, 10)\}$. From the set $aew(0, H)$, we can extract subset $aew(t_1, t_2)$ for any $0 \leq t_1 < t_2 \leq H$ by finding the intervals that lay between t_1 and t_2. The $AEWs$ cumulative length $|aew(t_1, t_2)|$ from t_1 to t_2 can be then obtained by summing the lengths of all windows in $aew(t_1, t_2)$. For arbitrary t_1 and t_2 (i.e., spanning for more than one hyperperiod), the cumulative length of the $AEWs$ can be obtained by first checking how many full hyperperiods are executed from t_1 to t_2. Then, we add two remaining parts that do not fit the full hyperperiod execution: the initial one from t_1 to the first hyperperiod start, and the terminal one from the last hyperperiod end to t_2. Figure 7 illustrates our approach. Similar approaches have already been suggested in [2] and [25].

$$|aew(t_1, t_2)| = \begin{cases} |aew(\phi_1, \phi_2)| & \text{if } \phi_1 < \phi_2 \text{ and } t_2 - t_1 \leq H \\ |aew(\phi_1, H)| + n \cdot |aew(0, H)| + |aew(0, \phi_2)| & \text{otherwise} \end{cases}$$

where $\phi_1 = t_1 \bmod H$, $\phi_2 = t_2 \bmod H$, and $n = (t_2 - t_1 - (H - \phi_1) - \phi_2)/H$.

We can therefore refine Equations (4) and (5) for computation of the total, respectively, minimal and maximal amount of trusted execution in any generic interval of length $\Delta > 0$ by considering only $t \in [0, H)$:

$$\alpha(\Delta) = \min_{0 \le t < H} |aew(t, t + \Delta)| \tag{6}$$

$$\beta(\Delta) = \max_{0 \le t < H} |aew(t, t + \Delta)| \tag{7}$$

The method's complexity is exponential in the number of victim tasks since it considers the windows in the entire hyperperiod which grows exponentially as a function of the largest task period. However, the method can be used in many practical applications where the number of victim tasks and the hyperperiod are restricted to be low. Otherwise, we can use the simple bounds on the minimal and on the maximal amount of the trusted execution time computed in a polynomial-time as:

$$\alpha(\Delta) \ge \max_{\tau_v \in V} \left\lfloor \frac{\Delta}{T_v} \right\rfloor \cdot \Omega_v \tag{8}$$

$$\beta(\Delta) \le \min \left\{ \sum_{\tau_v \in V} \left\lceil \frac{\Delta}{T_v} \right\rceil \cdot \Omega_v, \Delta \right\} \tag{9}$$

Formula (8) assumes that all victim task windows overlap with the largest window and Formula (9), on the contrary, assumes that the victims' windows do not overlap.

6.2 Response Time Analysis for Paranoid Approach

We first consider the scheduling problem with the paranoid defense mechanism where none of the tasks is executed within *AEW*. The task under analysis might suffer interference from the *AEWs* and all tasks with higher priority, whether trusted or untrusted. The interference from *AEW* can be seen as a virtual task with the highest priority, and its amount can be quantified using previously introduced Formulas (7) or (9). A safe upper bound on the worst-case response time R_i of task τ_i can be given by the smallest positive integer satisfying the following relation:

$$R_i = C_i + \beta(R_i) + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j \tag{10}$$

The solution can be found through a classic fixed-point iteration that starts with $R_i = C_i$ and terminates when the *LHS* and the *RHS* of the above relation are equal.

6.3 Response Time Analysis for Trusted Execution Approach

This section provides a response time analysis under the Trusted Execution Approach. The trusted tasks are now allowed to execute within the *AEWs* while untrusted tasks are still blocked. We break the analysis into two parts: for trusted tasks (Section 6.3.1) and for untrusted tasks (Section 6.3.2). We assume that the schedulability test is performed in decreasing priority order, starting with the highest priority task first.

6.3.1 *Trusted Task Response Time Analysis.*

We now compute the worst-case response time R_i of trusted task τ_i. Each trusted task is protected from the interference of higher priority untrusted tasks during the *AEWs* and can execute without any blocking during that time. However, the *AEWs* can also have a detrimental effect on the trusted tasks. Each *AEW* might lead to an increased interference of the untrusted task due to the accumulated execution during the *AEW* that must be executed after its end. We break our analysis into two separate cases. We will denote by $R_i^{trusted}$ the worst-case response of task τ_i under the assumption that it is fully executed during the trusted

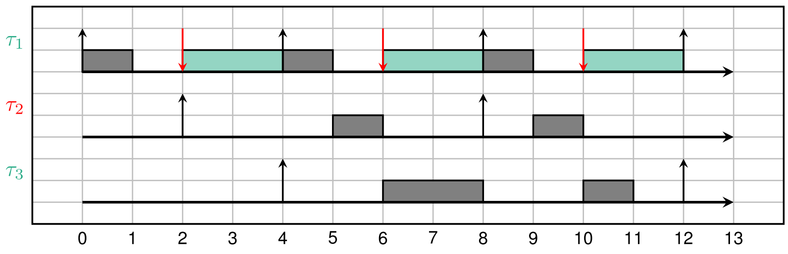

Fig. 8. Interference from higher priority untrusted tasks (critical instant).

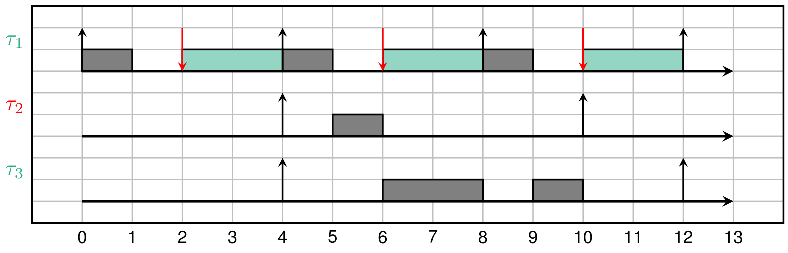

Fig. 9. Interference from higher priority untrusted tasks (critical instant counter-example).

Table 3. Task Parameters for the Example from Figure 8

Task	T	C	D	Ω	R	Type
τ_1	4.0	1.0	2.0	2.0	1.0	trusted
τ_2	6.0	1.0	2.0	2.0	1.5	untrusted
τ_3	8.0	3.0	8.0	-	8.0	trusted

time and by R_i^{normal} under the assumption that it is fully executed during a normal time when trusted and untrusted tasks compete for the processor. We consider $R_i = \min(R_i^{trusted}, R_i^{normal})$ to be the minimum of these two response times.

By blocking the untrusted tasks during the AEWs, the interference of these tasks can increase. Consider the example of three tasks shown in Figure 8. The task parameters are listed in Table 3. The figure shows the critical instant for task τ_3. We note that task τ_2 is not released synchronously with the other tasks. Task τ_2 released at time instant 2 is instantaneously blocked by the trusted task *AEW*. The first instance of τ_2, released at 4, will interfere with τ_3 and the next instance of τ_2, released at 8, will also interfere with τ_3. In the case of synchronous release of all tasks, which is the critical instant for classic preemptive fixed-priority scheduling [29], the second instance of τ_2 will not interfere with τ_3 as shown in Figure 9.

We model the higher priority untrusted task deferred interference with a release jitter. The untrusted task can be considered as a self-suspending task that self-suspends its execution during the *AEWs*. The interference of the self-suspending task τ_j can be modeled by assuming a release jitter equal to the task's latest possible starting time $R_j - C_j$ as proved in [3], [10], and [20]. Let R_i^{normal} be the task τ_i worst-case response time under the assumption that it is fully executed during the normal time when trusted and untrusted tasks compete for the processor. Its value is less than or equal to the smallest positive integer satisfying the following relation:

$$R_i^{normal} = C_i + \sum_{j \in thp(i)} \left\lceil \frac{R_i^{normal}}{T_j} \right\rceil \cdot C_j + \sum_{j \in uhp(i)} \left\lceil \frac{R_i^{normal} + R_j - C_j}{T_j} \right\rceil \cdot C_j \quad (11)$$

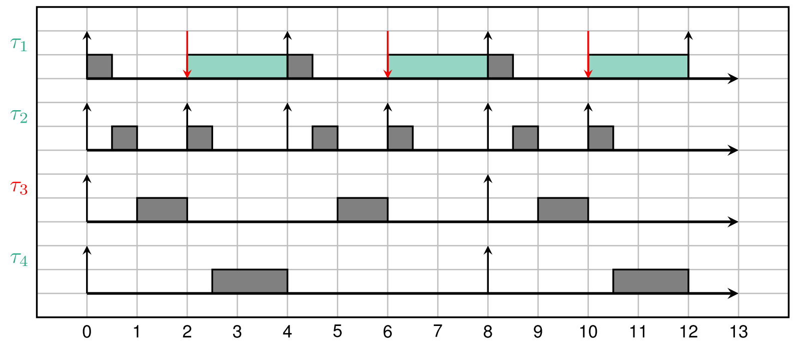

Fig. 10. Attack effective window protection for trusted tasks.

Table 4. Task Parameters for the Example from Figure 10

Task	T	C	D	Ω	R	Type
τ_1	4.0	0.5	2.0	2.0	0.5	trusted
τ_2	2.0	0.5	2.0	-	1.0	trusted
τ_3	8.0	2.0	8.0	-	8.0	untrusted
τ_4	8.0	1.5	8.0	-	4.0	trusted

The time within AEWs is reserved exclusively for the trusted tasks. Consider the example shown in Figure 10 and task set parameters listed in Table 4. Over any time interval of length $\Delta = 4$, at least 2 time units of processor time are reserved exclusively for the trusted tasks. When tasks τ_2 completes its execution during the AEW, task τ_4 can use the remaining window trusted execution time. The following condition checks if there is enough trusted execution time reserved for task τ_i and other higher priority trusted tasks within time interval $R_i^{trusted}$:

$$\alpha\left(R_i^{trusted}\right) \;\geq\; C_i + \sum_{j \in thp(i)} \left\lceil \frac{R_i^{trusted}}{T_j} \right\rceil \cdot C_j \tag{12}$$

To find the solution of the above equation, one can start the iteration with $R_i^{trusted} = C_i$. Then, we set $R_i^{trusted}$ to the *RHS*. If the *RHS* does not increase over two subsequent iterations and the relation is not satisfied, we set $R_i^{trusted}$ to the next value at which the *LHS* increases.

Suppose the trusted execution time is insufficient to complete task τ_i and other higher priority trusted tasks. In that case, we check if the tasks can be executed in the normal mode withstanding the interference from the higher priority untrusted tasks. The worst-case response time R_i of the trusted task τ_i is upper bounded by the smallest value given by Equations (11) and (12):

$$R_i \;=\; \min\left(R_i^{trusted}, R_i^{normal}\right) \tag{13}$$

6.3.2 Untrusted Task Response Time Analysis. An untrusted task τ_i during its execution can be blocked by the *AEWs* and by the higher priority tasks. Since the trusted higher priority tasks can freely execute during the *AEWs*, the part of their interference overlapping with *AEWs* might be ignored. Consider the example from Figure 11 representing a sample schedule of three tasks. The task parameters are given in Table 5. Three instances of task τ_1 (trusted) overlap with the *AEWs* of task τ_2, and as a result, task τ_3 (untrusted) does not suffer any interference from these three τ_1 instances. It is, therefore, unnecessary to take into account the interference from the trusted task instances that overlap with the *AEWs*.

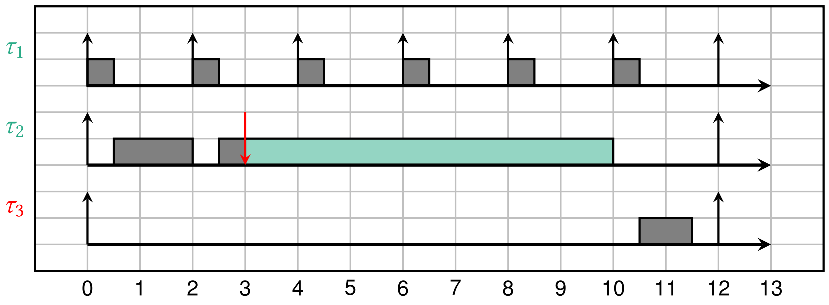

Fig. 11. Blocking by untrusted attack effective window.

Table 5. Task Parameters for the Example from Figure 11

Task	T	C	D	Ω	R	Type
τ_1	2.0	0.5	2.0	-	0.5	trusted
τ_2	12.0	2.0	3.0	7.0	3.0	trusted
τ_3	12.0	1.0	12.0	-	11.5	untrusted

The jobs of trusted higher priority tasks executed within the *AEW* can be excluded from the set of the interfering jobs. For trusted higher priority task τ_j, we derive a lower bound on the number of its jobs that must be entirely covered by an *AEW* of length Δ. Figure 12 illustrates our approach.

We look for a minimal number of task τ_j jobs that must fit entirely any time interval of length Δ. This happens when the time interval Δ starts immediately after the fastest completion of task τ_j instance (i.e., the task instance starts right after its release and completes as soon as possible). The last τ_j instance released within time interval Δ should start as late as possible to minimize the amount of its execution that overlaps with Δ. Since task τ_j is assumed to be schedulable, we assume that it starts $R_j - C_j$ after its release and completes R_j after its release. Please recall that the trusted tasks can follow a sporadic or periodic activation model. We also acknowledge that a tighter bound can be found for the periodic activation model.

$$overlap_j(\Delta) \;\geq\; \max\left\{0, \left\lfloor \frac{\Delta - R_j}{T_j} \right\rfloor \cdot C_j\right\} \tag{14}$$

We define $\beta_i(t_1, t_2)$ as an upper bound on the blocking time of the *AEWs* within time interval $[t_1, t_2]$ excluding the interference of the trusted higher priority tasks $thp(i)$ that must overlap within *AEWs* during this interval.

$$\beta_i(t_1, t_2) \quad \leq \sum_{[s,f]\in aew(t_1,t_2)} \left(f - s - \sum_{j\in thp(i)} overlap_j(f - s) \right) \tag{15}$$

For each interval $[s, f] \in \alpha(t_1, t_2)$, we compute its length, $f - s$, and subtract the amount of the minimal execution $overlap_j(f - s)$ of higher priority trusted jobs that can fall into any time interval of length $f - s$. We can now compute $\beta_i(\Delta)$ an upper bound on the blocking time of the *AEWs* over any time interval of length $\Delta > 0$ without the interference of the trusted jobs with priorities higher than i that must overlap within *AEWs* during this interval. To do so, as explained in Section 6.1, we must check all intervals starting within the hyperperiod of the victim tasks.

$$\beta_i(\Delta) \;\leq\; \max_{0\leq t<H} \beta_i(t, t + \Delta) \tag{16}$$

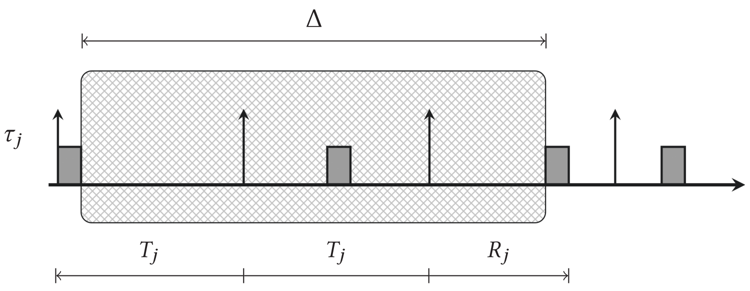

Fig. 12. Task τ_j minimal amount of execution within a generic interval Δ.

We can use the same approach as the one discussed in Section 6.1 and illustrated in Figure 7 for computing $\beta_i(\Delta)$ for larger values of Δ's length. A bound with polynomial-time complexity can be derived by combining Formulas (9) and (14) as follows:

$$\beta_i(\Delta) \;\leq\; \min\left\{\sum_{\tau_v \in V}\left\lceil\frac{\Delta}{T_v}\right\rceil \cdot \left(\Omega_v - \sum_{j \in thp(i)} overlap_j(\Omega_v)\right), \Delta\right\} \tag{17}$$

Finally, we can state that the upper bound R_i on the worst-case response time of untrusted task τ_i is given by the smallest positive integer satisfying the following relation:

$$R_i \;=\; C_i \;+\; \beta_i(R_i) \;+\; \sum_{j \in hp(i)}\left\lceil\frac{R_i}{T_j}\right\rceil \cdot C_j \tag{18}$$

7 IMPLEMENTATION

In this section, we describe how *SchedGuard++* was implemented in the Linux kernel before evaluating it in Section 8. To achieve the *SchedGuard++* functionality in the Linux kernel, we modified the kernel scheduler and made necessary changes to the cgroup interface as we chose to support containers. Containers offer low performance-overhead, support for Linux-based OS, ease of porting software, and isolation enforced by namespace [47]. They can be controlled through cgroups, making them compatible with the proposed security models. The implementation of *SchedGuard++* assumes that all trusted tasks run in containers and do not share containers with untrusted tasks. This implementation targets a partitioned multi-core system.

Linux cgroups are hierarchical groups that organize different resources for a collection of processes to perform resource allocation and monitoring. Examples include CPU, memory, device I/O, network, and the like. The CPU subsystem controls cgroup tasks access to the CPU. Each victim task should be configured to run in a cgroup pinned to only one CPU with its trusted tasks. If all trusted tasks and the victim cannot be fitted to a single CPU, some trusted tasks can be moved to a cgroup pinned to another CPU. More cgroups can be added as long as there are remaining CPUs not assigned to this trusted task set. Each cgroup has an added parameter to hold the task set identification number. To enable *SchedGuard++* blocking, one should first specify the protection window's length for the victim's cgroup. In our extension of the cgroup implementation, this can be achieved using the cgroup file system by setting the cpu.window_us attribute to a non-zero value. The cpu.window_us value is used to set the expiration time of the *SchedGuard++* hrtimer in the kernel.

To use the *SchedGuard++*, the victim task at the run time calls our newly added system call named cpu_block right before calling yield. The cpu_block shown in Function 2 ensures two functionalities: (1) it sets the kernel scheduler into protection mode (line-4); and (2) it programs the *SchedGuard++* hrtimer to fire in the future (line-7). It returns success if protection mode is set successfully and returns fail if the CPU is already in protection mode set by another task set. The victim should keep calling cpu_block until it returns success. In the protection mode, the kernel scheduler dequeues all rt_rqs on all available CPUs (Function 1 Line-5) that have real-time tasks ready to execute except the rt_rq of the victim's cgroup and the rt_rq of root task group's as it may have real-time kernel tasks. Suppose there are no real-time tasks ready for execution from the victim's cgroup or the root task group during protection mode(Function 1 Line-6). In that case, the kernel scheduler will skip scheduling all SCHED_NORMAL tasks (usually handled by the CFS scheduler) and select the system idle task for running until the protection window is finished. When the *SchedGuard++* hrtimer expires, it runs Function 3 to reset the kernel scheduler back to normal mode and enqueues all dequeued rt_rqs (Function 3 line-5). Since the blocking affects all CPUs, it is crucial to make sure the sum of all victims' protection window length does not exceed the available runtime of a single CPU. Otherwise, the entire system will be blocked indefinitely.

FUNCTION 1: Pick next rt task

```
1: function PICK_NEXT_TAK_RT(struct rq *rq)
2:     $p = \_pick\_next\_task\_rt(rq)$;
3:     if $protection = true \text{ and } p \neq kernel\_task \text{ and } p \rightarrow rt\_rq \neq victim\_rq$ then
4:         $list\_add(blocked\_rq\_list[cpu], p \rightarrow rt\_rq)$;
5:         $sched\_rt\_rq\_dequeue(p \rightarrow rt\_rq)$;
6:         return None;
7:     end if
         return p;
8: end function
```

FUNCTION 2: CPU Block

```
1: function CPU_BLOCK(struct task_struct *p)
2:     $spin\_lock(\&rt\_glock)$
3:     if $protection = false$ then
4:         $protection = true$;
5:         $monitor\_task = p$;
6:         $spin\_unlock(\&rt\_glock)$;
7:         $hrtimer\_start(victim \rightarrow timer)$;
8:         resched_cpus(); return success;
9:     end if
10:    $spin\_unlock(\&rt\_glock)$; return fail;
11: end function
```

8 EXPERIMENTS

This section describes the experimental setup where we have demonstrated our proposed approach's results on a realistic platform, a radio-controlled rover (RC) car. The implementation of *SchedGuard++* does not require the victim tasks to follow the LET model but can accept a more generic execution pattern. In the experiments, we do not enforce any execution pattern of the victim tasks to show *SchedGuard++* ability.

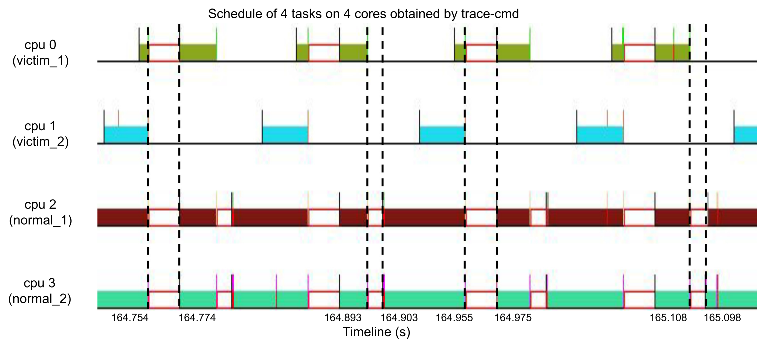

Fig. 13. Schedule of two victims and two background tasks on four cores.

FUNCTION 3: CPU Unblock Function

```
function CPU_UNBLOCK(struct task_struct *p)
    for_each_online_cpu(cpu)
        while !list_empty(&blocked_rq_list[cpu])
            rt_rq = list_entry(blocked_rq_list[cpu].next);
            sched_rt_rq_enqueue(rt_rq);
        end while
    end for
    protection = false;
    resched_cpus();
end function
```

8.1 Experimental Results on RC Car

The computing unit on the RC car employs a Raspberry PI 4B. It has quad-core cortex A-72 cores that run at 1.5 GHz each and comes with Linux kernel 4.19 pre-installed. To validate our approach's effectiveness, we first show the schedule of two victims with some synthetic untrusted task running on the Raspberry PI 4B with *SchedGuard++* enabled. This uses all four cores of the platform. Then to prove the defense capability, we show the results when *SchedGuard++* is used with a synthetic victim task and the ScheduLeak attack. Finally, we show how it can protect the RC car's autopilot application against a combined attack of ScheduLeak and output overwriting. Since ScheduLeak was developed for a single-core system, the last two experiments were done with only one core enabled.

8.1.1 Schedule of multi-victim on multi-core platform with SchedGuard++. This experiment shows the schedule of multiple victims on a multi-core platform when *SchedGuard* is enabled. All four cores on the Raspberry Pi 4B are used. The victim task has an execution time of around 30 ms and a period of 100 ms and is assigned with Linux real-time priority. The background task runs indefinitely with Linux normal priority. For core 0 and 1, each core is deployed with one victim task. For cores 2 and 3, each core is deployed with one background task. The victim task on core 0 is assigned a window time of 10 ms, while the victim task on core 1 is assigned a window time of 20 ms. trace-cmd is used to record the system schedule and displayed with kernelshark.

The schedule of the four tasks on four cores is shown in Figure 13. Whenever a victim task finishes execution, all cores will be blocked for the window time, including cores running other victim tasks.

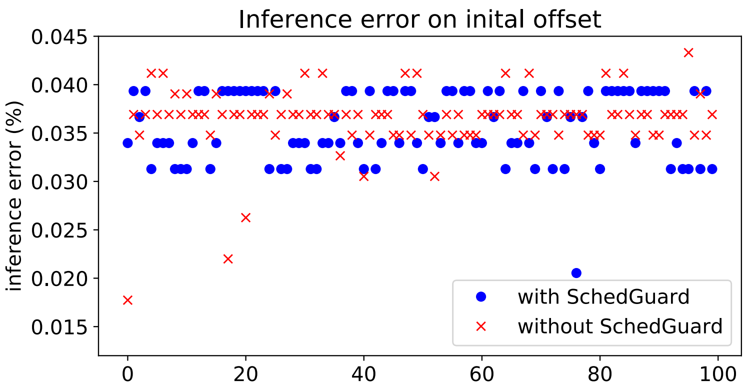

Fig. 14. ScheduLeak inference on task initial offset with and without *SchedGuard++*.

8.1.2 Defense against timing inference attack. There are works such as *ScheduLeak* [6] that exploit scheduling side-channel information to reconstruct a periodic victim task initial offset (i.e., the arrival time) and best-case execution time. An attacker can carry out an accurate timing-based attack without leaving any footprint with this information. *SchedGuard++* can affect the inference on execution time since it blocks the attacker task from obtaining any information during the protection window. Note that *ScheduLeak* is developed and verified on a single core system with one victim task. This experiment uses the same setup: enables only one core out of the four available cores with one victim task in the system.

The defense is demonstrated in the following example. The *ScheduLeak* algorithm is used to infer the victim task τ_v's initial offset a_v (i.e., the arrival time) and best-case execution time e_v. The observer task from *ScheduLeak* is configured as a SCHED_FIFO task with the lowest real-time priority in the system. The victim task is a periodic real-time task that runs with a 100-ms period. The measured average and best-case execution times for the victim task are 30 ms and 19 ms, respectively. We run only one periodic task (the victim task) in the system (excluding *ScheduLeak* itself and kernel threads) as this increases the chance the inference can succeed. The victim's period is passed to *ScheduLeak* as it is a prerequisite condition for it to succeed. To protect the victim task with *SchedGuard++*, the victim runs in a dedicated container alone, and a blocking window of 10 ms is assigned. This container is assigned a rt_runtime around 400 ms over a period of 1,000 ms to make sure its execution is not affected by the cgroup's RT throttling mechanism. The *ScheduLeak* algorithm runs in a different container following the vendor-oriented security assumption, and the rest of the system's remaining rt_runtime (550 ms) is assigned to it to increase its success rate. After the victim starts execution, *ScheduLeak* is invoked to run for 10 victim's period following the original article's recommendation.

The *ScheduLeak* algorithm is run 100 times for both *SchedGuard++* enabled and disabled cases. Inference results on the victim's initial offset and best-case execution time are shown in Figures 14 and 15. Figure 14 shows the percentage error in the victim task initial offset inference for both configurations. *ScheduLeak* can derive a very accurate a_v for the victim with only minor errors in both cases. This is because the *SchedGuard++* does not prevent the attacker from obtaining this information.

The inference results on the victim task's BCET are shown in Figure 15 for both configurations. The actual inference value instead of the percentage error is shown. The victim task has a true BCET of 19,000 us, while the majority of inference results fall between 20,000 us to 24,000 us when *SchedGuard++* is disabled. When *SchedGuard++* is enabled with a 10 ms ($\Omega = 10ms$) protection

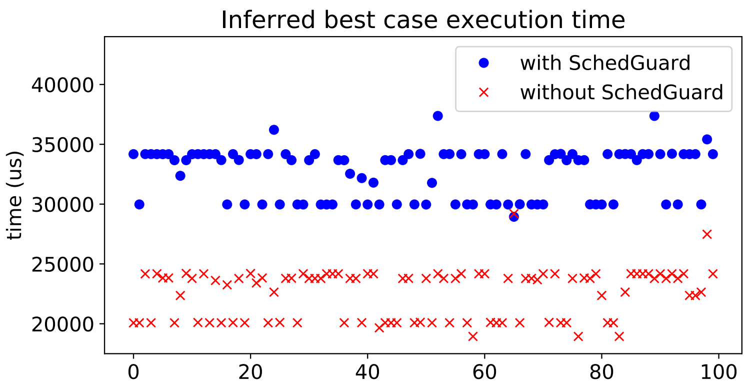

Fig. 15. ScheduLeak inference on victim task BCET with and without *SchedGuard++*.

window, most results range from 30,000 us to 34,000 us. Compared with the no protection case, the difference is the protection window size. This proves that *SchedGuard++* prevents the attacker from executing within 10 ms after the victim finishes and gives the attacker an impression that the victim has a longer execution time. With this false execution time, the attacker will launch a posterior attack at the wrong moment. If the protection window is longer than the attack effective window for that specific attack, the system is protected by *SchedGuard++*.

8.1.3 Defense against posterior control overwrite attack. In this experiment, we demonstrate the practibility of the proposed defense approach against an actual attack on an off-the-shelf RC car with Raspberry Pi 4 and Navio 2 sensor board.[1] The RoverBot software is utilized as the autopilot. RoverBot[2] is a modularized software stack that runs on Raspberry Pi 4 with a Navio 2 sensor board. RoverBot autopilot comprises functionally separated modules that may run in separate processes, such as Radio input, Localizer, Actuator, and so on. Communication among different modules implements a publish-subscribe mechanism using FastDDS[3] framework. To perform autonomous waypoint navigation, the Intel RealSense T265 tracking camera[4] is connected to the Raspberry Pi 4 computer to provide localization. The Intel RealSense SDK 2.0[5] is used to stream the vehicle's real-time positions RoverBot autopilot system, which drives the vehicle to waypoint locations.

This experiment adopts the same setup as the previous one: single core with one victim task. We launch the control output overwrite attack [6] that aims to override the PWM outputs governed by the Actuator task on the car system. To create a simpler environment for evaluating the attack and defense results, only the Actuator task is deployed as a SCHED_FIFO real-time task while others are run as non-real-time tasks. The Actuator task runs at a frequency of 100 Hz and has an average execution time of around 167 us. The container that runs the Actuator task is configured with rt_runtime as 400 ms, which ensures the task's execution is not throttled. To infer the Actuator task's initial offset, we launch a *ScheduLeak* attack as non real-time task in a separate container. The obtained initial offset is then used to launch the control output overwrite attack. In this attack, the attacker aims to override the steering to make the car turn right while the car is set to move straight. The experiment results are shown by the car's trajectories recorded under different test

[1] https://navio2.emlid.com/.
[2] https://github.com/bo-rc/Rover/blob/master/cpp/RoverBot.
[3] https://github.com/eProsima/Fast-DDS.
[4] https://www.intelrealsense.com/tracking-camera-t265/.
[5] https://github.com/IntelRealSense/librealsense.

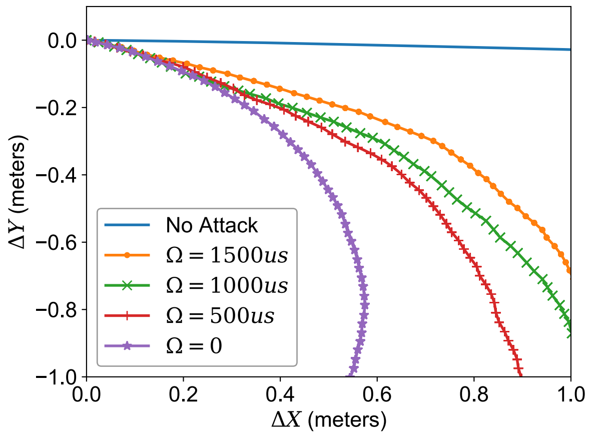

Fig. 16. The RC car's trajectories without an attack (the blue line) and with attacks under various ω_e settings are shown. In this experiment, the car's target is to move straight along the X-axis while the attacker tries to override the steering to make the car turn right.

settings as displayed in Figure 16. The blue line shows the car's trajectory without an attack as a reference. As the figure shows, the attack can make a sharp right-turn when no protection is involved ($\Omega = 0$). As the window length increases, the turn becomes flat and shaky. This is because the attacker is no longer occupying the *AEW*. The resulting PWM signal mixes the updates from the Actuator task and the attacker. As a result, the attacker cannot gain complete control of the car at will.

9 RELATED WORK

Logical Execution Time (LET) was first introduced in the *Giotto* programming language [17] which suggested using a time-triggered timing model with fixed reading/writing points at the beginning and the end of the task period for periodic hard real-time applications. One of the first studies about *LET* from OEM is Hennig et al. [16] from Daimler, who worked on a parallelizing legacy single-core application for a multi-core platform using *LET* programming paradigm. They identified independent tasks by analyzing dataflow architecture and ensuring their functional correctness through the implementation of the *LET* programming model on the multi-core platform with the help of Timing Definition Language (TDL) [43]. They also called for attention to integrate *LET* into AUTOSAR (AUTomotive Open System ARchitecture). Their effort is successful as now *LET* has been included in the *AUTOSAR* timing extension [54] and is being explored by both OEMs and Tier-1 suppliers. Resmerita et al. [44], working with Toyota, explored applying *LET* to legacy embedded control software. They parse the program's abstract syntax tree and control flow graph to optimize required I/O variables for efficient buffering. They implemented the approach in a tool suite and evaluated using industrial engine control software. Ziegenbein et al. [63] from Bosch presented a systematic co-engineering method between control and real-time analysis for automotive systems design. *LET* with worst-case response time (WCRT) brings determinism into the system, which eases the verification effort but at the cost of achieving a lower utilization of the targeted HW platform. They suggested using *LET* combined with typical worst-case response time (TWCRT), which is far smaller than WCRT for software design. They used simulation and formal verification to show the merits of the proposed approach. All the researches mentioned above focused on the application and implementation of *LET* into the automotive system with legacy software. However, they were not focusing on the security issues in the automotive system.

There have been several existing works that analyzed the attack surface of automotive systems [5, 27, 31, 41, 45, 49]. More and more attacks are focusing on the sensors or devices that interact with the physical world. A security team demonstrated that they could hack a Tesla through its WiFi system [37]. Several works have shown that the perception system of a car, such as cameras and LiDARs, can be attacked. Petit et al. [42] were able to perform blinding and confusing auto controls attacks on the camera system. They also successfully injected fake objects into the LiDAR by using relaying and spoofing attacks. Yan et al. [59] performed various attacks on the ultrasonic sensors, radars, and cameras and did demos using Tesla, Audi, Volkswagen, and Ford vehicles. Shin et al. [46] also used relaying attack to spoof LiDAR sensors to make it believe objects are closer than they are. They were also able to initiate a novel saturation attack to stop LiDAR from giving information in a particular direction. Cao et al. [4] built on Shin's work, controlled the spoofed points to trick the machine learning model, and modeled this as an optimization problem. Cao et al. [13] recently investigated the security issues in a Multi-Sensor Fusion system with a camera and LiDAR for autonomous driving. They developed an attack to show that an adversarial 3D-printed object can make both camera and LiDAR simultaneously ignore physical objects. They tested the proposed methods on real-world roads with real cars. Monowar and Mohan [15] implemented invariant checking within *ARM TrustZone* to protect the actuators under the same adversary model as employed in this article. However, based on the rover control case study, the measured runtime overhead of 43ms was several times higher than the worst-case execution times.

Side-channel attacks have been considered one of the major threats in the security community. A variety of them has been studied in the past in [19], [24], and [26]. Solutions such as cache flushing [18] and hardware/architectural [34, 50, 62, 64] modifications have been proposed as defense mechanisms without real-time constraints in mind.

The first work that demonstrated the leakage of information when scheduling tasks in a real-time environment is [48]. To defend a fixed-priority scheduler from leaking information, Volp et al. [56] suggest the use of a system idle thread. This approach does not consider what happens after the victim task has been completed. Similarly, works in [35] and [40] suggest defending against the schedule-based information leakage between the high- and low-security tasks by the introduction of flush tasks. This mechanism, however, introduces large overheads, resulting in poor response time of all the tasks in the system and effectively reducing system schedulability.

Another category of work to defend against the schedule-based attacks is to randomize the schedule [7, 60, 61]. However, these randomization-based approaches are not very effective and can easily be susceptible to attacks [36]. Our proposed work does not follow a schedule-randomization-based approach but instead tries to defend against the schedule-based attack by introducing the *AEW* and not allowing the attacker to run during this window.

Also, automotive networks are exposed to different kinds of cyber-attacks. For instance, if the attacker knows the victim message identifiers, a compromised device with access to a bus can perform spoofing or denial of service attacks. Lukasiewycz et al. [30] proposed to vary the message identifiers to mitigate these types of attacks. Since the message identifiers can determine the message priorities, as in the case of *Controller Area Network*, Lukasiewycz et al. provided the response time analysis to ensure that the message deadlines are not violated.

10 CONCLUSION

The *SchedGuard++* defense mechanism was introduced to defend against the posterior schedule-based attack using Linux containers on a multi-core processor. *SchedGuard++* prevents untrusted tasks from execution during the specified *AEW*. We proposed an exact method and a polynomial-time upper- and lower-bounds to compute the trusted execution time within a generic time interval, and response time analysis for trusted and untrusted tasks. We evaluated

SchedGuard++ with hardware experiments on an embedded platform with a real attack scenario. The results proved the effectiveness of the *SchedGuard++* defense mechanism.

REFERENCES

[1] [n.d.]. Hackers Remotely Kill a Jeep on the Highway–With Me in It. https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway. Accessed: 2022-02-01.

[2] S. K. Baruah. 1998. Feasibility analysis of recurring branching tasks. In *Proceedings of the 10th EUROMICRO Workshop on Real-Time Systems (Cat. No.98EX168)*. 138–145. https://doi.org/10.1109/EMWRTS.1998.685078

[3] Konstantinos Bletsas, Neil Audsley, Wen-Hung Huang, Jian-Jia Chen, and Geoffrey Nelissen. 2018. Errata for three papers (2004-05) on fixed-priority scheduling with self-suspensions. *Leibniz Transactions on Embedded Systems* 5, 1 (2018), 02–1–02:20. https://doi.org/10.4230/LITES-v005-i001-a002

[4] Yulong Cao, Chaowei Xiao, Benjamin Cyr, Yimeng Zhou, Won Park, Sara Rampazzi, Qi Alfred Chen, Kevin Fu, and Z. Morley Mao. 2019. Adversarial sensor attack on LiDAR-based perception in autonomous driving. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2267–2281.

[5] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, Tadayoshi Kohno. 2011. Comprehensive experimental analyses of automotive attack surfaces. In *Proceedings of the USENIX Security Symposium*, Vol. 4. San Francisco, 447–462.

[6] Chien-Ying Chen, Sibin Mohan, Rodolfo Pellizzoni, Rakesh B. Bobba, and Negar Kiyavash. 2019. A novel side-channel in real-time schedulers. In *Proceedings of the 25th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2019),* (Montreal, QC, Canada, April 16–18, 2019), Björn B. Brandenburg (Ed.). IEEE, 90–102. https://doi.org/10.1109/RTAS.2019.00016

[7] Chien-Ying Chen, Monowar Hasan, AmirEmad Ghassami, Sibin Mohan, and Negar Kiyavash. 2018. REORDER: Securing dynamic-priority real-time systems using schedule obfuscation. *arXiv preprint arXiv:1806.01393* (2018).

[8] Jiyang Chen, Zhiwei Feng, Jen-Yang Wen, Bo Liu, and Lui Sha. 2019. A container-based dos attack-resilient control framework for real-time UAV systems. In *Proceedings of the 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE'19)*. IEEE, 1222–1227.

[9] Jiyang Chen, Tomasz Kloda, Ayoosh Bansal, Rohan Tabish, Chien-Ying Chen, Bo Liu, Sibin Mohan, Marco Caccamo, and Lui Sha. 2021. SchedGuard: Protecting against schedule leaks using Linux containers. In *Proceedings of the 2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS'21)*. 14–26. https://doi.org/10.1109/RTAS52030.2021.00010

[10] Jian-Jia Chen, Geoffrey Nelissen, Wen-Hung Huang, Maolin Yang, Björn Brandenburg, Konstantinos Bletsas, Cong Liu, Pascal Richard, Frédéric Ridouard, Neil Audsley, Raj Rajkumar, Dionisio Niz, and Georg Brüggen. 2019. Many suspensions, many problems: A review of self-suspending tasks in real-time systems. *Real-Time Systems* 55, 1 (2019), 144–207.

[11] Rolf Ernst and Marco Di Natale. 2016. Mixed criticality systems -A history of misconceptions? *IEEE Design Test* 33, 5 (2016), 65–74. https://doi.org/10.1109/MDAT.2016.2594790

[12] Rolf Ernst, Stefan Kuntz, Sophie Quinton, and Martin Simons. 2018. The logical execution time paradigm: New perspectives for multicore systems (Dagstuhl seminar 18092). *Dagstuhl Reports* 8, 2 (2018), 122–149. https://doi.org/10.4230/DagRep.8.2.122

[13] Yulong Cao, Chaowei Xiao, Benjamin Cyr, Yimeng Zhou, Won Park, Sara Rampazzi, Qi Alfred Chen, Kevin Fu, and Z. Morley Mao. 2019. Adversarial sensor attack on lidar-based perception in autonomous driving. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2267–2281.

[14] Kai-Björn Gemlau, Leonie Köhler, Rolf Ernst, and Sophie Quinton. 2021. System-level logical execution time: Augmenting the logical execution time paradigm for distributed real-time automotive software. *ACM Trans. Cyber-Phys. Syst.* 5, 2, Article 14 (Jan. 2021), 27 pages. https://doi.org/10.1145/3381847

[15] Monowar Hasan and Sibin Mohan. 2019. Protecting actuators in safety-critical IoT systems from control spoofing attacks. In *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things (IoT S&P'19)* (London, United Kingdom). ACM, New York, 8–14. https://doi.org/10.1145/3338507.3358615

[16] Julien Hennig, Hermann von Hasseln, Hassan Mohammad, Stefan Resmerita, Stefan Lukesch, and Andreas Naderlinger. 2016. Poster abstract: Towards parallelizing legacy embedded control software using the LET programming paradigm. In *Proceedings of the 2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'16)*. 1–1. https://doi.org/10.1109/RTAS.2016.7461355

[17] Thomas A. Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. 2001. Giotto: A time-triggered language for embedded programming. In *Embedded Software*, Thomas A. Henzinger and Christoph M. Kirsch (Eds.). Springer Berlin, Berlin, 166–184.

[18] Wei-Ming Hu. 1992. Lattice scheduling and covert channels. In *Proceedings of the 1992 IEEE Computer Society Symposium on Research in Security and Privacy*. IEEE Computer Society, 52–52.

[19] Wei-Ming Hu. 1992. Reducing timing channels with fuzzy time. *Journal of Computer Security* 1, 3-4 (1992), 233–254.

[20] Wen-Hung Huang, Jian-Jia Chen, Husheng Zhou, and Cong Liu. 2015. PASS: Priority assignment of real-time tasks with dynamic suspending behavior under fixed-priority scheduling. In *Proceedings of the 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC'15)*. 1–6. https://doi.org/10.1145/2744769.2744891

[21] Hamidreza Jafarnejadsani, Hanmin Lee, Naira Hovakimyan, and Petros Voulgaris. 2017. Dual-rate L 1 adaptive controller for cyber-physical sampled-data systems. In *Proceedings of the 2017 IEEE 56th Annual Conference on Decision and Control (CDC'17)*. IEEE, 6259–6264.

[22] Jihan Kim, Gyunghoon Park, Hyungbo Shim, and Yongsoon Eun. 2016. Zero-stealthy attack for sampled-data control systems: The case of faster actuation than sensing. In *Proceedings of the 2016 IEEE 55th Conference on Decision and Control (CDC'16)*. IEEE, 5956–5961.

[23] Jihan Kim, Gyunghoon Park, Hyungbo Shim, and Yongsoon Eun. 2018. A zero-stealthy attack for sampled-data control systems via input redundancy. *arXiv preprint arXiv:1801.03609* (2018).

[24] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. 2012. {STEALTHMEM}: System-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*. 189–204.

[25] Tomasz Kloda, Bruno d'Ausbourg, and Luca Santinelli. 2016. EDF schedulability test for the E-TDL time-triggered framework. In *Proceedings of the 2016 11th IEEE Symposium on Industrial Embedded Systems (SIES'16)*. 1–10. https://doi.org/10.1109/SIES.2016.7509414

[26] Paul C. Kocher. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proceedings of the Annual International Cryptology Conference*. Springer, 104–113.

[27] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Hovav Shacham, and Stefan Savage. 2020. Experimental security analysis of a modern automobile. In *The Ethics of Information Technologies*. Routledge, 119–134.

[28] Joseph Y.-T. Leung and M. L. Merrill. 1980. A note on preemptive scheduling of periodic, real-time tasks. *Inform. Process. Lett.* 11, 3 (1980), 115–118. https://doi.org/10.1016/0020-0190(80)90123-4

[29] Chung Laung Liu and James W. Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)* 20, 1 (1973), 46–61.

[30] Martin Lukasiewycz, Philipp Mundhenk, and Sebastian Steinhorst. 2016. Security-aware obfuscated priority assignment for automotive can platforms. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 21, 2 (2016), 1–27.

[31] Charlie Miller and Chris Valasek. 2014. A survey of remote automotive attack surfaces. *Black Hat USA* 2014 (2014), 94.

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

[33] Yilin Mo and Bruno Sinopoli. 2009. Secure control against replay attacks. In *Proceedings of the 2009 47th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 911–918.

[34] Sibin Mohan, Stanley Bak, Emiliano Betti, Heechul Yun, Lui Sha, and Marco Caccamo. 2013. 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*. 65–74.

[35] Sibin Mohan, Man Ki Yoon, Rodolfo Pellizzoni, and Rakesh Bobba. 2014. Real-time systems security through scheduler constraints. In *Proceedings of the 2014 26th EUROMICRO Conference on Real-Time Systems*. IEEE, 129–140.

[36] Mitra Nasri, Thidapat Chantem, Gedare Bloom, and Ryan M. Gerdes. 2019. On the pitfalls and vulnerabilities of schedule randomization against schedule-based attacks. In *Proceedings of the 2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'19)*. IEEE, 103–116.

[37] Sen Nie, Ling Liu, and Yuefeng Du. 2017. Free-fall: Hacking tesla from wireless to can bus. *Briefing, Black Hat USA* 25 (2017), 1–16.

[38] OSEK 2005. *OSEK/VDX Operating System Specificatio*. OSEK. https://www.irisa.fr/alf/downloads/puaut/TPNXT/images/os223.pdf.

[39] Gyunghoon Park, Hyungbo Shim, Chanhwa Lee, Yongsoon Eun, and Karl H. Johansson. 2016. When adversary encounters uncertain cyber-physical systems: Robust zero-dynamics attack with disclosure resources. In *Proceedings of the 2016 IEEE 55th Conference on Decision and Control (CDC'16)*. IEEE, 5085–5090.

[40] Rodolfo Pellizzoni, Neda Paryab, Man-Ki Yoon, Stanley Bak, Sibin Mohan, and Rakesh B. Bobba. 2015. A generalized model for preventing information leakage in hard real-time systems. In *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 271–282.

[41] Jonathan Petit and Steven E. Shladover. 2014. Potential cyberattacks on automated vehicles. *IEEE Transactions on Intelligent Transportation Systems* 16, 2 (2014), 546–556.

[42] Jonathan Petit, Bas Stottelaar, Michael Feiri, and Frank Kargl. 2015. Remote attacks on automated vehicles sensors: Experiments on camera and LiDAR. *Black Hat Europe* 11, 2015 (2015), 995.

[43] Wolfgang Pree and Josef Templ. 2008. Modeling with the timing definition language (TDL). In *Model-Driven Development of Reliable Automotive Services*, Manfred Broy, Ingolf H. Krüger, and Michael Meisinger (Eds.). Springer Berlin, Berlin, 133–144.

[44] Stefan Resmerita, Andreas Naderlinger, Manuel Huber, Kenneth Butts, and Wolfgang Pree. 2015. Applying real-time programming to legacy embedded control software. In *Proceedings of the 2015 IEEE 18th International Symposium on Real-Time Distributed Computing*. 1–8. https://doi.org/10.1109/ISORC.2015.36

[45] Florian Sagstetter, Martin Lukasiewycz, Sebastian Steinhorst, Marko Wolf, Alexandre Bouard, William R. Harris, Somesh Jha, Thomas Peyrin, Axel Poschmann, and Samarjit Chakraborty. 2013. Security challenges in automotive hardware/software architecture design. In *Proceedings of the 2013 Design, Automation & Test in Europe Conference & Exhibition (DATE'13)*. IEEE, 458–463.

[46] Hocheol Shin, Dohyun Kim, Yujin Kwon, and Yongdae Kim. 2017. Illusion and dazzle: Adversarial optical channel exploits against LiDARs for automotive applications. In *Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 445–467.

[47] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. 2007. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. 275–287.

[48] Sang Hyuk Son, Craig Chaney, and Norris P. Thomlinson. 1998. Partial security policies to support timeliness in secure real-time databases. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy (Cat. No. 98CB36186)*. IEEE, 136–147.

[49] Ivan Studnia, Vincent Nicomette, Eric Alata, Yves Deswarte, Mohamed Kaâniche, and Youssef Laarouchi. 2013. Survey on security threats and protection mechanisms in embedded automotive networks. In *Proceedings of the 2013 43rd Annual IEEE/IFIP Conference on Dependable Systems and Networks Workshop (DSN-W'13)*. IEEE, 1–12.

[50] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. 2004. Secure program execution via dynamic information flow tracking. *ACM SIGPLAN Notices* 39, 11 (2004), 85–96.

[51] André Teixeira, Daniel Pérez, Henrik Sandberg, and Karl Henrik Johansson. 2012. Attack models and scenarios for networked control systems. In *Proceedings of the 1st International Conference on High Confidence Networked Systems*. 55–64.

[52] André Teixeira, Iman Shames, Henrik Sandberg, and Karl H. Johansson. 2012. Revealing stealthy attacks in control systems. In *Proceedings of the 2012 50th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 1806–1813.

[53] The AUTOSAR Consortium 2015. *Specification of Operating System*. The AUTOSAR Consortium. https://www.autosar.org/fileadmin/user_upload/standards/classic/4-2/AUTOSAR_SWS_OS.pdf.

[54] The AUTOSAR Consortium 2018. *AUTOSAR_RS_TimingExtensions, Specification of Timing Extensions*. The AUTOSAR Consortium. https://www.autosar.org/fileadmin/Releases_TEMP/Classic_Platform_4.4.0/MethodologyAndTemplates.zip.

[55] Steve Vestal. 2007. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS'07)*. 239–243. https://doi.org/10.1109/RTSS.2007.47

[56] Marcus Völp, Claude-Joachim Hamann, and Hermann Härtig. 2008. Avoiding timing channels in fixed-priority schedulers. In *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security*. 44–55.

[57] Franz Walkembach. 2016. *White paper: Model-Driven Development for Safety-Critical Software Components*. Technical Report MSU-CSE-06-2. Wind River. https://events.windriver.com/wrcd01/wrcm/2016/08/WP-model-driven-development-for-safety-critical-software-components.pdf.

[58] Jean-Paul Yaacoub and Ola Salman. 2020. Security analysis of drones systems: Attacks, limitations, and recommendations. *Internet of Things* (2020), 100218.

[59] Chen Yan, Wenyuan Xu, and Jianhao Liu. 2016. Can you trust autonomous vehicles: Contactless attacks against sensors of self-driving vehicle. *Def Con* 24, 8 (2016), 109.

[60] Man-Ki Yoon, Jung-Eun Kim, Richard Bradford, and Zhong Shao. 2019. TaskShuffler++: Real-time schedule randomization for reducing worst-case vulnerability to timing inference attacks. *arXiv preprint arXiv:1911.07726* (2019).

[61] Man-Ki Yoon, Sibin Mohan, Chien-Ying Chen, and Lui Sha. 2016. Taskshuffler: A schedule randomization protocol for obfuscation against timing inference attacks in real-time systems. In *Proceedings of the 2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'16)*. IEEE, 1–12.

[62] Man-Ki Yoon, Sibin Mohan, Jaesik Choi, Jung-Eun Kim, and Lui Sha. 2013. SecureCore: A multicore-based intrusion detection architecture for real-time embedded systems. In *Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS'13)*. IEEE, 21–32.

[63] Dirk Ziegenbein and Arne Hamann. 2015. Timing-aware control software design for automotive systems. In *Proceedings of the 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC'15)*. 1–6. https://doi.org/10.1145/2744769.2747947

[64] Christopher Zimmer, Balasubramanya Bhat, Frank Mueller, and Sibin Mohan. 2010. Time-based intrusion detection in cyber-physical systems. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*. 109–118.

Received 24 July 2021; revised 16 March 2022; accepted 14 September 2022