Resilient Scheduling of Real-Time Cyber-Physical Systems against Memory-Corruptions

Abdullah Al Arafat*, Kurt Wilson*, Sudharsan Vaidhun†, Bryan C. Ward‡, Zhishan Guo*
*North Carolina State University, †University of Central Florida, ‡Vanderbilt University
{aalaraf, kwilso24, zguo32}@ncsu.edu,vbsudharsan@gmail.com, bryan.ward@vanderbilt.edu

***Abstract*—Real-time cyber-physical systems (CPS) are increasingly deployed in command and control applications for safety- and mission-critical domains such as autonomous vehicles and critical infrastructure. To enable enhanced capabilities, CPS are becoming more complex and interconnected, yet this expanded functionality introduces new security vulnerabilities. Addressing these challenges, this paper presents a secure and resilient scheduling technique for hard real-time CPS applications that protects against common memory-corruption-based attacks. Our approach introduces a security-oriented dimension of criticality, enabling the system to selectively drop low-security-critical workloads in response to detected threats. This reduces the attack surface and allows for the timely rescheduling of both victim task re-executions and system recovery processes. We demonstrate that traditional mixed-criticality scheduling approaches are overly conservative and inadequate for accommodating dynamic recovery requirements under this security model. To address this, we propose a novel scheduling algorithm tailored for security-aware CPS, along with a schedulability test using a security-criticality demand-bound function. The proposed framework is implemented in FreeRTOS with micro-ROS and validated using a hardware-in-the-loop simulation of a flight control task. Extensive schedulability experiments reveal that our model outperforms existing approaches with required adaptation, improving acceptance ratios by over 30 percent in heavily utilized CPS environments. This work advances secure, real-time scheduling to enhance both the resilience and safety of critical cyber-physical applications.**

***Index Terms*—real-time cyber-physical system, mixed-criticality systems, resilient scheduling, secure recovery**

I. Introduction

Real-time and embedded systems are foundational to the control and coordination of complex cyber-physical systems (CPS) across various sectors, including autonomous vehicles, power grids, and the Industrial Internet of Things (IIoT). In applications such as these, onboard computational systems enable efficiency, autonomy, and responsiveness. However, the increasing complexity and interconnectedness of these embedded devices also broaden the attack surface, introducing new security vulnerabilities that could compromise system integrity and safety. Unlike general-purpose systems, embedded CPS often lack robust cybersecurity defenses, leaving them more susceptible to exploitation. For instance, the Mirai botnet [7], illustrated how basic vulnerabilities, such as default passwords in IoT devices, can be exploited at scale, turning thousands of embedded systems into attack vectors. Given the mission-critical nature of CPS, it is essential for these systems to not only detect and mitigate threats but also maintain stringent real-time performance and resilience in response to attacks.

Traditional cybersecurity techniques, such as address-space layout randomization (ASLR), are ubiquitous in general-purpose computing and enabled by default in operating systems like Windows, macOS, and Linux. However, real-time and embedded systems often avoid such randomization-based defenses due to the high unpredictability and performance overhead they can introduce, which conflicts with the strict timing constraints required by CPS [15], [20].

The most common class of vulnerabilities in CPS is *memory-corruption* ones, which attackers can exploit to overwrite or access unintended memory regions. For example, unchecked array access may read or write beyond their allocated memory, potentially altering executable code and disrupting the intended program flow. According to reports from Microsoft and Google, memory-corruption vulnerabilities account for roughly 70% of the vulnerabilities in their codebases [34], [41]. While methods to eliminate these vulnerabilities exist, such as Softbound [33], which incurs over 100% overhead in memory usage and CPU cycles, they are often too costly for real-time CPS applications. Alternatively, rewriting codebases in memory-safe languages like Rust, though effective, remains impractical at scale. As a result, most runtime defenses against memory-corruption-based attacks in CPS aim to detect exploitation attempts and prevent further compromise by terminating the affected process. For example, control-flow integrity (CFI) [4], [47] performs checks at control-flow transitions to ensure valid branch targets and crashes the process upon invalid control flow. Importantly, runtime defenses are *integral* to the protected task, that is, they are executed within the protected process, not in a separate process as in monitoring-based security approaches [22], [23], [24], [25]. Runtime defenses are, therefore, proactive, designed to prevent exploitation by stopping attacks before they escalate, whereas monitoring-based approaches are reactive, detecting anomalies and exploitation evidence post-occurrence. We note that runtime defenses are designed to prevent exploitation, while monitoring-based approaches detect anomalies and evidence of such exploitation.

However, in real-time systems that often control critical physical processes, crashing a compromised process can jeopardize the integrity and stability of the entire system. To safely integrate such defensive mechanisms, it is essential to provide *recovery capabilities* that can respond to detected malicious events without disrupting real-time constraints. Specifically, the system must be able to restart a process, return it to a

known safe state, and re-execute it promptly to maintain safe and continuous control. Restarting a real-time process (*i.e.*, job) introduces additional computational demands that can significantly impact its ability to complete within its deadline. Without careful management, these demands may further jeopardize the temporal integrity of other tasks, potentially cascading into broader system instability. Therefore, effective mitigation strategies are required to ensure that recovery operations uphold both the security and timing requirements essential to the reliable operation of cyber-physical systems.

These observations motivate the need for new task models and scheduling algorithms to enable resilience to memory-based attacks, *i.e.*, *the ability to maintain some safe level of operation while recovering from an attack.* While there may be naïve means of supporting such behavior in existing scheduling and analysis frameworks, maximizing the platform's utilization while enabling such resilience requires new models, algorithms, and analysis. Note that we do not specify the details of restoring the system to a known good state after an attack. Our focus is ensuring, given the timing properties of the system tasks and restoration task, that the recovery task can be completed without causing deadline misses in other critical tasks.

Mixed Criticality. This problem shares several important similarities with mixed-criticality (MC) scheduling, particularly in its capacity to operate in a degraded mode of execution. However, traditional MC scheduling models have primarily been developed to address a single type of aberrant behavior—temporal overruns—rather than security incidents. Notably, Burns has argued for the generalization of MC systems into multi-mode systems [14]. This work exemplifies this argument by demonstrating a multi-mode system in which mode switches are triggered by security events rather than timing overruns.

There are several important *similarities and differences* between the standard Vestal-model [44] for MC scheduling and the need of a resilient real-time recovery model. For example, when a security event is detected, it is useful to shed less-critical workloads to ensure the continued correct operation of high-critical workloads. Shedding workloads is especially useful for security as it can also reduce the attack surface of the system. There are, however, several important differences. First, when a defense prevents an attack, it crashes the process, requiring re-execution of the job and additional processing time. Another key difference with security criticality is that we assume that an attacker can target at most one task, not all tasks simultaneously. Mainly, memory corruption attacks target vulnerabilities in code, and because different tasks have different codes, they are not likely to be vulnerable to the same exploit payloads [40]. Tasks handling IO are ideal targets for memory attacks since mistakes in input processing can provide entry points to the system, making them attractive targets for attackers. Finding vulnerabilities is often difficult[1] and constructing malicious payloads is challenging, especially when modern defenses are employed. It is, therefore, unreasonable to assume that an attacker targets all processes in a system with unique exploits simultaneously.

Thus, it is too pessimistic to adapt existing results in MC analysis. In addition, in an MC environment, the system often returns to normal mode when a transient overload condition subsides. In contrast, returning to a normal mode of execution after detection of a security threat may require additional recovery processing for computations such as (i) adding the malicious input to a blocklist to ensure the re-executed task will not be attacked [30], (ii) forensic analysis, (iii) human-operator communication, and/or (iv) other actions to harden the security posture of the system, such as substituting binaries with stronger-defended ones, *etc.* Such additional computation time must also be modeled and analyzed. Notably, shedding less critical workload, with the proper analysis, frees computation time to enable such recovery processing without affecting the utilization of the normal mode. Moreover, existing non-MC and MC scheduling schemes cannot guarantee the immediate execution of the recovery process after the detection of an attack to prevent replay attacks on the victim task.

Contributions. Based on these observations, **(i)** we propose SR^3, a secure and resilient real-time recovery task model (Sec. II) that can recover from a memory-corruption-based attack at runtime while maintaining the correctness of high-security-critical tasks. Specifically, we propose a novel approach to model the recovery process by designing a sporadic server with guaranteed execution of the recovery process to sanitize the victim's task before re-executing; **(ii)** we develop a scheduling algorithm for the presented task model using the earliest deadline first (EDF) with virtual deadlines for security-critical tasks and corresponding demand-bound-function-based schedulability test (Sec. III); **(iii)** we present a system implementation of SR^3 in FreeRTOS running on micro-ROS and system implementation overhead analysis using a flight control task in hardware-in-the-loop simulation (Sec. V); and **(iv)** we conduct schedulability evaluations of SR^3 using synthetic workloads (Sec. IV). Our extensive evaluation demonstrates the effectiveness of SR^3 over adapted existing scheduling schemes and the feasibility of system implementation.

II. Model and Problem

A. Threat Model

We assume a threat model consistent with other prior works on runtime defenses [18], [47], [56]. Specifically, we assume a *write-what-where vulnerability*[2] that an attacker can leverage to corrupt code pointers [3] to hijack control flow to attacker-specified location(s). Significant research has shown that even such simple and common vulnerabilities can be exploited using return-oriented programming (ROP) [37] or other attack techniques (*e.g.*, [49]) to completely hijack control flow and

[1]Many companies have bug bounty programs that pay significant rewards for reporting vulnerabilities.

[2]An attacker's ability to write any value to any memory location, *e.g.*, leveraging a buffer overflow.

[3]A code pointer is any address stored in a data section that points to executable code. Return addresses on the stack are frequent attacker targets.

implement Turing-complete attacker-controlled logic. This is a very common and powerful threat model.

We assume the system is instrumented with an existing real-time runtime defense such as control-flow integrity (CFI) [18], [47], [56], [36], or data-flow integrity [12]. Notably, all of these defense techniques detect an attack on a task at or before the completion of its execution budget and prevent further exploitation of the task by crashing it.

Attacks on the scheduler or RTOS itself are outside the scope of our threat model. However, the scheduler and RTOS can be made trustworthy if using a verified RTOS (*e.g.*, seL4 [27]) or by using a trusted execution environment (*e.g.*, ARM TrustZone [48]).

B. System Model

Let $\Gamma' = \{\tau_1, \tau_2, ..., \tau_n\}$ be a set of independent n sporadic and constrained-deadline tasks scheduled on a uniprocessor. Each task τ_i can be represented by a tuple $(C_i, T_i, D_i, \varsigma_i)$, where C_i is the worst-case execution time (WCET)[4], T_i is the minimal inter-arrival separation, and D_i is the relative deadline (*i.e.*, $D_i \leq T_i$) of the task instances (*i.e.*, jobs). We assume each task can potentially release an infinite sequence of jobs. Let $\varsigma_i \in \{0, 1\}$ denote whether task τ_i is of low or high security criticality. We use $\Gamma_\varsigma = \{\tau_i | \varsigma_i = 0\}$ and $\Gamma_\varsigma = \{\tau_i | \varsigma_i = 1\}$ to denote the set of low-security-criticality (LO-security) tasks and high-security-criticality (HI-security) tasks, respectively. We model LO- and HI-security tasks based on the observation that some tasks are not essential to maintain the safe or secure operation of the system, especially when the system may be under attack. This is depicted in Table I. For example, in an automotive system, infotainment services are not mission-essential functions and should neither interfere with high security nor temporal criticality tasks. Some tasks are also high criticality with respect to both security and temporal criticality, as they support mission-critical functionality. However, there are some tasks that could be critical to the security of the system but be less critical to the temporal correctness of the system. For example, key management for encrypted communication may be critical to the security of the system, even if their timing is not mission-critical. Alternatively, some sensor readings may support optional or non-mission-critical functionalities, which could be disabled in the presence of a security threat. However, in order to maintain a consistent state, their processing is critical to 'timing'.

Note that LO-security tasks may themselves contain vulnerabilities. When the system is under attack, minimizing the attack surface is a valuable defense in and of itself. Given this motivation and model, we define the following terms:

Definition 1 (Victim Task and Targeted Task)**.** *Any task $\tau_v \in \Gamma'$ is a victim task when it is attacked during runtime. We further denote a* HI*-security victim task τ_v as a* Targeted Task*, $\tau_t \in \Gamma_\varsigma$, with an execution budget of C_t.*

Control-flow-hijacking attacks exploit one or more vulnerabilities within a *single* process to construct a malicious payload. Finding vulnerabilities and constructing malicious payloads is challenging, especially in the presence of modern defenses, and is application-specific. We, therefore, assume that an attack may target, at most, a single task at a particular time instant.

[4]Contrary to traditional MC, C_i does not change in different system modes

TABLE I: Tasks of differing temporal and security criticalities.

		Temporal Criticality	
		High	Low
Security Criticality	High	Safety-critical Control Processing	Encryption key management software
	Low	Processing non-mission-critical sensor inputs	Infotainment

Definition 2 (System Modes)**.** *The system will begin its execution under* **normal mode***, during which no attack on a security task is detected. Once a victim task is identified during runtime, the system will immediately switch into* **recovery mode***. Proper* actions *(see below) will be taken during recovery mode to prevent the system from further exploitation.*

When transitioning to the recovery mode, additional actions may be taken to facilitate recovery, for example, additional monitoring or validation of the system, forensic analysis, communication with human operators, *etc.* We model this additional workload as a *recovery workload*. To accommodate the recovery workload, we propose to use a 'sporadic server' [39] in the recovery mode. Modeling the resource reservation for recovery workload via a sporadic server instead of modeling by a regular task has multiple benefits. First, the behavior of the recovery workload often depends on the specific recovery scenario, which could potentially be only one instance task (*i.e.*, job) or a periodic task; thereby, modeling as a sporadic server allows for accommodating mixed types of recovery workloads. Second, it ensures that the recovery workload receives a guaranteed bandwidth and is not deferred as a best-effort task or simply scheduled as a background task, thereby allowing for quick recovery actions. This sporadic server for the recovery workload—denoted as *'Recovery Server'*—is in addition to the regular HI- and LO-security tasks defined below:

Definition 3 (Recovery Server)**.** *The recovery server $\tau_R = \{C_R, T_R\}$ is a sporadic server that is activated/released upon detection of an attack during runtime, where C_R is its execution budget and T_R is the period. The initial resource replenishment time of the recovery server is equal to the system mode switch instant.*

The design choices of server parameters are discussed later in this section. Note that we use a standard simple sporadic server [39] as the recovery server. Therefore, the resource consumption and replenishment rules for the simple sporadic server [39] directly apply to the recovery server. In addition, as the sporadic server behaves as a regular sporadic task, the schedulability analysis of the recovery server would be carried on as a sporadic task.

Design of Recovery Server *(to prevent Denial-of-Service (DoS) or Replay attacks)***.** Since it is important to put the malicious inputs into a 'blocklist' [30] upon detection of an

TABLE II: Workload considered in Example 1.

Task ID	C_i	T_i	ς_i
τ_1	1	3	0
τ_2	2	9	1
τ_3	5	25	1
τ_R	0.4	4	–

attack to prevent simply replaying the same attack on the tasks to perform DoS by an adversary, it is necessary to make sure that the recovery server gets the highest priority at the beginning of the recovery mode so that the malicious input can be block-listed by the recovery process. How can one ensure that the recovery process starts being executed by the recovery server at the beginning of recovery mode? This depends on the design of the scheduling algorithm for the system's overall workload. We will design the recovery server in section III after discussing the proposed scheduling algorithm, which is one of the key contributions of this paper.

The whole SR^3 system workload contains the HI- and LO-security tasks, as well as the recovery server, *i.e.*, $\Gamma = \Gamma' \cup \tau_R$.

Correctness Criteria. Given the SR^3 system, which contains a set of HI- and LO-security tasks and the recovery server, a *correct* scheduler must

1) guarantee that all HI-and LO-security tasks receive enough execution budget and meet their deadlines during normal mode;
2) ensure that all HI-security tasks (without failure, *i.e.*, except the Targeted task) continue to receive normal execution budget and meet their deadlines during recovery mode;
3) ensure that if the victim task is a Targeted task (the failing HI-security[5] task being attacked), then the victim task will receive another full re-execution budget (of its original WCET, C_i) beyond the mode switch point and meets its original deadline;
4) provide enough budget to the recovery server following the resource consumption and replenishment rules of sporadic server [39] during recovery mode;

Our objective is to identify a correct online scheduling mechanism and derive an offline schedulability test. Note that once there is a detected attack (and thus a mode switch), guarantees to service of LO-security tasks are no longer required, and this workload is dropped to minimize the attack surface. The targeted task must be re-executed before its original deadline in order to maintain continuous safe operations. We assume the malicious input can be placed in a blocklist using a proper recovery process and that a known-safe or sanitized input is used for the re-executed job. This assumption is consistent with prior work [30].

Unfortunately, the standard uniprocessor scheduling algorithms (*e.g.*, EDF) perform poorly on the task set. Besides, it is also non-trivial to prevent replay attacks in non-mixed-criticality systems as the execution of the recovery task before re-executing the victim task cannot be simply ensured with EDF without any additional constraint.

[5]When the victim task is a LO-security one, a mode switch is triggered immediately, while no re-execution budget will be allocated, as no guarantees are provided to LO-security tasks in recovery mode.

Example 1. *Consider a task set* $\Gamma = \{\tau_1, \tau_2, \tau_3, \tau_R\}$ *with parameters presented in Table II (*τ_R *is carefully chosen for dynamic priority algorithm). For ease of discussion, we consider an implicit deadline workload here. This regular sporadic task set is schedulable on a uniprocessor system under the EDF scheduler as the utilization (*$\sum_{\forall i} \frac{C_i}{T_i}$*) of the task set is* 0.855 *(including the utilization of recovery server,* τ_R*) which passes the optimal schedulability test (i.e.,* $\sum_{\forall i} \frac{C_i}{T_i} \leq 1$*) for implicit deadline workloads on a uniprocessor.*

Now let us map the SR^3 *system workload to a sporadic task model for EDF scheduling by doubling the execution of* HI*-security tasks,* $(C'_2 = 4, C'_3 = 10)$ *and keep the recovery server always active. After mapping the task set to a sporadic task model for EDF scheduling, the utilization of the mapped task set becomes* 1.277*. Therefore, the mapped task set with security awareness is not schedulable by EDF. We will later see that the task set is schedulable under our proposed scheduling algorithm.*

As only the targeted task is re-executed instead of all HI*-security tasks, intuitively, an optimistic case would be doubling only the execution of the targeted task instead of all* HI*-tasks while mapping to the sporadic task model from SR*^{3} *system workload. However, as any* HI*-security task can be the targeted task, one cannot simply double the execution budget of a task to cover the re-execution of any* HI*-security task. One might (pessimistically) think of designing a server (e.g., sporadic server) with an execution budget as the maximum WCET of any* HI*-task and server period as the minimum period of any* HI*-task. It is not entirely guaranteed that the server can re-execute a targeted task within the original deadline if the attack is detected right at the deadline moment of the job. This is why we have mapped the workloads for EDF as doubling the execution of all* HI*-tasks.*

III. Scheduling Algorithm

In this section, we first present our proposed scheduling algorithm (denoted as **sEDF-VD**) for the SR^3 system workload. As demonstrated by example 1, directly employing an EDF scheduler may lead to a too narrow scheduling window between the attack detection instant to its deadline instant for a HI-security task to re-execute upon an attack and thus lead to a deadline miss. Therefore, we proposed to adopt the concept of *virtual deadlines*, such that the HI-security tasks receive proper 'promotion' under the normal mode. We then present the design of the recovery server for the proposed sEDF-VD algorithm so that the recovery server receives a guaranteed execution budget at the beginning of the recovery mode. Finally, we present the schedulability test of sEDF-VD based on demand-bound functions (DBF) analysis.

A. Algorithm sEDF-VD

Let us start with a general overview of our proposed algorithm. At any instant in normal mode, we aim to promote

the execution of jobs of HI-security tasks over LO-security tasks, maintaining the deadline constraints of all tasks. To do so, we compute a virtual deadline $D_i^v = x \cdot D_i$ for each HI-security task such that the virtual deadline is less than or equal to the original deadline of the tasks (*i.e.*, no task exceeds the original deadline), where $x \in (0, 1]$ is a deadline shrinkage parameter. After computing a suitable virtual deadline for each HI-security task, HI-security tasks are scheduled using their virtual deadline and LO-security tasks with their original deadline following the EDF algorithm. The window between the virtual and actual deadlines for each HI-security job is 'reserved' for the HI-security task's potential re-execution upon attack/mode switch. Further, in recovery mode, all LO-security tasks are dropped immediately at system mode-switch instant. Then, in recovery mode, all HI-security tasks and the recovery server are scheduled following the EDF algorithm using the original deadlines.

B. Design of Recovery Server

We will now design the recovery server scheduled using sEDF-VD so that it can receive the highest priority at the beginning of the recovery mode. The following theorem states a sufficient condition for the server parameter:

Theorem 1. *The recovery server $\tau_R = \{C_R, T_R\}$ with system workload Γ' scheduled using* sEDF-VD *is guaranteed to receive the highest priority at the beginning of recovery mode* if *the period of the server is $T_R = \min\{D_i - D_i^v\}$, $\forall \tau_i \in \Gamma_\varsigma$.*

Proof. Since the workload would be scheduled by sEDF-VD, to receive the highest priority at the start of recovery mode, the server deadline has to be the earliest one among all active jobs at the mode-switch instance.

Following the system model, a job (let denoted as J) of a targeted task is the executing job during the mode-switch. Assuming all jobs met their respective (virtual) deadline in the normal mode (which is a necessary condition for schedulable workload), there are two cases for a mode-switch to recovery mode such as the mode-switch instant is (i) the absolute virtual deadline instant of the job J, and (ii) a time instant before the absolute virtual deadline of J.

Case (i): Since J is the executing job among all active jobs, J has the earliest virtual deadline among all active jobs. After the mode switch, all active jobs' deadlines will be updated to actual (absolute) deadlines from the virtual deadlines. Therefore, for all active jobs, the deadline would be at least $D_i - D_i^v$ units ahead of the mode-switch instant. Hence, the server with $\min\{D_i - D_i^v\}$ would be the earliest deadline (i.e., highest priority) job.

Case (ii): Since the mode-switch instant is before the virtual deadline of J and the deadline of all active jobs are updated to their respective absolute deadlines from virtual deadline instances (where all virtual deadlines of active jobs are after the mode-switch instant), the server with $\min\{D_i - D_i^v\}$ would be the earliest deadline (i.e., highest priority) job.

Note that if the deadline shrinkage factor $x = 1$, then $D_i = D_i^v$ implying $T_R = 0$, *i.e.*, server would be activated. However, $x = 1$ can only be possible if the system is not under attack, which is not the case for the schedulability test developed in the following section. Note that the schedulability test is considered the worst-case scenario, where the targeted task must be reexecute in the recovery mode. Therefore, $D_i - D_i^v$ for any $\tau_i \in \Gamma_\varsigma$ cannot be 0. ∎

C. Schedulability Test

We present a schedulability test of the sEDF-VD algorithm for the SR^3 system workload via the workloads' demand-bound functions (DBF) analysis across system modes. The DBFs of tasks (to be executed in the recovery mode) are inherently related between normal and recovery modes as the tasks (specifically HI-security tasks) would execute in both system modes. Therefore, we can shift demand from one mode to another by tuning the virtual deadline, $D_i^v = x \cdot D_i$ of the HI-security tasks.

Here, we derive DBF for HI-security tasks, LO-security tasks, and the recovery server. We then use those functions to determine the schedulability of sEDF-VD for the task set. Let us first define the DBF for a sporadic task [9] as follows:

Demand bound function (DBF), $\mathsf{dbf}(\tau_i, \ell)$**,** gives an upper bound of maximum possible execution of all jobs of a task $\tau_i = (T_i, C_i, D_i)$ that have both their arrival times and deadlines in the scheduling window, ℓ. The demand bound function is defined as follows [9],

$$\mathsf{dbf}(\tau_i, \ell) = \max\left\{\left\lfloor \frac{\ell - D_i}{T_i} \right\rfloor + 1, 0\right\} \cdot C_i \tag{1}$$

Baruah *et al.* [9] developed a necessary and sufficient condition for EDF scheduling of a non-mixed-critical sporadic task set using DBFs, which we stated as follows:

Theorem 2 (From Theorem 1 in [9])**.** *A task set τ can be successfully scheduled by the earliest deadline first (EDF) algorithm on a uniprocessor with a dedicated resource supply,* if and only if*,*

$$\sum_{\tau_i \in \tau} \mathsf{dbf}(\tau_i, \ell) \leq \ell, 0 \leq \ell \leq \ell_{max}, \tag{2}$$

where,

$$\ell_{max} = \min\left\{T + \max_{1 \leq i \leq n}\{D_i\}, \frac{U}{1-U} \cdot \max_{1 \leq i \leq n}\{T_i - D_i\}\right\},$$

where $T = \mathsf{lcm}_{1 \leq i \leq n}\{T_i\}$ and $U = \sum_{i=1}^{n} \frac{C_i}{T_i} < 1$.

We will formulate the DBFs for different tasks in the SR^3 workloads using the equation (1) and then the schedulability test leveraging Theorem III-C. Ekberg and Yi [19] proposed to analyze the DBFs for different critical tasks of the mixed-critical system separately. We have adapted a similar technique to derive the DBFs for different task categories of our task set separately. Let us first define a general DBF notation to calculate DBF for different tasks in different system modes as $\mathsf{dbf}^{\mathsf{task\_id}}_{\mathsf{sys\_mode}}(\mathsf{task}, \mathsf{length})$, *e.g.*, $\mathsf{dbf}^i_j(\tau_i, \ell)$ implies the DBF of i^{th} task $\tau_i \in \Gamma$ in j^{th} system mode for all time instants $\ell \geq 0$.

Now, we use the following breakdowns of DBF calculation for the task set:

- $\mathsf{dbf}_n^i(\tau_i, \ell)$—the demand of a task $\tau_i \in \Gamma'$ in *normal mode* (denoted as n), $\forall \ell \geq 0$.
- $\mathsf{dbf}_r^t(\tau_t, \ell)$—the demand of the targeted task τ_t in *recovery mode* (denoted as r), $\forall \ell \geq 0$.
- $\mathsf{dbf}_r^{i \neq t}(\tau_i, \ell)$—the demand of a task $\tau_i \in \Gamma_\varsigma \setminus \tau_t$ in *recovery mode*, $\forall \ell \geq 0$.
- $\mathsf{dbf}_r^R(\tau_R, \ell)$—the demand of the recovery server τ_R in *recovery mode*, $\forall \ell \geq 0$.

Let us consider each task of the task set we will use for the DBF-based schedulability test as a five-tuple $\{C_i, D_i^v, D_i, T_i, \varsigma_i\}$, and $\varsigma_i \in \{0, 1\}$. Here, D_i and D_i^v are the actual and virtual deadline of task τ_i, respectively, where $D_i = D_i^v \leq T_i, \forall \tau_i \in \Gamma_\setminus$, and $D_i^v \leq D_i \leq T_i, \forall \tau_i \in \Gamma_\varsigma$.

We will now derive DBF-based schedulability constraints of the SR^3 system workload considering the presence of a targeted task ($\tau_t \in \Gamma_\varsigma$), *i.e.*, the victim task is a HI-security task instead of any arbitrary task as a victim task. We then leverage the results to derive the schedulability constraints for the task set that includes LO-security victim task ($\tau_v \in \Gamma_\setminus$). Let us first derive the DBF bound for the normal system mode and then for the recovery system mode.

DBF in normal mode. When the system is in normal mode, each task $\tau_i \in \Gamma'$ works as a regular sporadic task on a uniprocessor with a deadline D_i^v and period T_i. Similar to a standard non-mixed-critical system, all tasks are trivially scheduled on a uniprocessor using EDF. So, we get tight-bound DBF for each task as follows [9],

$$\mathsf{dbf}_n^i(\tau_i, \ell) = \max\left\{\left\lfloor \frac{\ell - D_i^v}{T_i} \right\rfloor + 1, 0\right\} \cdot C_i;\ \forall \tau_i \in \Gamma', \forall \ell \geq 0 \tag{3}$$

Now, we will derive the **DBF of the tasks in recovery mode**. The demand for LO-security tasks in recovery mode immediately drops to zero. We will consider the tasks to be executed in the recovery mode into three mutually exclusive subsets of tasks, such as *regular* HI-*security tasks* (excluding the targeted task), *recovery workloads*, and the *targeted task*, to compute the DBFs separately. Notably, however, the resource reservation for recovery workloads would be maintained by the recovery server. Therefore, instead of computing DBFs for recovery workloads, we will compute the DBF for the recovery server. Before deriving the execution demand of tasks in recovery mode, we will derive the scenarios when a job of a regular HI-security task may have carry-over (defined in the following paragraph) during the mode-switch from normal mode to recovery mode. The targeted task and recovery server do not have any carry-over as the targeted task will re-execute again in recovery mode, and the recovery server receives the first resource allocation at the mode-switch instant.

Demand of Carry-Over Jobs. A carry-over job is a job of HI-security task that is released in normal mode and has a deadline in recovery mode. Fig. 1 illustrates a carry-over execution scenario. In recovery mode, we need to finish the remaining

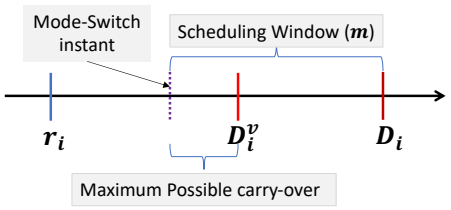


Fig. 1: HI-*security task's carry-over during mode-switch*

execution of any HI-security tasks ($\in \Gamma_\varsigma \setminus \tau_t$) before their deadlines. The amount of remaining execution depends on the task's carry-over demand. The maximum possible carry-over demand is the remaining execution window of each task from the mode-switch instant to the virtual deadline (D_i^v) of the task (Fig. 1). We need to consider the carry-over demand during the derivation of DBFs for 'regular HI-security tasks' in the recovery mode. The properties of carry-over jobs illustrated in Lemma III.3 in [19] also hold in our setup.

DBF of the regular HI-security tasks in recovery mode can be computed in similar way as [19]. Instead of direct calculation of carry-over demand, we first calculate the minimum execution requirement of carry-over jobs in normal mode before mode-switch as follows [19],

$$\mathsf{done}(\tau_i, \ell) = \begin{cases} \max\{C_i - m + D_i - D_i^v, 0\}; \\ \qquad\qquad \text{if } D_i - D_i^v \leq m \leq D_i \\ 0; \qquad\qquad \text{otherwise} \end{cases} \tag{4}$$

where $\tau_i \in \Gamma_\varsigma \setminus \tau_t$, τ_t is the targeted task and $m = \ell \mod T_i$.

Now, let us calculate the DBF of a task considering the carry-over job will fully execute in the remaining scheduling window $(D_i - D_i^v)$ for $\tau_i \in \Gamma_\varsigma \setminus \tau_t$ as follows [19],

$$\mathsf{full}(\tau_i, \ell) = \max\left\{\left\lfloor \frac{\ell - (D_i - D_i^v)}{T_i} \right\rfloor + 1, 0\right\} \cdot C_i \tag{5}$$

Therefore, the *maximum bound* of DBF of any 'regular HI-security task' $\tau_i \in \Gamma_\varsigma \setminus \tau_t$ is,

$$\mathsf{dbf}_r^{i \neq t}(\tau_i, \ell) = \mathsf{full}(\tau_i, \ell) - \mathsf{done}(\tau_i, \ell) \tag{6}$$

DBF of targeted task in recovery mode include the malicious job of the targeted task, which needs to re-execute in the scheduling window of $(D_t - D_t^v)$. Therefore, the DBF of the targeted task is as follows,

$$\mathsf{dbf}_r^t(\tau_t, \ell) = \max\left\{\left\lfloor \frac{\ell - (D_t - D_t^v)}{T_t} \right\rfloor + 1, 0\right\} \cdot C_t \tag{7}$$

DBF of the recovery server. As mentioned in Sec. II, the recovery server is designed using a sporadic server, which behaves exactly the same as a sporadic task from the schedulability point of view. Besides, the recovery server receives the first resource allocation at the mode-switch instant. So, the DBF of recovery server $\tau_R = (C_R, T_R)$ in recovery mode can be formulated as follows,

$$\mathsf{dbf}_r^R(\tau_R, \ell) = \max\left\{\left\lfloor \frac{\ell}{T_R} \right\rfloor, 0\right\} \cdot C_R \tag{8}$$

Using the equations of DBFs of different tasks and leveraging Theorem III-C, we get the following schedulability test:

Theorem 3. *A* SR^3 *system workload* Γ *can be successfully scheduled by EDF with virtual deadlines on a uniprocessor* if, *for any targeted task* $\tau_t \in \Gamma_\varsigma$*, both of the following conditions hold for* $\forall \ell \geq 0$*,*

$$A(\ell): \sum_{\tau_i \in \Gamma'} \mathsf{dbf}_n^i(\tau_i, \ell) \leq \ell$$

$$B(\ell): \sum_{\tau_i \in \Gamma_\varsigma \setminus \tau_t} \Big(\mathsf{dbf}_r^{i \neq t}(\tau_i, \ell)\Big) + \mathsf{dbf}_r^R(\tau_R, \ell) + \mathsf{dbf}_r^t(\tau_t, \ell) \leq \ell$$

Proof. The proof follows the Theorem III-C that the total demand in normal mode (condition $A(\ell)$) is less or equal to the available scheduling window at any instant, same as for recovery mode (condition $B(\ell)$). Condition $B(\ell)$ in Theorem 3 is need to verified for any $\tau_t \in \Gamma_\varsigma$ as any task in Γ_ς could be a targeted task. ■

Although in the DBF test in Theorem III-C it is sufficient to test up to ℓ_{max} instead of all $\ell \geq 0$ (ref. equation (2)), Theorem 3 is developed for all $\ell \geq 0$. Due to space limitations, we did not present such a result in the main paper.

Note. Any LO-task could also be attacked, and therefore, the system must need to switch recovery mode to recover from the threat once the attack is detected. However, as the LO-task would not re-execute in the recovery mode, only the HI-tasks can create the worst-case scenario for the schedulability test as considered in Theorem 3.

Determination of recovery server parameters, T_R, C_R, and deadline shrinkage parameter x. In Algorithm 1, we present a binary-search algorithm for the deadline shrinkage parameter $x \in (0, 1]$. For each x, we need to find the appropriate server parameters T_R and C_R using the recovery server's utilization u_R (ref. Line 6, 7). Then, adding the recovery server task to the system workload (ref. Line 7), we check the DBF-based schedulability constraints (ref. Line 8) presented in Theorem 3. Note that in the calculation of C_A and C_B in Line 8, we need to consider $\forall \ell \in [0, l_{max}]$ instead of $\forall \ell \geq 0$ and can be efficiently computed using quick processor demand analysis (QPA) [55]. With an initial value of x, Line 8 returns the schedulability constrains C_A and C_B. Later, Lines 9 to 11 check whether the current value of x is feasible for the schedulability of the workload or not. Depending on C_A and C_B, the algorithm either returns the current value of x as a common shrinkage factor for all HI-security tasks or searches for a feasible x in the half of the search space of previous iteration. In short, Algorithm 1 returns either a common shrinking factor x for each HI-security task if the task set is schedulable and the recovery server's parameters or `FAILURE` if the task set is not schedulable within the precision tolerance limit ϵ.

Example 2. *We again consider the task set in Table II. To test the schedulability of the workload scheduled by the sEDF-VD, we need to find whether there exists a feasible x that satisfies both the schedulability constraints of Theorem 3. Following the Algorithm 1, the task set is schedulable. Since the Algorithm 1 returns only a single feasible value of x, we further find out the range of x as $0.36 < x < 0.76$ (via an exhaustive search for $x \in (0, 1]$) for $D_i^v = x \cdot D_i$ for all* HI*-security tasks which satisfy the schedulability constraints in Theorem 3 with precision accuracy of $\epsilon = 0.01$.*

Algorithm 1: Procedure for finding server parameters $\{C_R, T_R\}$ and deadline shrinkage parameter x

Input: A SR^3 system workload $\Gamma' = \{\Gamma_\varsigma, \Gamma_\backslash\}$, u_R (utilization of τ_R), and precision accuracy ϵ

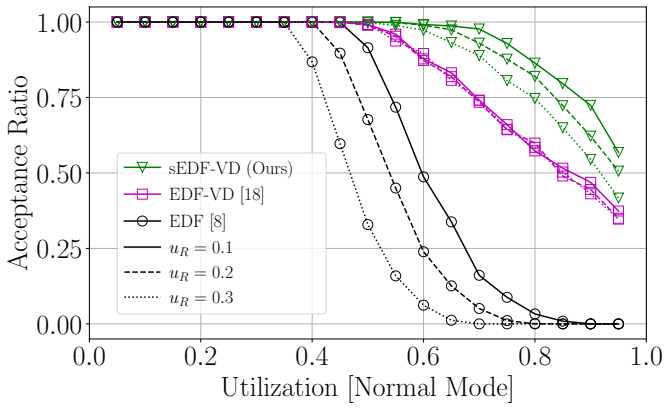
1 $\delta \leftarrow 0.5; x \leftarrow \delta$; // `step size for binary search and initial shrinking factor`
2 **while** $\delta \geq \epsilon$ **do**
3 $\quad \delta \leftarrow \delta/2$; // `updating step size`
4 $\quad$ **for** *each* $\tau_i \in \Gamma_\varsigma$ **do** $D_i^v \leftarrow x \cdot D_i$;
5 $\quad T_R = \min\{D_i - D_i^v | \tau_i \in \Gamma_\varsigma\}$; // `set the server period for current` x
6 $\quad C_R = u_R \times T_R$; // `execution budget of the server`
7 $\quad \Gamma = \Gamma' \cup \{C_R, T_R\}$
8 $\quad C_A, C_B =$ *Check* (Condition $A(\ell)$ & $B(\ell)$ in Thm 3) ; // where, $\ell \in [0, \ell_{max}]$
9 $\quad$ **if** $C_A \wedge C_B$ **then** return $x, \{C_R, T_R\}$;
10 $\quad$ **else if** $C_A \wedge \neg C_B$ **then** $x \leftarrow x - \delta$;
11 $\quad$ **else if** $\neg C_A \wedge C_B$ **then** $x \leftarrow x + \delta$;
12 $\quad$ **else** return `FAILURE`; // `no` x `can be found`
13 **end**
14 return -1 ; // `still possible to find a` x `with smaller` ϵ

IV. Schedulability Evaluation

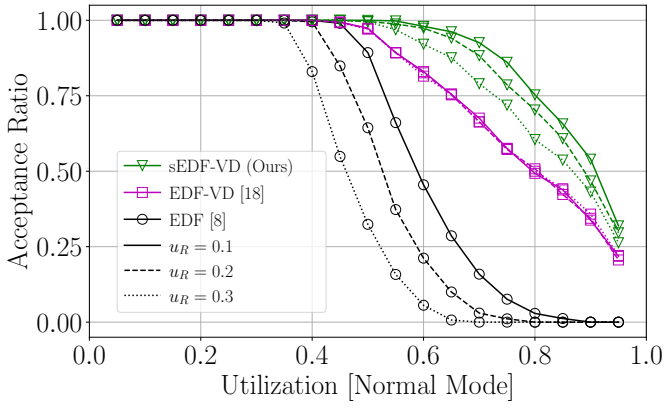
We evaluated the schedulability of sEDF-VD through synthetic task sets.

Baselines. It is important to note that existing algorithms do not guarantee the execution of recovery tasks immediately after attack detection. However, for schedulability ratio comparison with sEDF-VD, we assumed existing algorithms could be adapted to run the recovery process properly without additional overheads. To compare sEDF-VD with the standard deadline-based algorithms, we mapped the SR^3 workloads to the non-mixed-critical sporadic workload model to schedule using EDF [29] and Vestal's mixed-criticality workload model [44] to schedule using EDF-VD [19]. We discussed the mapping to each model and the corresponding scheduling test used to compare as follows:

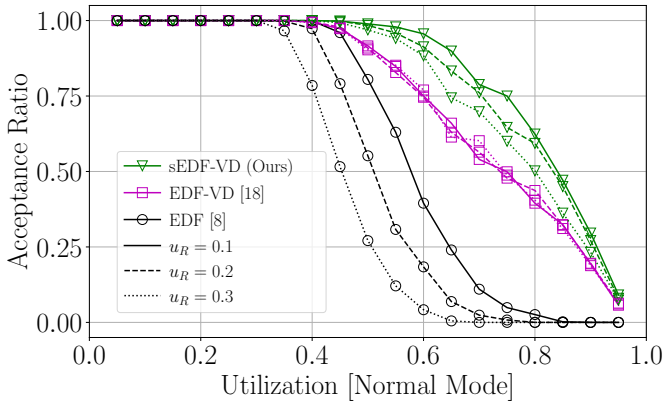
- **EDF** *(Non-Mixed Critical Scheduling Baseline).* We map an SR^3 system workload to the standard sporadic task model via doubling the execution of each HI-security task. Additionally, we also include the recovery server. Note that we justify such mapping in the example 1. With our proposed task model mapped to the sporadic task model, we use the **DBF-based schedulability test [9]** to determine the EDF schedulability for the transferred set as the DBF test is optimal for constrained deadline tasks scheduled by EDF on a uniprocessor.



(a) $R_D = 0.9$



(b) $R_D = 0.8$



(c) $R_D = 0.7$

Fig. 2: *Acceptance ratio of the workloads under varying utilization of the recovery server, subfigures correspond to the different deadline-to-period ratios of the constrained deadline task sets.*

- **EDF-VD** *(Mixed-Critical Scheduling Baseline).* We map our SR^3 system to the MC task model by doubling the utilization in the recovery mode for all HI-security tasks. We also add the recovery task as a HI-criticality task to the system where the normal execution budget of the recovery task is set as 0. We use EDF with a virtual deadline (EDF-VD) [8] as the scheduling algorithm. EDF-VD is a widely accepted scheduler for Vestal's MC task model. We apply **DBF-based schedulability test [19]** to determine the schedulability of the transferred task set.

Workload generation. The SR^3 system workload generation is controlled by the following parameters:

- $n = \{10\}$: Number of tasks in a task set
- $u_R = \{0.1, 0.2, 0.3\}$: Utilization of the recovery server
- $U = U_{\text{LO}} + U_{\text{HI}} = \{x/20 \mid 1 \leq x < 20\}$: Total utilization of the task set in normal mode
- $P = \{0.5\}$: Probability of a task being HI-security
- $R_D = \{0.9, 0.8, 0.7\}$: Deadline-to-period ratio

The task set generation begins with a target value for normal mode utilization given by U. Using the UUniFast algorithm [13], we derive the set of task utilizations in normal mode. The utilization of the recovery server is given by the u_R parameter. Each task has a randomly selected period from 2 to 625, and the execution time is found by multiplying the task utilization by its period. Deadline-to-period ratio R_D is used to generate constrained deadline workload. For each setting, we generate 1000 task sets and present the results below.

Figure 2 reports the variation in acceptance ratios for varying system utilizations under different recovery server utilizations, where each subfigure demonstrates a different deadline-to-period ratio of the constrained workload. All the results are compared with two baseline algorithms.

Observations. When applying EDF [9], Figure 2 shows that the acceptance ratio begins to drop as U increases beyond 0.5, while the utilization of recovery server has noticeable effects on the acceptance ratio. While sEDF-VD outperforms both of the baseline algorithms, there is a noticeable performance gap between the baselines too. EDF-VD [8] performs significantly better than the EDF [9], which justifies the importance of our modeling the SR^3 system as a mixed-criticality system instead of a standard of non-mixed-critical system. However, there is around 20% performance gap between sEDF-VD and EDF when system utilization is over 70%, mainly for pessimistic modeling of SR^3 workload to Vestal's MC model, implying the empirical dominance of sEDF-VD over EDF-VD.

We observe that the acceptance ratio decreases as the recovery server utilization increases. We see that this degradation is independent of the algorithms considered and is a direct result of the added utilization in the recovery mode. Besides, we demonstrated the effect of constrained deadlines of the workloads on the schedulability through Figure 2a to 2c. As expected, the more the deadlines become smaller than the periods of the tasks, the less the number of the tasks becomes schedulable. Such patterns are observable in Figure 2a to 2c.

V. Case Study

This section presents a case study implementation of SR^3 system using FreeRTOS with micro-ROS and validated using a hardware-in-the-loop simulation of a flight control task.

A. Implementation

We implement SR^3 on a testbed running micro-ROS [2] and FreeRTOS [1]. The hardware platform is an STM32 F446RE Nucleo-64 development board, which is based around an ARM

Cortex-M4 core. ROS 2 [3] is a popular framework for robotics applications, and due to the resource-constrained nature of microcontrollers, micro-ROS has been developed to make core ROS 2 functionality available to embedded platforms. We use micro-ROS on our platform using FreeRTOS [1] as the operating system. We test the system's performance with hardware-in-the-loop simulation, where the microcontroller flies a 3DR Iris quadcopter simulated in Gazebo, and use ROS 2 Foxy as the communication layer.

The FreeRTOS kernel (version 10.2.1) is modified to implement the sEDF-VD scheduling policy. To implement sEDF-VD in FreeRTOS, we modify the existing round-robin ready queue into a queue prioritized by absolute deadlines—denoted as `readyQueue`. The deadline shrinkage factor x is applied to HI-security tasks at the moment they are added to the `readyQueue`.

Workloads. We modeled the flight control and communication tasks as HI-security tasks, while the logging and dummy tasks as LO-security tasks. The `micro_ros_subscriber` task subscribes to two topics: IMU messages from the simulator, and control inputs from the pilot. For the purpose of the case study, the flight control inputs are kept constant, instructing the quad to fly level while slowly ascending. An additional value is sent along with the control inputs that can be manipulated to cause a stack overflow. This value does not affect flight performance, but its value controls the position of an array write operation. The vulnerable task does not check the bounds of the array write, so the extra control input can be used to cause memory corruption. The dummy tasks run `NOP` instructions to emulate additional workload. We present the summary of the workloads used in the case study in Table III. The flight controller is a PID controller that maintains level flight by adjusting motor output to minimize angular velocity on each axis. The workload is schedulable following sEDF-VD with a shrinkage factor $x = 0.5$.

Attack Formation. To emulate the attack, the extra control input is used to trigger a stack overflow. The extra control input does not affect the flight behavior of the quad and is intended to simulate an input vulnerable to attack. During the experiment, a victim task writes to an array using the control input as the index. Setting the control input to a high value causes the task to write invalid data to the top of its stack, activating FreeRTOS's stack overflow detection. Since the position of the sentinel value is known for each task, we can reliably trigger the FreeRTOS stack overflow detector.

Runtime Defense. As a defense against a memory-corruption attack, we use FreeRTOS's built-in stack overflow detection to serve as a basic runtime defense. The FreeRTOS stack overflow detector has two modes available. The first checks whether the pointer to the top of the stack has increased beyond the end of the stack, and the second checks whether a sentinel value at the end of the stack has been overwritten. Both checks happen at the scheduler's interrupt tick, which is runs at 1000hz by default, and whenever a task is switched out. For evaluation, we use the second mode. When a stack overflow is detected, FreeRTOS calls a user-defined

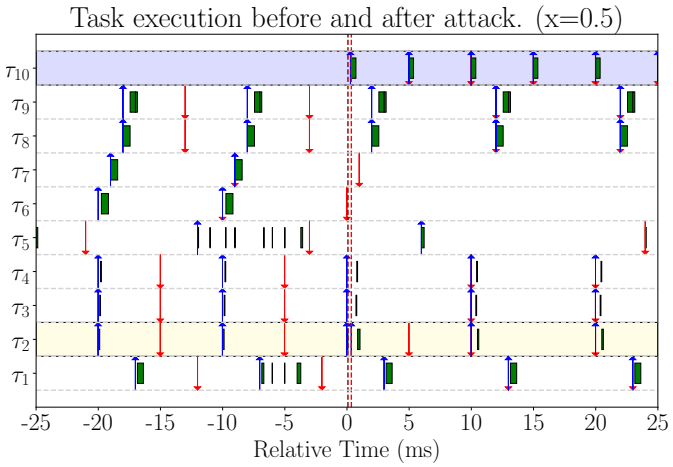


Fig. 3: *Events timeline for the task set in Table III scheduled using sEDF-VD algorithm on FreeRTOS running micro-ROS. Each box represents the execution of a job. The purple vertical line indicates the arrival of the attacking message to the highlighted victim task, τ_2, and the red vertical line indicates when the system detected the error. Task releases are denoted with a blue upwards arrow, and task deadlines are denoted with a red downwards arrow. Following the proposed approach in Algorithm 1, the highlighted recovery task, τ_{10}, is always executed immediately after the mode switch and victim task τ_2 re-executed by its original deadline.*

`vApplicationStackOverflowHook` function.

We set the user-definable overflow handler to kill the attacked task, change the mode into recovery mode, drop all LO-security tasks, and set the deadline scaling factor to 1. It also releases the recovery task. If the victim task was a HI-security task, it restarts the task. During the mode switch, the new scaling factor is applied by updating the relative deadline values with the new scaling factor, finding the new absolute deadlines for tasks in the ready queue, and re-sorting it.

To measure task execution, we use the `traceTASK_SWITCHED_IN` and `SWITCHED_OUT` trace hooks provided by FreeRTOS. FreeRTOS calls the corresponding hook whenever the scheduler switches a task in or out, passing the task as an argument to the callback, allowing us to determine exactly when tasks are executed. We record these events, along with task deadlines and releases, into a circular buffer, which is sent over ROS 2 at the end of the experiment.

B. Evaluation

We evaluate the usefulness of the proposed solution by investigating the following research questions (RQs): **RQ1:** Can this be run on a real system? **RQ2:** Can the schedulability test be replicated on a real system? **RQ3:** What are the possible implementation overheads?

Evaluating RQ1: Real World System Implementation. We report the workloads for the case study of a quadcopter with hardware-in-the-loop simulation in Table III. In the experiment depicted by Fig. 3, the targeted task is τ_2, where an attack

TABLE III: Description of workloads used in the case study

ID	Name	Description	Parameters
τ_1	m_ros_subscriber	Receive & decode ROS 2 messages over serial	$T = 10ms, C = 507\mu s, \varsigma = 1$
τ_2	calculate_angles	Process raw gyroscope and accelerometer observations, and correct gyroscope drift	$T = 10ms, C = 180\mu s, \varsigma = 1$
τ_3	calculate_errors	Calculate desired state and error from control inputs	$T = 10ms, C = 64\mu s, \varsigma = 1$
τ_4	pid_controller	Determine new control actions from the calculated errors	$T = 10ms, C = 64\mu s, \varsigma = 1$
τ_5	esc_publish	Publish control actions to simulator via ROS 2	$T = 18ms, C = 285\mu s, \varsigma = 1$
$\tau_{6,7}$	dummy_task_hi	Dummy load	$T = 10ms, C = 500\mu s, \varsigma = 1$
$\tau_{8,9}$	dummy_task_lo	Dummy load	$T = 10ms, C = 500\mu s, \varsigma = 0$
τ_{10}	recovery task	Stops LO-tasks and restarts the attacked task. After recovery, it serves as a dummy task to simulate the recovery server budget.	$T = 1000ms, C = 860\mu s$

has occurred at $144465\ ms$ (purple line) into the experiment. Note that we used relative time in Fig. 3 mapping $144465\ ms$ to $0\ ms$ for ease of the presentation. Once τ_2 is switched out by the scheduler, the stack overflow check occurs and fails, triggering a mode switch. The mode switch also releases the recovery task of τ_{10}. $\tau_{8,9}$, both LO-security tasks, are stopped and removed. The scaling factor is reset to 1.0, and the scheduler re-sorts the ready queue to account for the changed deadlines. τ_2 is re-created and released immediately, maintaining the target job's original absolute deadline as the new deadline. The effect of the scaling factor on deadlines of the HI-security tasks is visible; the new deadlines are expanded to ensure adequate time for recovery. This demonstrates the feasibility of SR^3 on real-world systems.

Evaluating RQ2: Schedulability Evaluation via randomly generated task sets. We also generated tasksets to run on our hardware implementation. Similar to the case study setting, we generated tasksets with 12 tasks ($n = 12$), where half are LO-security tasks and half are HI-security ($P = 0.5$). The recovery task utilization u_R is randomly selected from $\{0.1, 0.2, 0.3\}$. The schedulability test results for these are shown in Fig. 4. Similar to the evaluation of synthetic workload in Fig. 2, sEDF-VD performs better than the baseline algorithms. We randomly selected 100 tasksets that passed our proposed schedulability test, and 100 that did not, and ran them on the hardware implementation while recording task response times. We show the response times of these tasks in Fig. 5.

Although both HI- and LO-security tasks are selected uniformly and with equal numbers in each task, we observed the consistent response time differences between the HI- and LO-security tasks. As shown in Fig. 5, regardless of the utilization of the task set, the response time of HI-security tasks is higher than the LO-security tasks, which is expected as the targeted task needs to re-execute, and all HI-security tasks remain active in recovery mode with recovery tasks. Besides, we also observed that the tasks that passed the schedulability test never missed the deadlines; however, we also observed that most of the task sets that failed in the schedulability test also did not miss any deadlines. We believe there are two potential reasons for such a relatively low deadline missed ratio of the tasks that failed to pass the schedulability test failed tasks: (1) it could be the artifact of our schedulability as the test is sufficient only, or (2) we ran the test on the board to a limited time horizon, such that the worst-case scenario in our system implementation never occurred.

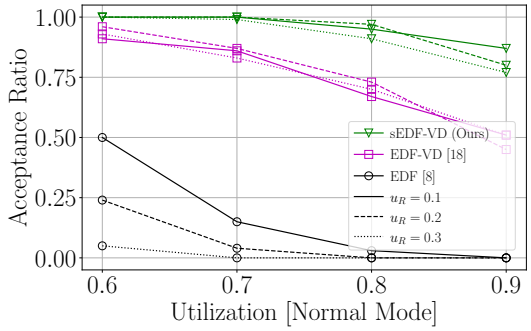


Fig. 4: *Schedulability of tasksets tested on our system implementation.*

Evaluating RQ3: Implementation Overhead Analysis. There are four sources of overhead in our implementation: detecting that an attack has occurred, killing LO-security tasks, adjusting deadlines of tasks in the ready queue and re-sorting it, and creating the recovery task. FreeRTOS performs its stack overflow check in the timer tick interrupt, so the maximum time taken to detect a stack overflow is governed by the scheduler's tick frequency. We set the scheduler to 1000 Hz, so the maximum possible delay for overflow detection is 1 ms. Note that the detection latency is dependent on the system's CFI implementation. In our tests, killing a task in FreeRTOS takes $8\ \mu s$. The time to kill a task did not change with the stack size, the task's execution time, and the number of tasks in the system. After adjusting the deadlines of tasks in the `readyQueue`, the queue needs to be re-sorted. We use the queue implementation provided by FreeRTOS, which stores items in a linked list and uses insertion sort. Re-building the list with 12 tasks took $27\ \mu s$ in the worst case. In FreeRTOS, tasks are created with a user-specified stack size. The time required to create the recovery task scales with the allocated stack size. Creating a recovery task with a stack size of 300 words (which we used when running tasksets) took $62\ \mu s$, while creating a recovery task with a stack size of 1000 words (which we used in the evaluation), took $172\ \mu s$. While running tasksets from the schedulability tests, the worst-case latency from overflow detection to attack recovery was $151\ \mu s$, and during the evaluation, the maximum recovery time was $272\ \mu s$.

The overhead for the recovery steps depends heavily on the implementation of the task scaling, recovery task, CFI method,

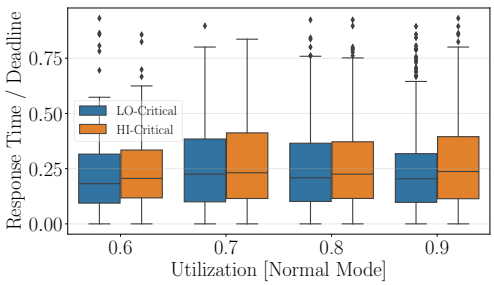


Fig. 5: *The response times of tasks from the generated tasksets when ran on our system implementation.*

and the OS being used. In our implementation, recovery time scales with the stack size of started tasks and the number of tasks on the systems. Additional overhead may occur if the recovery task performs extra checks on the system to ensure stability. The FreeRTOS stack corruption implementation used in this evaluation checks a canary value at the top of the running task's stack during scheduler ticks, which adds a minimal amount of overhead, but will not catch all attacks. A more advanced CFI mechanism may add additional overhead.

VI. Related Work

Real-time systems security gained significant attention from the community a while ago. Son *et al.* [38] first analyzed fixed-priority multi-level systems for a covert channel and revealed potential threats on scheduler-based attacks. Since then, researchers demonstrated different types of attacks [46], [45], [21], [26], [16], [42] on a regular basis necessitating the importance of secure real-time system designs. While several works [10], [31], [32], [50], [28] developed attack prevention/mitigation techniques through scheduler constraints. These techniques, in general, cannot prevent memory-based attacks. In contrast, SR^3 aims to recover a system instrumented with runtime defense from memory-based attacks.

Several existing papers developed attack detection methods modifying or adding hardware [11], software [47], [51], [52], [53], [54] to monitor the malicious activities. Besides, previous work has studied co-scheduling security monitor tasks [22], [23], [24], [25] with real-time tasks in fixed-priority partitioned multi-core systems with or without allowing migration of monitor tasks. These papers assume that the security tasks monitor security events and potentially detect the attacks (*i.e.*, works as IDS). However, IDS does not *stop* attacks; they merely attempt to detect malicious activity, while runtime defenses (*e.g.*, CFI [18], [47], [56], data flow integrity (DFI) [12]) prevent attacks from succeeding by crashing the process. Note that, unlike IDS, runtime defenses are integral to the task itself, and are *not independently scheduled.* Therefore, detection using runtime defenses is real-time and has no scheduling overhead. In SR^3, tasks are instrumented with a runtime defense instead of IDS.

Fault tolerance systems typically adopt different redundancy techniques, including the *re-execution* [6], [35] of the faulty tasks. While prior fault-tolerant works have studied re-execution for fault tolerance, there are several important distinctions in SR^3. Our recovery task models the need for additional workload to respond and recover from a malicious threat, whereas in fault tolerance, the fault is assumed benign and simply restarted [17]. Furthermore, by dropping the workload in the system in SR^3, we are reducing attack surfaces to the system at large. Also, runtime defenses detect attacks immediately, while detecting benign faults can be challenging and delayed (see Table 1 of [43] for a comparison of fault detection techniques). [5] is a work-in-progress of this paper.

VII. Conclusions

We have presented SR^3—a secure and resilient real-time recovery model, scheduler, and analysis. This model is built upon the mixed-criticality scheduling framework, where 'security' is the source of criticality instead of 'safety'. Our model is an example of a multi-mode mixed-criticality system, in which there are two operating modes, normal and recovery, and tasks are either low- or high-security criticality tasks. Additionally, to facilitate recovery from a security event, a recovery task executes during recovery mode. To avoid pessimism when adapting existing MC analysis, we developed a uniprocessor scheduling algorithm with the modified virtual deadline for each HI-security task and provided a DBF-based schedulability test. We implemented the proposed framework on FreeRTOS and evaluated it through a case study of control tasks. Finally, we experimentally show that SR^3 performs better than the existing uniprocessor scheduling schemes such as EDF and EDF-VD for mixed-criticality systems upon model transformation via simulation on synthetic workload.

Acknowledgments

This work was supported in part by the National Science Foundation under Grant CMMI 2246672 and also by the NASA Aeronautics Research Mission Directorate (ARMD) University Leadership Initiative (ULI) under cooperative agreement number 80NSSC24M0070. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Aeronautics and Space Administration.

References

[1] Freertos^{tm} real-time operating system for microcontrollers. https://www.freertos.org/. Accessed: 2022-04-07.

[2] micro-ros puts ros 2 onto microcontrollers. https://micro.ros.org/. Accessed: 2022-04-07.

[3] Ros 2 documentation. https://docs.ros.org/en/foxy/index.html. Accessed: 2022-04-07.

[4] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security*, 2005.

[5] A. Al Arafat, S. Vaidhun, B. C. Ward, and Z. Guo. A secure resilient real-time recovery model, scheduler, and analysis. *9th Workshop on Mixed Criticality Systems*, 2022.

[6] Z. Al-bayati, J. Caplan, B. H. Meyer, and H. Zeng. A four-mode model for efficient fault-tolerant mixed-criticality systems. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016.

[7] M. Antonakakis, T. April, M. Bailey, M. Bernhard, et al. Understanding the mirai botnet. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1093–1110. USENIX Association, Aug. 2017.

[8] S. Baruah, V. Bonifaci, G. DAngelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *2012 24th Euromicro Conference on Real-Time Systems*, 2012.

[9] S. Baruah, A. Mok, and L. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *[1990] Proceedings 11th Real-Time Systems Symposium*, pages 182–190, 1990.

[10] M. Bechtel and H. Yun. Denial-of-service attacks on shared cache in multicore: Analysis and prevention. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[11] N. Bellec, S. Rokicki, and I. Puaut. Attack detection through monitoring of timing deviations in embedded real-time systems. In *ECRTS 2020-32nd Euromicro Conference on Real-Time Systems*, pages 1–22, 2020.

[12] N. B. Bellec, G. Hiet, S. Rokicki, F. T. Tronel, and I. Puaut. RT-DFI: Optimizing data-flow integrity for real-time systems. In *ECRTS*, 2022.

[13] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

[14] A. Burns. Multi-model systems — an mcs by any other name. In *8th International Workshop on Mixed Criticality Systems*, 2020.

[15] N. Burow, R. Burrow, R. Khazan, H. Shrobe, and B. C. Ward. Moving target defense considerations in real-time safety-and mission-critical systems. In *7th ACM Workshop on Moving Target Defense*, 2020.

[16] C.-Y. Chen, S. Mohan, R. Pellizzoni, R. B. Bobba, and N. Kiyavash. A novel side-channel in real-time schedulers. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[17] N. Chen, S. Zhao, I. Gray, A. Burns, S. Ji, and W. Chang. Msrp-ft: Reliable resource sharing on multiprocessor mixed-criticality systems. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 201–213. IEEE, 2022.

[18] Y. Du, Z. Shen, K. Dharsee, J. Zhou, R. J. Walls, and J. Criswell. Holistic Control-Flow protection on Real-Time embedded systems with kage. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, Aug. 2022. USENIX Association.

[19] P. Ekberg and W. Yi. Bounding and shaping the demand of generalized mixed-criticality sporadic task systems. *Real-Time Syst.*, 50(1):48–86, Jan. 2014.

[20] J. Fellmuth, T. Göthel, and S. Glesner. Instruction caches in static WCET analysis of artificially diversified software. In *30th Euromicro Conference on Real-Time Systems (ECRTS)*, 2018.

[21] X. Gong and N. Kiyavash. Quantifying the information leakage in timing side channels in deterministic work-conserving schedulers. *IEEE/ACM Transactions on Networking*, 24(3):1841–1852, 2015.

[22] M. Hasan, S. Mohan, R. B. Bobba, and R. Pellizzoni. Exploring opportunistic execution for integrating security into legacy hard real-time systems. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 123–134. IEEE, 2016.

[23] M. Hasan, S. Mohan, R. Pellizzoni, and R. B. Bobba. Contego: An adaptive framework for integrating security tasks in real-time systems. *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, 2017.

[24] M. Hasan, S. Mohan, R. Pellizzoni, and R. B. Bobba. A design-space exploration for allocating security tasks in multicore real-time systems. In *Design, Automation & Test in Europe Conference & Exhibition*, 2018.

[25] M. Hasan, S. Mohan, R. Pellizzoni, and R. B. Bobba. Period adaptation for continuous security monitoring in multicore real-time systems. In *DATE*. IEEE, 2020.

[26] S. Kadloor, N. Kiyavash, and P. Venkitasubramaniam. Mitigating timing side channel in shared schedulers. *IEEE/ACM Transactions on Networking*, 24(3):1562–1573, 2015.

[27] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating Systems Principles*, pages 207–220, 2009.

[28] K. Krüger, M. Volp, and G. Fohler. Vulnerability analysis and mitigation of directed timing inference based attacks on time-triggered systems. *LIPIcs-Leibniz International Proceedings in Informatics*, 106:22, 2018.

[29] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, Jan. 1973.

[30] J. S. Mertoguno, R. M. Craven, M. S. Mickelson, and D. P. Koller. A physics-based strategy for cyber resilience of cps. In *Autonomous Systems: Sensors, Processing, and Security for Vehicles and Infrastructure 2019*. International Society for Optics and Photonics, 2019.

[31] S. Mohan, M. K. Yoon, R. Pellizzoni, and R. Bobba. Real-time systems security through scheduler constraints. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 129–140. IEEE, 2014.

[32] S. Mohan, M.-K. Yoon, R. Pellizzoni, and R. B. Bobba. Integrating security constraints into fixed priority real-time schedulers. *Real-Time Systems*, 52(5):644–674, 2016.

[33] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *ACM Conference on Programming Language Design and Implementation*, PLDI, 2009.

[34] C. Project. Memory safety, 2020.

[35] F. Reghenzani, Z. Guo, L. Santinelli, and W. Fornaciari. A mixed-criticality approach to fault tolerance: Integrating schedulability and failure requirements. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2022.

[36] G. Serra, P. Fara, G. Cicero, F. Restuccia, and A. Biondi. PAC-PL: Enabling control-flow integrity with pointer authentication in FPGA SoC platforms. In *RTAS '22*, 2022.

[37] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *14th ACM Conference on Computer and Communications Security*, page 552–561, 2007.

[38] J. Son and Alves-Foss. Covert timing channel analysis of rate monotonic real-time scheduling algorithm in mls systems. In *2006 IEEE Information Assurance Workshop*, pages 361–368. IEEE, 2006.

[39] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, 1(1):27–60, 1989.

[40] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013.

[41] G. Thomas. A proactive approach to more secure code, 2019.

[42] D. Trilla, C. Hernandez, J. Abella, and F. J. Cazorla. Cache side-channel attacks and time-predictability in high-performance critical real-time systems. In *55th Annual Design Automation Conference*, 2018.

[43] G. Upasani, X. Vera, and A. González. Avoiding core's due & sdc via acoustic wave detectors and tailored error containment and recovery. *ISCA*, 2014.

[44] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 239–243, 2007.

[45] M. Völp, B. Engel, C.-J. Hamann, and H. Härtig. On confidentiality-preserving real-time locking protocols. In *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.

[46] M. Völp, C.-J. Hamann, and H. Härtig. Avoiding timing channels in fixed-priority schedulers. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, 2008.

[47] R. J. Walls, N. F. Brown, T. Le Baron, C. A. Shue, H. Okhravi, and B. C. Ward. Control-flow integrity for real-time embedded systems. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

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

[49] B. C. Ward, R. Skowyra, C. Spensky, J. Martin, and H. Okhravi. The leakage-resilience dilemma. In *ESORICS 2019*, page 87–106.

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

[51] M.-K. Yoon, S. Mohan, J. Choi, M. Christodorescu, and L. Sha. Learning execution contexts from system call distribution for anomaly detection in smart embedded system. In *Proceedings of the Second International Conference on Internet-of-Things Design and Implementation*, 2017.

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

[53] M.-K. Yoon, S. Mohan, J. Choi, and L. Sha. Memory heat map: Anomaly detection in real-time embedded systems using memory behavior. In *52nd Design Automation Conference (DAC)*. IEEE, 2015.

[54] M. M. Z. Zadeh, M. Salem, N. Kumar, G. Cutulenco, and S. Fischmeister. Sipta: Signal processing for trace-based anomaly detection. In *2014 International Conference on Embedded Software*. IEEE, 2014.

[55] F. Zhang and A. Burns. Schedulability analysis for real-time systems with edf scheduling. *IEEE Transactions on Computers*, 58(9), 2009.

[56] J. Zhou, Y. Du, Z. Shen, L. Ma, J. Criswell, and R. J. Walls. Silhouette: Efficient protected shadow stacks for embedded systems. In *29th USENIX Security Symposium*, 2020.