Securing Real-Time Systems using Schedule Reconfiguration

Zain A. H. Hammadeh*, Monowar Hasan†, Mohammad Hamad‡

*Institute for Software Technology, German Aerospace Center, Braunschweig, Germany

†School of Electrical Engineering & Computer Science, Washington State University, Pullman, WA, USA

‡Department of Computer Engineering, Technical University of Munich, Munich, Germany

Email: Zain.HajHammadeh@dlr.de, monowar.hasan@wsu.edu, mohammad.hamad@tum.de

***Abstract*—Modern real-time systems are susceptible to cyber-attacks. The growing adoption of multi-core platforms, where safety and non-safety critical tasks coexist, further introduces new security challenges. Existing solutions suffer from either a lack of determinism or excessive cost. This paper addresses these shortcomings and proposes an offline analysis to compute all feasible schedules for real-time tasks running on a multi-core platform, isolating compromised tasks while guaranteeing a fail-operational system and low-cost *reconfigurable* scheduling. Our experimental results using a UAV autopilot system on a quad-core platform (Raspberry Pi) demonstrate that the proposed scheme incurs run-time recovery overhead at the level of microseconds. Also, the reconfiguration process covers up to 100% of all possible responses for compromised tasks in the synthetic test cases.**

***Index Terms*—Real-time Systems, Schedule Reconfiguration, Multicore, Security.**

I. Introduction

Real-time systems are integral to numerous critical applications in our everyday life, including automobiles, aerospace systems, smart grids, industrial control systems, and space rovers, to name a few. For safe operations, designers of modern real-time autonomous systems must ensure the systems function correctly and perform required tasks within predefined time bounds, often termed as *deadlines*. Due to the high value to the adversaries, use of off-the-shelf components, and wider network/Internet connectivity, modern real-time systems are also exposed to cyber breaches, as evidenced by recent attacks on automobiles, police drones, and medical robots.

A recent trend in real-time system design is to use *multi-core* chips for better performance and energy efficiency. Typically, safety and non-safety critical tasks in a multi-core platform can coexist [1]. However, sharing the same resources between critical and non-critical tasks can expose the system to certain risks, particularly regarding security attacks [2]. Compromising a non-critical task, as demonstrated by the researchers [3], could potentially allow attackers to compromise the entire system [4].

Safety-critical applications require autonomy to endure security attacks and execute in a "fail-safe" manner. Existing response options to thwart security breaches include run-time reconfiguration [5], task redundancy deployment [6], or task restart procedures [7], [8]. However, run-time reconfiguration lacks determinism — a critical requirement for real-time systems. Besides, implementing task redundancy can be prohibitively expensive and increases task response times, thus resulting in missed deadlines. In addition, restarting any malicious task (or the entire platform) does not eradicate the vulnerability, as an adversary can repeat the attack. Further, they are platform-dependent, require architectural modifications, and increase system downtime.

Due to the limitations of existing solutions, this paper investigates the following problem:

> *How can a real-time system running on a multi-core platform remain safe and operational (i.e., meet temporal constraints of all tasks) under security breaches where attackers compromise one or more tasks?*

In response to the above problem, we propose a *reconfigurable scheduling* approach (named RESCUE) that works at the software (viz., scheduler) level (i.e., does not require custom hardware). Our proposed technique *autonomously adjusts task schedules and core allocation* based on pre-computed "recovery plans." Our idea stems from the fact that rather than an immediate task restart, it could be more beneficial to accumulate comprehensive *insights* into the attacker's motives and the root cause of the attack. At the same time, the system is functional with the same level of autonomy (viz., does not miss any deadlines). This entails monitoring the compromised task to gain valuable information for devising effective attack mitigation strategies [9]. Hence, our technique will aid in systems *forensics*. Task termination becomes imperative at a certain stage due to escalated risks, a predefined threshold being reached, or sufficient information being acquired. However, one concern is ensuring the attack's repercussions do not extend to other tasks allocated to the same resource (i.e., processor cores). Thus, we need a *recovery plan*. This plan outlines the steps to reschedule these tasks onto alternative resources, ensuring the system's continued functionality and security.

Contributions. We propose an offline analysis technique (RESCUE) that *(a)* explores the space of all feasible scheduling for a given set of real-time autonomous tasks running on a multi-core platform, and *(b)* computes the necessary recovery plans invoked during a suspicion of an

979-8-3503-7128-4/24/$31.00 ©2024 IEEE

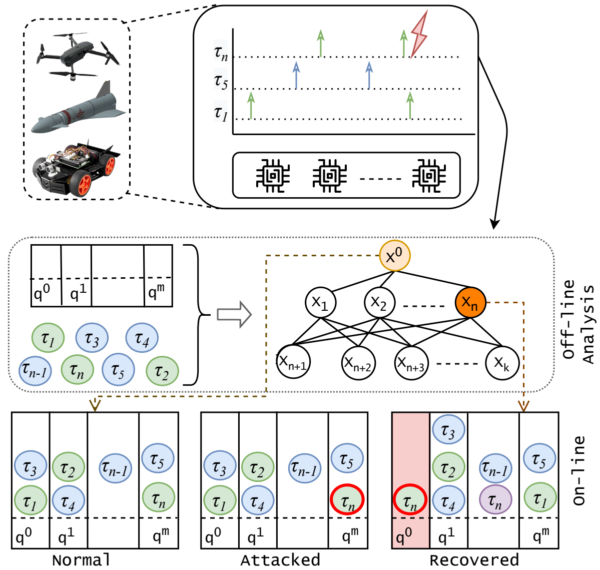


Fig. 1. Our proposed reconfiguration process (RESCUE) — offline analysis evaluates safety-critical (green) and non-safety-critical (blue) tasks and core counts, generating reconfiguration plans: a base schedule (χ^0) and compromised task responses. χ^0 remains in use until a task compromise, e.g., τ_n, triggers core q^0 isolation and activates recovery plan χ_n.

attack. Each plan (we refer to it *configuration*) represents a feasible schedule of at least the safety-critical tasks while isolating compromised tasks on other cores. Fig. 1 shows a case when the safety-critical task τ_n is got compromised. The system will be autonomously rescheduled to the recovery plan χ_n in which τ_n is isolated on core q^0, and the other tasks, including a fresh instant of τ_n are scheduled on the other three cores.

In this paper, we made the following contributions.

- An offline analysis that explores the space of feasible scheduling for different compromised tasks (§III).
- A security-aware partitioning algorithm that allows the designers to integrate security constraints (§III-C).

We performed a comprehensive evaluation using synthetic workloads and an autonomous system case study (UAV autopilot systems running on Raspberry Pi). Our approach enables run-time reconfiguration with microsecond-level overhead while covering up to 100% of potential responses for compromised tasks (§IV).

II. Model and Assumptions

A. System Model

We consider a real-time system equipped with M identical cores $Q = \{q^0, q^1, \dots, q^{M-1}\}$ running a finite set of N independent real-time tasks $\mathcal{T}$ under a real-time operating system (RTOS). Tasks are periodic and each task τ_i is defined as a tuple $\{C_i, D_i, T_i, \pi_i, \alpha_i\}$, where C_i is the worst-case execution time (WCET); D_i is the relative deadline; T_i is the period, such that $C_i \leq D_i \leq T_i$; π_i is the priority; α_i is the affinity set, which contains the indices of the cores on which τ_i can run. The affinity set can change during run-time. We consider the task level fixed-priority preemptive scheduling, i.e., each task τ_i is assigned a fixed priority π_i that does not change at run-time. The task set $\mathcal{T}$ is sorted by task priorities, i.e., $\forall i : \pi_i \leq \pi_{i+1}$ where 0 is the highest priority.

The worst-case response time R_i^+ of a task τ_i is then the maximum response time among all jobs of τ_i. We consider a constrained deadline model, i.e., $D_i \leq T_i$. A task τ_i misses its deadline if and only if: $\exists \ell \in \mathbb{N}^+ : R_i^\ell > D_i$, where R_i^ℓ is the response time of the job ℓ. All tasks $\in \mathcal{T}$ can be assigned to any core. However, the worst-case execution time of τ_i does not change.

The tasks are classified into safety-critical and non-safety-critical tasks. We denote by τ^s (resp. $\tau^{\tilde{s}}$) the safety-critical (resp. non-safety-critical) task. Consequently, we denote by $\mathcal{T}^s$ (resp. $\mathcal{T}^{\tilde{s}}$) the set of all safety-critical (resp. non-safety-critical) tasks, where

$$\mathcal{T} = \mathcal{T}^s \bigcup \mathcal{T}^{\tilde{s}} : \mathcal{T}^s = \{\tau_i^s | \forall \tau_i \in \mathcal{T}\},\ \mathcal{T}^{\tilde{s}} = \{\tau_i^{\tilde{s}} | \forall \tau_i \in \mathcal{T}\}.$$

We assign the tasks to the cores using the partitioned scheduling with a task-level fixed priority scheduling policy. Hence, $\forall \tau_i \in \mathcal{T} : |\alpha_i| = 1$. Our partitioning algorithm is implemented as Satisfiability Modulo Theories (SMT) constraints (§III-C). The system may have *security constraints (SC)* that the partitioning algorithm has to consider. For example, a security constraint may state that two tasks τ_i and τ_j with different criticality must not be allocated on the same core. The system may have any number of security constraints.

We assume that the tasks are subject to attacks. If a task got compromised, denoted $\hat{\tau}$, it will be isolated from other tasks. On the one hand, all non-safety-critical compromised tasks will be assigned to only one core q^ρ. Formally, if $\hat{\tau}_i^{\tilde{s}}$ and $\hat{\tau}_j^{\tilde{s}}$ then $\alpha_i \bigcup \alpha_j = \{q^\rho\}$. In other words, we have a best-effort approach to isolate non-safety critical tasks, such that we isolate only what can be scheduled to one core; other tasks are ignored. On the other hand, we isolate every compromised safety-critical task on one core alone. Formally: if $\hat{\tau}_i^s$ then: $\forall \tau_j \in \mathcal{T} : \alpha_i \bigcap \alpha_j = \phi$. The system has a *safe-mode*, in which only the safety-critical tasks run on some cores to guarantee a fail-operational system.

B. Threat Model

We assume that an adversary may compromise one or several specific tasks. However, we do not make explicit assumptions about the methods by which a task could become malicious. For example, an attacker might exploit existing system vulnerabilities or bugs, or use social engineering tactics [10]. The specific means by which a task can be compromised is outside the scope of this paper. Further, for multi-vendor systems, a rouge vendor could inject malicious logic that may trigger at run-time [11]. Our focus here is to "isolate" anomalous tasks and ensure other, time-critical, benign tasks can meet their temporal requirements, and hence, the system remains *safe*. We assume that the RTOS provides mechanisms that support the isolation of various processes. We further assume the existence of a detection mechanism running on a protected space (say within the kernel) that triggers the reconfiguration process when a task has been compromised.

TABLE I. A toy system configuration (seven tasks running on four cores).

Task	C_i	$D_i = T_i$	π_i	α_i	t_i^{out}	Type
τ_0	10	50	0	[1]	20	Critical
τ_1	10	50	1	[0]	20	Critical
τ_2	15	50	2	[0]	30	Critical
τ_3	40	200	3	[0]	80	NonCritical
τ_4	60	150	4	[1]	120	NonCritical
τ_5	110	1000	5	[2]	220	NonCritical
τ_6	65	400	6	[3]	130	NonCritical

III. RESCUE: Security-cognizant Scheduling

This paper proposes a reconfigurable scheduling technique, RESCUE, that *reacts* to security attacks on real-time tasks. Our goal is to reschedule the non-compromised tasks on a subset of cores to isolate the compromised task/s on the other cores while guaranteeing the real-time requirements of other tasks. The compromised task will be isolated for a predefined period.

Table I shows a real-time taskset running on a quad-core platform ($M = 4$). In the task set, τ_0 and τ_2 must not be scheduled on the same core as a design requirement. Consider that τ_3 got compromised. Letting τ_3 continue running on q^0 with the safety-critical tasks τ_1, τ_2 is dangerous. Our solution aims to *reconfigure* the system to a new configuration in which we re-schedule the non-compromised tasks on cores q^0, q^1, q^2 and isolate the compromised task τ_3 on q^3. Such isolation may save more auditing logs on the activity of τ_3 for forensics analysis.[1] When τ_3 is isolated, it cannot read/write to I/O and not interfere with any non-compromised task.

We now present how to compute the configuration, the reconfiguration methodology, and the timing analysis. We start with a few definitions.

A. Configurations

Defining a feasible schedule for the real-time tasks under partitioned scheduling requires finding feasible partitions. We must allocate each task to one core where its schedulability is guaranteed. Note that there might be more than one feasible schedule for a task set and a multi-core platform.

Definition 1. *For a given task set and a multi-core platform, a configuration, denoted by χ_l, is an allocation of the tasks to the cores such that all the real-time tasks are schedulable.*

Hence, there might be more than one configuration for a given task set and a multi-core platform. We propose an SMT-based partitioning algorithm to compute a configuration. The algorithm uses a "busy-window"-like analysis as SMT constraints (§III-C). The schedule of tasks to the cores where no compromised tasks might be given or computed using our partitioning algorithm. We refer to this configuration as the *basic configuration* χ_0.

Definition 2. *For a given task set, a combination, denoted by $\mathcal{C}_l$, is a set of tasks where each task could be compromised or not, i.e., the compromised safety-critical task will be represented by its compromised instance.*

[1]Post isolation forensics, however, is not within the scope of this work.

Definition 3. *If τ_i is not compromised in $\mathcal{C}_l$ and τ_i is also not compromised in $\mathcal{C}_k$, we say $\mathcal{C}_k \geq \mathcal{C}_l$.*

For every combination, we need to compute a new configuration. There are 2^N combinations representing a lattice equipped by the partial order $\geq$. However, we cannot find a feasible schedule for every combination because we aim to isolate the compromised tasks on some cores. Also, a minimum set of cores is needed for the safety-critical tasks to keep them running. Hence, we can compute configurations for a subset of the lattice before moving to a *safe-mode configuration*.

Definition 4. *A safe-mode configuration χ^σ is a configuration where only the safety-critical tasks, $\mathcal{T}^s$, are scheduled.*

For a combination $\mathcal{C}_l$, if there is no feasible schedule, we assign the safe-mode configuration χ^σ. The configurations also can be ordered in a similar way to the combinations: $\chi_k \geq \chi_l$ if $\mathcal{C}_k \geq \mathcal{C}_l$.

RESCUE builds on three algorithms to compute and connect the configurations in a lattice form. Fig. 2 shows the interactions between the algorithms. Alg. I computes the feasible configuration by utilizing the security-aware scheduling presented in Alg. II. Alg. III illustrates the procedure to connect the configurations to compose the final output of the offline phase of RESCUE.

Algorithm I (BuildConfigurations). We define the configuration χ_l as a data structure as Line 3 shows:

- *key* is a unique name given to χ_l;
- *comb* is the associated combination $\mathcal{C}_l$;
- *schedule* is the affinity set of the tasks according to χ_l;
- *previous* is a pointer to the previous configuration χ_k where $\chi_k \geq \chi_l$;
- *next* is a pointer to the next configuration χ_m, where $\chi_l \geq \chi_m$.

Note that χ_l may have multiple configurations in its next and previous fields. We start with computing the basic configuration χ_0 (Lines 6-10), in which we try to balance the load on the cores. The *comb* is assigned an empty set because there are no compromised tasks. Next, we compute the safe-mode configuration χ^σ (Lines 11-13). For χ^σ, we follow Definition 4 and consider only the safety-critical tasks ($\mathcal{T}^s$) and try to schedule them on the minimum number of cores. The rest of the algorithm tries to compute a feasible configuration χ_l for every combination $\mathcal{C}_l$ (Lines 15-31).

For every combination $\mathcal{C}_l$: First, we check if there is a feasible schedule of the non-compromised tasks (Line 16). We try to minimize the number of cores needed to schedule them to make room to isolate the compromised tasks. If *(a)* there is no feasible schedule or the schedule uses all the cores, and *(b)* there is no room to isolate any compromised task, we skip this combination (Lines 17-20), i.e., we assign the safe-mode configuration χ^σ to $\mathcal{C}_l$. Second, we start with isolating as much as possible of the compromised safety-critical tasks (Lines 22-27). We isolate, i.e., schedule, each compromised safety-critical task $\hat{\tau}_i^s$ on one

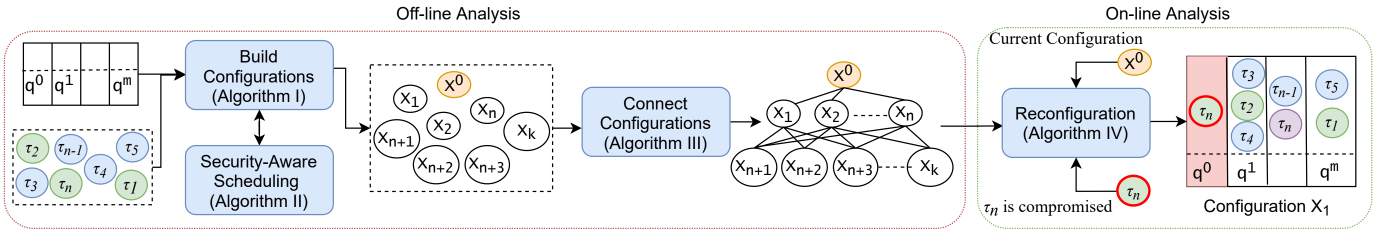


Fig. 2. Offline and online analysis phases and the various steps used in RESCUE.

core. Third, we try to isolate (schedule) as much as possible of compromised non-safety-critical tasks on one core (Lines 28, 29). Finally, we call Alg. III to connect the computed configurations.

Algorithm II (SECURITYAWARESCHED). This algorithm represents the partitioning approach and the schedulability test, which is called in five places in Alg. I. The partitioning approach and the schedulability test are security-aware and SMT-based solutions (§III-C). However, we present Alg. II here for better readability.

The algorithm can work in two modes: *balance* and *best*. In the *balance* mode, the algorithm calls the *call_smt* procedure to allocate the tasks into all cores (Lines 4, 5). This mode is used for the basic configuration χ_0. In the *best* mode, the algorithm calls the *call_smt* procedure in a loop (Lines 6-13). In each iteration, one more core is added to the set of cores given to the SMT solver. It stops once the SMT solver can find a solution, which guarantees that the tasks are scheduled on the minimum number of cores. It stops with an empty schedule, i.e., it fails if there are no more cores exist.

Lines 17-24 present the *call_smt* procedure. It builds the scheduling constraints (Lines 18-20) and the security constraints (Lines 21, 22). We call the SMT solver in Line 23.

Algorithm III (CONNECTCONFIGURATIONS). This is the last algorithm in the offline phase of RESCUE. It takes the configurations generated by Alg. I and connects them depending on the $\geq$ relation. For each configuration χ_l, we search in the lattice for a parent configuration χ_k, i.e., $\chi_k \geq \chi_l$. We update the *previous* attribute of the child (χ_l) and the *next* of the parent (χ_k), where the *key* is the index of the compromised task (Lines 5-8). The configuration χ_l may have more than one parent depending on the number of compromised tasks in $\mathcal{C}_l$. If the configuration has no parents, we delete the configuration (Lines 9, 10). For each configuration χ_l that does not have any child, it has to connect to the safe-mode configuration χ^σ, i.e., add χ^σ to $\chi_l.next$. Also, if the length of the *next* attribute of χ_l is smaller than the number of compromised tasks in $\mathcal{C}_l$ (Line 19), then there are unfeasible configurations that had to be children of χ_l. In this case, we add the χ^σ to replace the missing children. In other words, the unfeasible configurations are replaced by the safe-mode configuration.

In the example task set (Table I), the schedule of tasks to cores represents the basic configuration χ_0. In this example, there are 128 feasible configurations. If τ_3 is compromised, the system will move to the configuration χ_4 in which τ_3 is isolated on q^3. According to the proposed algorithms (Alg. I -

Algorithm I: BuildConfigurations

1 **Input:** taskset $\mathcal{T}$, cores Q, SC
2 **Output:** lattice of configurations
3 Declare struct χ {key, comb, schedule, previous, next}
4 $lattice \leftarrow \{\}$
5 $M \leftarrow |Q|$
/* create the basic configuration */
6 $sched \leftarrow SecurityAwareSched(\mathcal{T}, Q, SC, balance)$
7 **if** *not sched* **then**
8 **return** *lattice* /* not schedulable */
9 $\chi_0 \leftarrow \chi$(key=ConfigBasic, comb={}, schedule = *sched*, previous = None)
10 $lattice \leftarrow lattice \bigcup \chi^0$
/* create the safe-mode*/
11 $sched \leftarrow SecurityAwareSched(\mathcal{T}^s, Q, SC, best)$
12 $\chi^\sigma \leftarrow \chi$(key=ConfigSafe, comb=$\mathcal{T}$, schedule = *sched*, next=χ^0)
13 $lattice \leftarrow lattice \bigcup \chi^\sigma$
14 $Combs \leftarrow computeAllCombs(\mathcal{T})$
/* create other configurations */
15 **for** $\mathcal{C}_l \in Combs$ **do**
16 $sched, Q_a \leftarrow SecurityAwareSched(\mathcal{T}^s \bigcup \{\tau_i^s | \forall \tau_i \in \mathcal{C}_l\}, Q, SC, best)$
17 **if** *not sched* **then**
18 **continue** /* not schedulable */
19 **if** $Q_a = \emptyset$ **then**
20 **continue**
21 $sched^{\hat{s}} \leftarrow \{\}$
22 **for** $\hat{\tau}_i{}^s \in \{\hat{\tau}^s | \forall \hat{\tau}^s \in \mathcal{C}\}$ **do**
23 **while** $Q_a \neq \emptyset$ **do**
24 $q \leftarrow Q_a.pop$
25 $\hat{\alpha}_i{}^s \leftarrow q.index$
26 $sched^{\hat{s}} \leftarrow \hat{\alpha}_i{}^s$
27 break
28 **if** $Q_a \neq \emptyset$ **then**
29 $sched^{\hat{\tilde{s}}} \leftarrow SecurityAwareSched(\{\hat{\tau}^{\tilde{s}} | \forall \hat{\tau}^{\tilde{s}} \in \mathcal{C}\}, Q_a, best)$
30 $\chi_l \leftarrow \chi$(key=Config+$\mathcal{C}$, comb=$\mathcal{C}$, schedule = $sched \cup sched^{\hat{s}} \cup sched^{\hat{\tilde{s}}}$)
31 $lattice \leftarrow lattice \bigcup \chi$
/* connect configurations */
32 $lattice \leftarrow ConnectConfigurations(lattice)$
33 **return** *lattice*

Algorithm II: SecurityAwareSched

```
Input: taskset 𝒯, cores Q, SC, approach
Output: sched, Q
New smt_solver
if approach = balance then
    success, sched = call_smt(𝒯, Q, SC)
if approach = best then
    success ← False
    Q_u ← {}
    while not success do
        if Q = ∅ then
            break
        Q_u ← Q_u ∪ Q.pop()
        success, sched = call_smt(𝒯, Q_u, SC)
Q_a ← Q
return sched, Q_a

call_smt(𝒯, Q, SC):
for q^ρ ∈ Q do
    for τ_i ∈ 𝒯 do
        smt_solver.add_constraint(Σ_{j≤i} ⌈D_i/T_j⌉.C_j.x_j^ρ ≤ D_i)
for sc ∈ SC do
    smt_solver.add_constraint(sc)
success, sched = smt_solver.solve()
return success, sched
```

Algorithm III: ConnectConfigurations

```
Input: lattice
Output: lattice
for χ_l ∈ lattice do
    for χ_k ∈ lattice do
        if χ_k ≥ χ_l then
            key = χ_k.comb \ χ_l.comb
            χ_l.previous[key] = χ_k
            χ_k.next[key] = χ_l
    /* delete orphan configurations */
    if χ_l ≠ χ_0 & χ_l ≠ χ^σ & !χ_l.previous then
        delete χ_l
/* connect to the safe-mode */
counter ← 0
for χ_l ∈ lattice do
    if χ_l ≠ χ^σ then
        if !χ_l.next then
            χ_l.next[all] = χ^σ
            χ^σ.previous[all + counter] = χ_l
            counter ← counter + 1
        else
            if |𝒯| − |{τ̂|∀τ̂ ∈ 𝒞}| > |χ_l.next| then
                χ_l.next[all] = χ^σ
                χ^σ.previous[all + counter] = χ_l
                counter ← counter + 1
return lattice
```

Algorithm IV: Reconfiguration

```
1  Input: lattice, current config: χ, event, taskID: i
2  Output: schedule
3  χ_0 ← lattice[ConfigBasic]
4  χ^σ ← lattice[ConfigSafe]
   /* task got compromised */
5  if event is Isolate then
6      if χ.key is ConfigSafe then
7          return χ^σ.schedule
8      if i in χ.next then
9          return χ.next[i].schedule
10     else
11         return χ^σ.schedule
   /* time-out */
12 if event is Integrate then
13     if χ.key is ConfigSafe then
14         return χ_0.schedule
15     if i in χ.previous then
16         return χ.previous[i].schedule
```

Alg. III) we have: $\chi_0.next[3] = \chi_4$ and $\chi_4.previous[3] = \chi_0$.

The proposed algorithms generated the full connected lattice of configurations within 1.3 seconds. The size of the generated lattice is 4.696 kilobyte.

B. Reconfiguration Methodology

As explained earlier, the configurations, including the safe-mode configuration, are computed offline. We define a time-out for each task, denoted t_i^{out}, which designers can specify based on the application. The time-out period represents how long we keep the compromised tasks isolated before killing the job and keeping the new job of the safety-critical task or starting a fresh job of the non-safety-critical task. The reconfiguration is a procedure triggered when a task is compromised and after the time-out. Alg. IV presents the reconfiguration procedure. During the reconfiguration, we move from the configuration χ_k to χ_l after the task τ_i got compromised if $\chi_k \geq \chi_l$. Similarly, we move from the configuration χ_l to χ_k after t_i^{out}.

Algorithm IV (Reconfiguration). This algorithm represents the online phase of RESCUE and is integrated with the scheduler. The algorithm utilizes the lattice of configurations (generated by the offline phase) and an *event*. If the event is *Isolate*, i.e., τ_i got compromised, then the algorithm moves to the child configuration in the *next* attribute of the current configuration depending on the index of the compromised task i (Lines 5-11). After a timeout t_i^{out}, a new event will be triggered to *Integrate* τ_i. In this case, the algorithm moves to the parent configuration in the *previous* of the current configuration (Lines 12-16).

In our example (Table I), if τ_3 got compromised, the schedule moves to the configuration χ_4. After a timeout = $\tau_3^{out} = 80$, a new reconfiguration is triggered, in which the scheduler moves back to the configuration χ_0.

Let M_{min} denote the minimum number of cores needed to schedule all tasks; hence, $M_{min} \leq M$. We bound the number of isolated tasks using the following lemma.

Lemma 1. *The maximum number of compromised safety-critical tasks that RESCUE can isolate is* $M - M_{min}$.

Proof. We isolate each compromised safety-critical task $\hat{\tau}^s$ on one core, and we need at least M_{min} cores to schedule the task set. If some non-safety-critical tasks got compromised, we give the priority to isolate the safety-critical tasks first. □

Lemma 2. *RESCUE can cover all 2^N combinations and isolate all compromised tasks if $1 + |\mathcal{T}^S| + M_{min} \leq M$ and the compromised non-safety-critical tasks can fit in one core.*

Proof. If such an over-provisioned system exists, the proposed approach could cover all combinations and isolate all compromised task. □

An interesting indicator of the resilience of the schedule reconfigurability against security attacks is the length of the *critical path*, denoted by $\mathcal{L}$, as we formally define below.

Definition 5. *The critical path is a sequence of configurations starting from the basic configuration with no compromised tasks and ending in the safe-mode.*

The length of the critical path indicates the number of compromised tasks the system can tolerate before entering the safe-mode. This length, $\mathcal{L}$, depends on the number of cores M, the schedulability of the task set, and the security constraints. For systems not equipped with proper reaction mechanisms like RESCUE, the best case scenario is that the system will move to the safe-mode after the first attack. Hence, the length of the critical path is equal to one. We use the maximum number of compromised non-safety-critical tasks that may be excluded after a reconfiguration, i.e., neither executed nor isolated, as an indicator of the degradation of the efficiency of RESCUE, denoted by $\mathcal{D}$. In the experiments, we use these two indicators, $\mathcal{L}$ and $\mathcal{D}$ to study the efficiency of RESCUE.

C. Partitioning Algorithm

The busy-window [12] under the fixed priority preemptive (FPP) scheduling is computed as follows:

$$BW_i^{n+1} = \sum_{j \leq i} \left\lceil \frac{BW_i^n}{T_j} \right\rceil * C_j \tag{1}$$

with $BW_i^0 = C_i$. The recurrence stops at the convergence (i.e., $BW_i^{n+1} = BW_i^n$) or when when $BW_i > D_i$ (i.e., task unschedulable).

Lemma 3. *For constrained-deadline periodic tasks, τ_i meets its deadline in the worst-case if:*

$$BW_i \leq D_i \tag{2}$$

Proof. In FPP, if $BW_i \leq T_i$ then BW_i has one job and $BW_i = R_i^+$. Hence, if Eq. (2) holds, schedulability is guaranteed. □

Lemma 4. *Let us consider a fixed-size window of D_i. For constrained-deadline periodic tasks, τ_i meets its deadline if:*

$$\sum_{j \leq i} \left\lceil \frac{D_i}{T_j} \right\rceil * C_j \leq D_i \tag{3}$$

Proof. For constrained-deadline periodic tasks, if the accumulative load within a window of D_i can finish before D_i then τ_i meets its deadline, i.e., the condition in Eq. (2) is satisfied. □

The sufficient condition in Eq. (3) does not need iteration to be computed. However, it is more pessimistic than the sufficient condition in Eq. (2), because, if τ_i meets its deadline, i.e., Eq. (2) is satisfied:

$$BW_i \leq \sum_{j \leq i} \left\lceil \frac{D_i}{T_j} \right\rceil * C_j \tag{4}$$

1) SMT constraints: Now, we have a linear non-iterative sufficient schedulability condition for τ_i. To map τ_i to the core q^ρ, we have to guarantee that Eq. (3) is satisfied for all higher priority tasks already mapped to q^ρ. We can formulate the sufficient condition in Eq. (3) as a linear constraint as follows:

$$\sum_{j \leq i} \left\lceil \frac{D_i}{T_j} \right\rceil . C_j . x_j^\rho \leq D_i \tag{5}$$

where $x_j^\rho \in \{0, 1\}$ a binary variable indicates whether τ_j is mapped to the core q^ρ or not. Hence, we can define the partitioned scheduling for periodic tasks under FPP as an SMT with $M*N$ variables and constraints for M cores and N tasks.

The SMT-based partitioned scheduling has the following constraints:

$$\forall q^\rho \in Q \quad \forall \tau_i \in \mathcal{T}: \quad \sum_{j \leq i} \left\lceil \frac{D_i}{T_j} \right\rceil . C_j . x_j^\rho \leq D_i \tag{6}$$

Also, the SMT integrates security constraints, e.g., τ_i and τ_j are not allowed to run on the same core as follows:

$$\neg (x_j^\rho \wedge x_i^\rho) \tag{7}$$

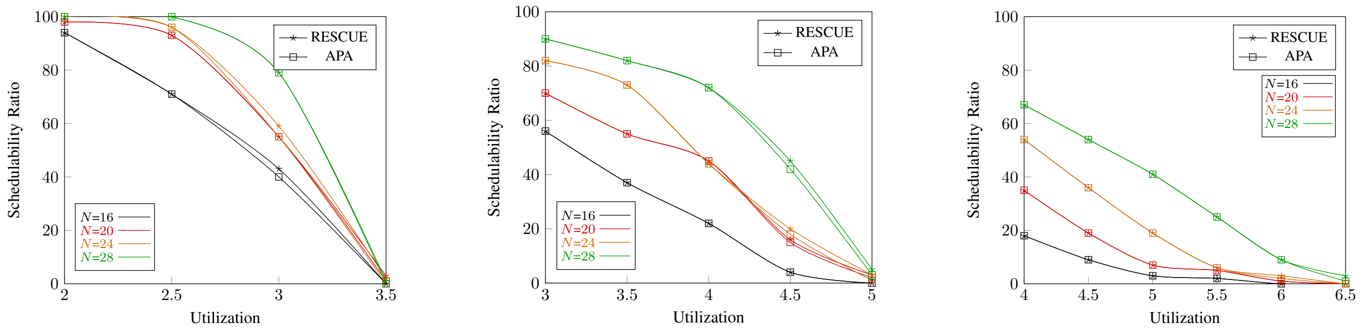
IV. Experimental Evaluation

We evaluate RESCUE on two fronts: *(a)* synthetically generated workload for broader design-space exploration (§IV-A) and *(b)* case study on a real-time platform (§IV-B).

A. Synthetic Test Cases

Workload Generation. We randomly generated synthetic test cases to carry out our experiments as follows: we distributed the utilization over the N tasks using the UUniFast algorithm [13]. Periods were generated randomly as follows: $T_i = f * p$ where $f \in \{1, 2, 3, 4, 5, 6, 7, 8\}$, and $p \in \{280, 340, 450, 500\}$. Therefore, the periods of the task set or a subset of it can be harmonic. Relative deadlines were implicit $D_i = T_i$. Priorities were assigned according to the deadline monotonic (DM) approach. The number of critical tasks was decided randomly so that it is at most 40% of the total tasks.

1) Schedudualbility of the Partitioning Algorithm: The proposed security-aware SMT-based partitioning scheduling utilizes a linear sufficient condition of schedulability using Eq. (3), which is more pessimistic than, e.g., the original busy-window analysis, i.e., Eq. (2). This experiment tests the schedulability of our partitioning algorithm compared to the Arbitrary Processor Affinity (APA) scheduling [14]. APA



(a) 4 cores

(b) 6 cores

(c) 8 cores

Fig. 3. The schedulability ratio of RESCUE vs the APA scheduling [14]. RESCUE outperforms the APA scheduling for high utilization. The schedulability ratio of RESCUE and APA scheduling decreases when U/N increases.

scheduling dominates partitioned scheduling. However, the schedulability test of APA scheduling, which depends on the response time analysis of global scheduling [15], is very pessimistic. Hence, it may report a lower schedulability ratio than the partitioned scheduling. In this experiment, we support our decision to use an SMT-based partitioned scheduling by showing the schedulability ratio of the APA scheduling, namely the heuristic-based scheduling, presented in [14], and RESCUE. We ran the experiments for $M = \{4, 6, 8\}$. Fig. 3 shows the results. We generated 100 synthetic test cases for each value of the utilization between $M/2 \leq U \leq M$ with step=0.5. Fig. 3 reports the results for $N = \{16, 20, 24, 28\}$. APA scheduling cannot outperform RESCUE. In contrast, the SMT-based partitioned scheduling in RESCUE can outperform the APA scheduling, e.g., for $U = 3.0$ when $M = 4$. The schedulability ratio of RESCUE and APA scheduling decreases when U/N increases.

2) Scalability and Efficiency of RESCUE: We generated 100 test cases for each value of the utilization between $M/2 \leq U \leq M$ with step=0.5. We repeated the experiment for $N = \{8, 10, 12, 14\}$ tasks and for $M = \{4, 6, 8\}$. As a security constraint, we considered three safety-critical tasks to run on three different cores. Fig. 4 presents the coverage ratio for $M = 4$. The plot boxes in orange (the upper figures) consider no security constraints, while the blue plot boxes report the results considering the security constraints. For all tasks, RESCUE can achieve a coverage ratio of 100% for low utilization. However, the coverage ratio drops with the high utilization values. The coverage ratio is explainable by the schedulability ratio of the security-aware SMT-based partitioning algorithm. In addition, security constraints impact the coverage ratio because they impose limitations on the scheduling options.

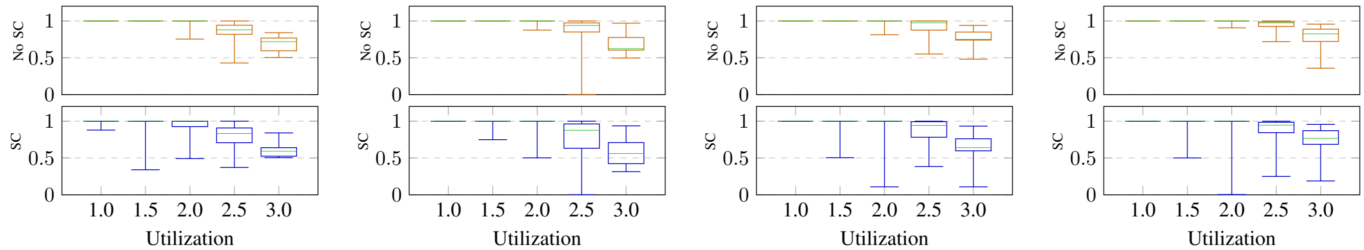
We studied the length of the critical path $\mathcal{L}$ as an indicator of the resilience of RESCUE. Fig. 5(a) shows $\mathcal{L}$ for $M = 4$. For instance, $U = 1.0$, the values are reported in order from left to right for $N = \{8, 10, 12, 14\}$. The base-line represents the value of $\mathcal{L}$ for systems that are not equipped with proper reaction mechanisms such as RESCUE. When the utilization is low, RESCUE can react to any scenario of compromising tasks. However, for large values of utilization like $U = 3.0$, RESCUE does not have many options than moving to the safe-mode.

Fig. 5(b) shows the degradation of the efficiency of RESCUE in the form of the number of compromised non-critical tasks that are neither scheduled nor isolated. In the worst-case, $\mathcal{D}$ can be equal to the number of non-safety-critical tasks. This happens when RESCUE moves to the safe-mode. However, the upper whisker ≤ 1, which means that for 75% of the test cases RESCUE misses no more than one task when configuring the system after an attack.

To study the scalability of RESCUE, we computed the execution time of the offline phase for $N = \{8, 10, 12, 14\}$ tasks and $M = \{4, 6, 8\}$. The experiment ran on a general-purpose Linux server (18-core Intel Xeon CPU and 251 GB Memory). Fig. 6 shows the execution time for $M = 4$. The sub-figures show a similar trend. Hence, we explain the results in Fig. 6(b). The median in the orange plot boxes (no security constraints) increases with the utilization. This can be explained by Alg. II (Lines 8-13), because increasing the utilization requires more cores to schedule the task set, which imposes more iterations on the *while*-loop. The decreasing median for the blue boxes, where the security constraints are considered, is mainly due to the low coverage ratio, i.e., the small size of the lattice (see Fig. 4). Also, The execution time of the SMT solver scales with the number of constraints. Hence, the SMT solver consumes more time to find a feasible partitioning for the tasks when there are security constraints.

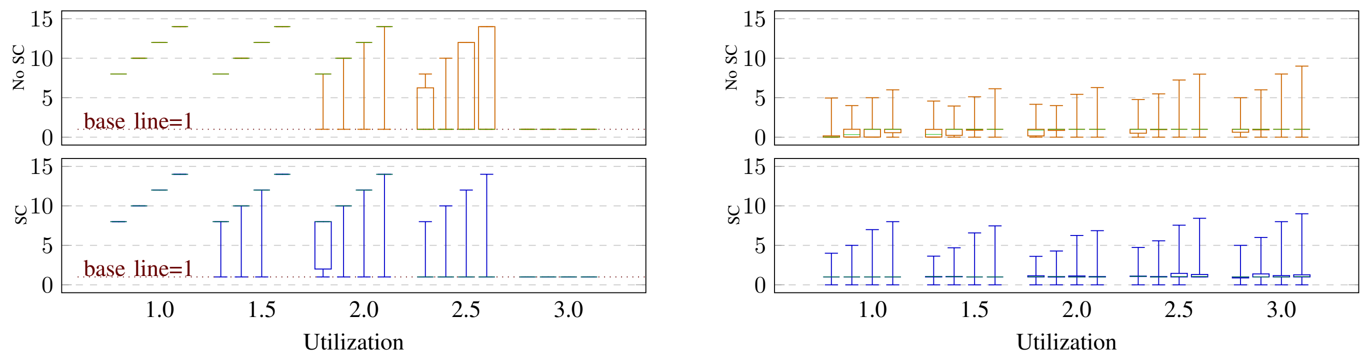
Increasing the number of cores for a given number of tasks increases the number of iterations that the `while`-loop takes in Alg. II (Lines 8-13). The main factor affecting the scalability is the number of tasks N, where the execution time increases with N. We now discuss the implementation details of RESCUE to explain this exponential growth.

RESCUE simulation engine is implemented in Python. The Algorithm BUILDCONFIGURATIONS (Alg. I) is implemented such that the `for`-loop works in parallel, utilizing all available cores. However, the Algorithm CONNECTNODES (Alg. III) is the bottleneck in RESCUE. Results show that Alg. III consumes no more than 2% of the execution time for N=8,



(a) N = 8 (b) N=10 (c) N=12 (d) N=14

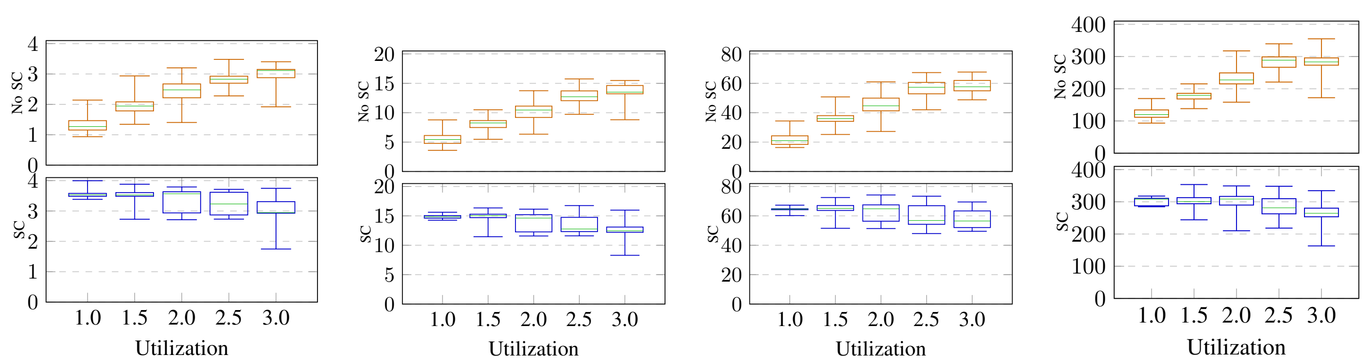
Fig. 4. Coverage ratio of RESCUE for M = 4. The coverage ratio represents the ratio between the number of the feasible configurations, i.e., the size of the lattice, over 2^N. The green line represents the median. RESCUE can achieve coverage ratio of 100% for low utilization. The coverage ratio is related to the schedulability ratio.



(a) The length of the critical path $\mathcal{L}$. The base line represents the value of $\mathcal{L}$ for systems without RESCUE.

(b) The degradation of the efficiency $\mathcal{D}$. In the worst-case, $\mathcal{D}$ can be equal to the number of non-safety-critical tasks.

Fig. 5. $\mathcal{L}$ an $\mathcal{D}$ for M = 4. For instance, U = 1.0 , the values are reported in order from left to right for $N = \{8, 10, 12, 14\}$. The green line represents the median.



(a) N=8 (b) N=10 (c) N=12 (d) N=14

Fig. 6. Execution time of the offline phase in RESCUE for M =m4. The green line represents the median. The main factor is the number of tasks N. Also, number of constraints in the SMT solver, and the utilization are the main players.

and up to 43% for N=14. For N=18, the percentage can reach 92%. Adding parallelism to Alg. III is left for future development. However, the execution time of the offline phase has no impact on the online phase.

Fig. 7 depicts the lattice size as the output of Alg. III. The reported numbers represent the maximum size of the lattice for several tasks. The experiments show that the size of the lattice has no relation with the number of cores. The size of the lattice grows exponentially with the number of tasks. However, the size of the file is acceptable for nowadays embedded systems.

Note. Although RESCUE's offline analysis scales exponentially with the number of tasks, we stress that the calculation occurs during the design phase (before the system deployment). Hence, the runtime behavior is not impacted, as the time needed to adopt and perform the reconfiguration is independent of the number of tasks and can be accomplished within a few microseconds, even for large systems.

B. *Case Study*

We use the ArduCopter[2] UAV autopilot system as a case study. Table II shows the tasks considered in this case study.

[2]https://github.com/ArduPilot/ardupilot/blob/master/ArduCopter/Copter.cpp

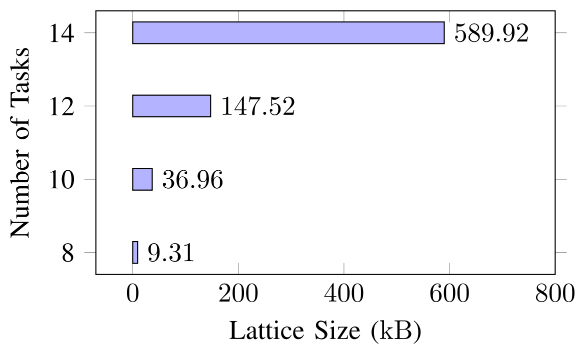


Fig. 7. The maximum lattice size in kB.

TABLE II. Task parameters for the ArduCopter UAV case study.

Task	Execution Time	Period/ Deadline	Type
rc_loop	130	4000	Critical
throttle_loop	75	20000	Critical
update_GPS	200	20000	Critical
update_optical_flow	160	5000	Critical
update_altitude	140	100000	Critical
run_nav_updates	100	20000	Critical
update_thr_average	90	10000	NonCritical
three_hz_loop	75	333333	NonCritical
compass_accumulate	100	5000	NonCritical
barometer_accumulate	90	20000	NonCritical
update_notify	90	5000	NonCritical
ekf_check	75	100000	NonCritical
landinggear_update	75	100000	NonCritical
lost_vehicle_check	50	100000	NonCritical
gcs_check_input	180	2500	NonCritical
gcs_send_heartbeat	110	5000	NonCritical

The tasks are periodic with implicit deadlines. As vanilla ArduCopter does not distinguish critical-vs-non-critical tasks, we manually inspected source code and documentation and classified them into critical and non-critical tasks based on their functionalities for our demonstration purposes. Also, we consider a quad-core platform. In this case study, the tasks are considered independent and implemented as threads.

We built a lattice of 65536 feasible configurations with 100% coverage ratio. Building the lattice took 1429.26 s. The critical path from the basic configuration to the safe-mode consists of 16 configurations, i.e., $\mathcal{L}$ = 16, including the basic and the safe-mode. Hence, compromising any combination of safety-critical and non-safety-critical tasks will lead to a feasible configuration. The system only goes to the safe-mode when the attack can compromise all tasks. The lattice size (as a file) is 2.6 MiB.

1) Attack Scenario and Findings: We consider that the task `run_nav_update` depends on the task `update_GPS` and the task `update_optical_flow`, such that compromising `update_GPS` or `update_optical_flow` will help the attacker to compromise the task `run_nav_update`, especially if they are running on the same core, i.e., sharing the same L1 cache.

Let us consider the tasks `update_optical_flow` and `run_nav_update` are compromised after t_{prop} time. If the system is not equipped with the reconfiguration mechanism, it will be in the safe-mode of the system. However, using our technique, the system will reconfigure to a feasible schedule where `update_optical_flow` and `run_nav_update` are isolated on two different cores, i.e., two new (viz., uncompromised) instances of these two safety-critical tasks will be scheduled. In particular, the following reconfigurations will take place:

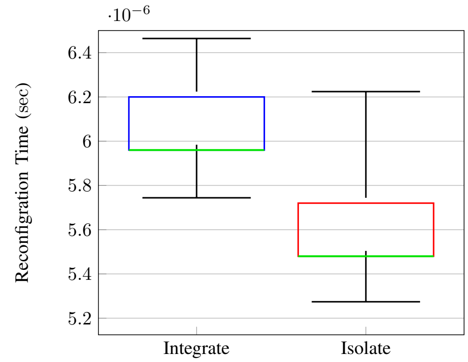


Fig. 8. The overhead of the reconfiguration procedure as measured on a multi-core embedded platform (Raspberry Pi).

- Reconfigure from the basic configuration *ConfigBasic* to *Config_1_3* when the task `update_optical_flow` got compromised; let this point of time be t_0.
- Reconfigure from *Config_1_3* to *Config_2_62* when the task `run_nav_update` got compromised; let this point of time be $t_1 > t_0$.
- After a period of time $t_0 + t^{out}_{update\_optical\_flow}$, the system moves to the configuration *Config_1_9*, in which only `run_nav_update` is isolated.
- After a period of time $t_1 + t^{out}_{run\_nav\_update}$, the system moves to *ConfigBasic*.

As we know the critical relation between the task `update_optical_flow` and `run_nav_update`, we will add a constraint on the SMT solver such that these two tasks are not allocated to the same core. We add a second constraint for `update_GPS` and `run_nav_update`. The new lattice of configurations also $\mathcal{L}$ = 16. Hence, we can reduce the probability of compromising `run_nav_update`.

2) Reconfiguration Overhead: To measure the reconfiguration overhead, we tested the reconfiguration procedure (Alg. IV) on an embedded system (Raspberry Pi3) with quad-core Cortex-A72 (ARM v8) 64-bit SoC, working at a speed of 1.8 GHz. We repeated the test 3000 times and computed the execution time. Fig. 8 shows the result. The overhead is negligible as it is at the microsecond level, which could be acceptable for many real-time autonomous applications.

V. Related Work

Researchers explore real-time security problems from various contexts [16]. Hasan et al. propose scheduler-level techniques to integrate security monitoring tasks [17], [18]. However, unlike ours, such approaches do not consider the aftereffect of detecting an anomalous task. There exists architecture-level solutions [8], [19], [20] that proactively reset the system to remove malicious entities. As we mention in §I, such techniques may not be feasible for some use cases. Further, they need custom hardware. In contrast, we propose scheduler-level defense techniques. There are also efforts to integrate security mechanisms for both fixed [9], [21], [22] and dynamic priority [23], [24] systems. However, they are designed for single-core platforms only. Pre-computed recovery planes for distributed systems can improve the availability of the system in case of single computing node failure [25]. Neither security nor multi-core aspects are covered in earlier research [25]. To the best of our knowledge, this is one of the first efforts to introduce the notion of "schedule reconfigurability" to isolate anomalous tasks and thus ensure safety while retaining temporal guarantees in *multi-core* platforms.

VI. Conclusion

Threats to critical real-time systems are growing, and traditional security measures designed for more general-purpose systems are not always sufficient. Safeguarding systems with safety-critical and non-safety-critical tasks that share resources is a challenging security problem. The increasing trend of consolidating tasks onto multi-core platforms further amplifies this concern. This paper presents a scheduler-level defense approach, RESCUE, involving pre-computed recovery plans. Through comprehensive evaluations, including synthetic workload and UAV case study, we show the effectiveness of RESCUE. Our reactive reconfiguration approach enhances security and resilience, especially those that use multi-core chips, such as time-critical IoT applications and autonomous systems. While our immediate focus here is on the reconfiguration of tasks in response to cyberattacks, similar ideas can be expanded to address other problems, such as safety failures.

Acknowledgment

This paper is supported in part by the European Union-funded project CyberSecDome (Agreement 101120779) and the US National Science Foundation (Award 2312006). Any findings, opinions, recommendations, or conclusions expressed in the paper are those of the authors and do not necessarily reflect the views of sponsors.

References

[1] R. Ernst and M. Di Natale, "Mixed criticality systems—a history of misconceptions?" *IEEE Design & Test*, vol. 33, no. 5, pp. 65–74, 2016.

[2] M. Bechtel and H. Yun, "Memory-aware denial-of-service attacks on shared cache in multicore real-time systems," *IEEE Transactions on Computers*, vol. 71, no. 9, pp. 2351–2357, 2022.

[3] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno, "Comprehensive experimental analyses of automotive attack surfaces," in *20th USENIX security symposium (USENIX Security 11)*, 2011.

[4] 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.

[5] M. Hamad, M. Tsantekidis, and V. Prevelakis, "Red-zone: Towards an intrusion response framework for intra-vehicle system." in *VEHITS*, 2019.

[6] L. Miedema, B. Rouxel, and C. Grelck, "Task-level redundancy vs instruction-level redundancy against single event upsets in real-time dag scheduling," in *2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, 2021, pp. 373–380.

[7] E. Dubrova, *Fault-tolerant design*. Springer, 2013.

[8] F. Abdi, C.-Y. Chen, M. Hasan, S. Liu, S. Mohan, and M. Caccamo, "Guaranteed physical security with restart-based design for cyber-physical systems," in *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 2018, pp. 10–21.

[9] M. Hamad, Z. A. Hammadeh, S. Saidi, V. Prevelakis, and R. Ernst, "Prediction of abnormal temporal behavior in real-time systems," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, 2018, pp. 359–367.

[10] 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 *2017 IEEE/ACM Second International Conference on Internet-of-Things Design and Implementation (IoTDI)*, 2017, pp. 191–196.

[11] C.-Y. Chen, M. Hasan, and S. Mohan, "Securing real-time Internet-of-things," *Sensors*, vol. 18, no. 12, 2018.

[12] J. P. Lehoczky, "Fixed priority scheduling of periodic task sets with arbitrary deadlines," in *IEEE Real-Time Systems Symposium*, 1990.

[13] E. Bini and G. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1-2, 2005.

[14] A. Gujarati, F. Cerqueira, and B. B. Brandenburg, "Outstanding paper award: Schedulability analysis of the linux push and pull scheduler with arbitrary processor affinities," in *2013 25th Euromicro Conference on Real-Time Systems*, July 2013, pp. 69–79.

[15] M. Bertogna and M. Cirinei, "Response-time analysis for globally scheduled symmetric multiprocessor platforms," in *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, 2007, pp. 149–160.

[16] M. Hasan, A. Kashinath, C.-Y. Chen, and S. Mohan, "SoK: Security in real-time systems," *ACM Comput. Surv.*, Feb. 2024.

[17] M. Hasan, S. Mohan, R. Pellizzoni, and R. B. Bobba, "Period adaptation for continuous security monitoring in multicore real-time systems," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020, pp. 430–435.

[18] M. Hasan, S. Mohan, R. Pellizzoni, and R. B. Bobba, "A design-space exploration for allocating security tasks in multicore real-time systems," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 225–230.

[19] L. Niu, D. Sahabandu, A. Clark, and R. Poovendran, "Verifying safety for resilient cyber-physical systems via reactive software restart," in *2022 ACM/IEEE 13th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 2022, pp. 104–115.

[20] R. Romagnoli, B. H. Krogh, D. de Niz, A. D. Hristozov, and B. Sinopoli, "Runtime system support for cps software rejuvenation," *IEEE Transactions on Emerging Topics in Computing*, 2023.

[21] S. Mohan, M.-K. Yoon, R. Pellizzoni, and R. B. Bobba, "Integrating security constraints into fixed priority real-time schedulers," *Real-Time Systems*, vol. 52, pp. 644–674, 2016.

[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)*. IEEE, 2016, pp. 123–134.

[23] V. Lesi, I. Jovanov, and M. Pajic, "Security-aware scheduling of embedded control tasks," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, pp. 1–21, 2017.

[24] S. Baruah, "Security-cognizant real-time scheduling," in *2022 IEEE 25th International Symposium On Real-Time Distributed Computing (ISORC)*. IEEE, 2022, pp. 1–9.

[25] A. Lund, Z. A. Haj Hammadeh, P. Kenny, V. Vishav, A. Kovalov, H. Watolla, A. Gerndt, and D. Lüdtke, "Scosa system software: the reliable and scalable middleware for a heterogeneous and distributed on-board computer architecture," *CEAS Space Journal*, vol. 14, no. 1, pp. 161–171, 2022.