Period Adaptation for Continuous Security Monitoring in Multicore Real-Time Systems

Monowar Hasan*, Sibin Mohan*, Rodolfo Pellizzoni† and Rakesh B. Bobba‡
Email: {*mhasan11, *sibin}@illinois.edu, †rodolfo.pellizzoni@uwaterloo.ca, ‡rakesh.bobba@oregonstate.edu

***Abstract*—We propose HYDRA-C, a design-time evaluation framework for integrating monitoring mechanisms in multicore real-time systems (RTS). Our goal is to ensure that security (or other monitoring) mechanisms execute in a "continuous" manner – *i.e.,* as often as possible, across cores. This is to ensure that any such mechanisms run with few interruptions, if any. HYDRA-C is intended to allow designers of RTS to integrate monitoring mechanisms without perturbing existing timing properties or execution orders. We demonstrate the framework using a proof-of-concept implementation with intrusion detection mechanisms as security tasks. We develop and use both, *(a)* a custom intrusion detection system (IDS) as well as *(b)* Tripwire – an open source data integrity checking tool. We compare the performance of HYDRA-C with a state-of-the-art multicore RT security integration approach and find that our method does not impact the schedulability and, on average, can detect intrusions 19.05% faster without impacting the performance of RT tasks.**

I. Introduction

Multicore processors have found increased use in the design of modern real-time systems (RTS) [1]. However, the use of such processors increases the security problems (*e.g.,* due to parallel execution of critical tasks) [2]. Successful attacks/intrusions into RTS are often aimed at impacting the safety guarantees of such systems, as evidenced by recent intrusions (*e.g.,* attacks on control systems [3], automobiles [4], medical devices [5], *etc.* to name but a few). In this paper we evaluate design alternatives to improve the security posture of RTS through integration of 'security tasks' while ensuring that the existing real-time (RT) tasks are not affected by such integration. The security tasks could be carrying out any one of protection, detection or response-based operations, depending on the system requirements. In Table I we present some examples of security tasks that can be integrated into RTS (this is by no stretch meant to be an exhaustive list). Integrating such tasks into multicore platforms is more challenging since designers have multiple choices to retrofit security tasks. For instance, is it better to *statically partition cores for security tasks* or is it better to *execute them continuously across any available core* (in conjunction with the RT tasks), and if so, *how to determine their periods*?

Our main goal is to explore design mechanisms that can *raise the responsiveness of such monitoring tasks by increasing their frequency of execution*. For instance, consider an intrusion detection system (IDS) *e.g.,* that checks the integrity of file systems. If such a system is interrupted (before it can complete entire checking), then an adversary could use that opportunity to intrude into the system and, perhaps, stay resident in the part of the filesystem that has already been checked. If, on the other hand, the IDS task is able to execute with as few interruptions as possible (*e.g.,* by moving immediately to an empty core when it is interrupted), then there is much higher chance of successful detection and correspondingly, a much lower chance of successful adversarial action.

TABLE I
Example of Security Tasks*

Security Task	Approach/Tools
File-system checking	Tripwire [6], AIDE [7], *etc.*
Network packet monitoring	Bro [8], Snort [9], *etc.*
Hardware event monitoring	Statistical analysis based checks [10] using performance monitors (*e.g.,* perf [11], OProfile [12], *etc.*)
Application specific checking	Behavior-based detection [13], [14]

*Note: We do not target our framework towards any specific security mechanism – our focus is to integrate any designer-provided security technique into a multicore-based RTS. We used *Tripwire* and our *in-house custom-developed malicious kernel module checker* to demonstrate the feasibility of our approach (§IV) – the solutions proposed in this paper is more broadly applicable to other security mechanisms.

In this paper we present a design-time framework (named HYDRA-C) for partitioned[1] RTS that enables *continuous execution* of security tasks (*i.e.,* execute as frequently as possible) across cores, without impacting schedulability of existing RT tasks. HYDRA-C extends our existing work [16] (that uses a partitioned scheduling approach and does not allow runtime migration) to ensure better security (*e.g.,* faster detection time) and schedulability. We also present an implementation on a realistic ARM-based multicore rover platform (§IV-A) and carry out a design space exploration to study the trade-offs for schedulability and security (§IV-B). Our evaluation shows that proposed approach can achieve better execution frequency (consequently quicker intrusion detection) when compared with both fully-partitioned and global scheduling approaches while providing same or better schedulability.

II. Model and Assumptions

A. Real-time Tasks and Scheduling Model

Consider a set of N_R RT tasks $\Gamma_R = \{\tau_1, \tau_2, \cdots, \tau_{N_R}\}$, scheduled on a multicore platform with M identical cores $\mathcal{M} = \{\pi_1, \pi_2, \cdots, \pi_M\}$. Each RT task τ_r is represented by the tuple (C_r, T_r, D_r) where C_r is the worst-case execution time (WCET), T_r is the minimum inter-arrival time (*e.g.,* period) and D_r is the relative deadline. We assume constrained deadlines for RT tasks (*e.g.,* $D_r \leq T_r$) and the task priorities

The material in this paper is based upon work supported in part by the National Science Foundation (NSF) grant number SaTC 1718952 and by the Natural Sciences and Engineering Research Council (NSERC).

[1]Since this is the commonly used multicore scheduling approach for many commercial and open-source OSs – mainly due to its simplicity and efficiency [15], [16].

are assigned according to rate-monotonic (RM) [17] order. All events in the system happen with the precision of integer clock ticks. RT tasks are scheduled using partitioned fixed-priority preemptive scheme [1], [15]. We further assume that the RT tasks are *schedulable*, *viz.,* the worst-case response time (WCRT), denoted as $\mathcal{R}_r$, is less than deadline.

B. Security Model

Our focus is on integrating security mechanisms (abstracted as security tasks) into an existing (legacy) multicore RTS without impacting its RT functionalities. While we use specific mechanisms (*e.g.,* Tripwire) to demonstrate our approach, it is somewhat agnostic to the security mechanisms. The security model used and the design of security tasks are orthogonal problems. Since we aim to maximize the frequency of execution of such tasks, mechanisms whose performance improves with the frequency of execution (*e.g.,* intrusion monitoring and detection tasks) benefit the most from our approach.

C. Security Task Integration

We propose to improve the security posture by integrating additional N_S periodic security tasks $\Gamma_S = \{\tau_1, \tau_2, \cdots, \tau_{N_S}\}$ (*e.g.,* tasks that are specifically designed for monitoring purposes) – a common approach for RT security integration frameworks [16], [18], [19]. HYDRA-C also leverages *opportunistic execution* [16], [18], *i.e.,* security tasks will only execute during the slack time (*e.g.,* when a core is idle) and the timing requirements of existing RT tasks will not be perturbed. However, in contrast to existing work (called HYDRA) [16] where the security tasks are statically bound to their respective cores, in this paper we allow security tasks to continuously migrate at runtime whenever any core is available (*e.g.,* when other RT or higher-priority security tasks are not running). As we shall see in §IV, allowing security tasks to execute on any available core will give us the opportunity to execute them more frequently and that leads to better responsiveness (faster intrusion detection time).

We adopt the periodic security task model [18] and represent each security task as (C_s, T_s, T_s^{max}) where T_s is the unknown period and T_s^{max} is a designer provided upper bound of the period – if the period of the security task is larger than T_s^{max} then the responsiveness is too low and security checking may not be effective. We assume that the priorities of of the security tasks are distinct and specified by the designers (*e.g.,* derived from specific security requirements). These tasks have implicit deadlines, *i.e.,* they need to finish execution before the next invocation. We also assume that task migration and context switch overhead is negligible compared to the WCETs.

III. Period Selection

One fundamental question is to figure out *how often to execute security tasks* so that the system remains schedulable and also can execute within a designer provided frequency bound (so that the security checking remains effective).[2] Mathematically period selection can be expressed as: $\underset{T_s, \forall \tau_s \in \Gamma_S}{\text{minimize}} \sum_{\tau_s \in \Gamma_S} T_s$, subject to $\mathcal{R}_s \leq T_s \leq T_s^{max}, \forall \tau_s \in \Gamma_S$. This is a non-trivial optimization problem since the period of τ_s can be anything in $[\mathcal{R}_s, T_s^{max}]$ and the response time $\mathcal{R}_s$ is a variable as it depends on the period of other higher priority security tasks. We first derive the WCRT of the security tasks (§III-A1) and use it as a (lower) bound to find the periods (§III-B).

A. Response Time Analysis

In the following we determine the response time of a job τ_s^k of security task τ_s using an iterative method and the response time in each iteration is denoted by x.

1) Interference Caused by RT Tasks: The interference $I_{\tau_s \leftarrow \tau_i}$ caused by a task τ_i on τ_s^k is the number of time units in the busy period[3] when τ_i executes while τ_s^k does not. We first calculate the *workload*[4] of the RT tasks using the following lemma and use this to derive the interference.

Lemma 1. *The maximum workload of RT tasks executed on a given core π_m (in any possible time interval of length x) is obtained when all RT tasks are released synchronously at the beginning of the interval.*

Since RT tasks are statically partitioned to cores and they have higher priority than any task that is allowed to migrate between cores, their worst-case workload can be obtained based on the critical instant [17] used for single-core fixed-priority scheduling case (formal proof in Appendix).

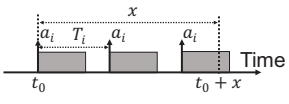


Fig. 1. Workload of the RT tasks for a window of size x. a_i denotes the arrival time.

Let $\Gamma_R^{\pi_m} \subseteq \Gamma_R$ denote the set of RT tasks partitioned to core π_m. Based on Lemma 1, an upper bound to the workload of RT tasks on π_m can be obtained by assuming that each RT task τ_r is released at the beginning of the interval and each job of τ_r executes as early as possible after being released (see Fig. 1). We thus obtain the workload for RT task τ_r: $W_r^R(x) = \left\lfloor \frac{x}{T_r} \right\rfloor C_r + \min(x \bmod T_r, C_r)$ and summing over all RT tasks on π_m yields a total workload $\sum_{\tau_i \in \Gamma_R^{\pi_m}} W_i^R(x)$. Note that by definition, the interference caused by a group of tasks executing on the same core π_m on τ_s cannot be greater than $x - C_s + 1$. Therefore, the maximum interference caused by RT tasks can be bounded as: $I_{\tau_s \leftarrow \Gamma_R^{\pi_m}}\big(x, \sum_{\tau_i \in \Gamma_R^{\pi_m}} W_i^R(x)\big) = \min\big(\sum_{\tau_i \in \Gamma_R^{\pi_m}} W_i^R(x), x - C_s + 1\big)$.

2) Interference Caused by Other Security Tasks: We next consider the workload of security tasks with higher priority than τ_s. The workload computation for this case depends on the arrival time of the task relative to the beginning of the busy period. Let us define a task τ_i as a *carry-in* task (CI) if there exists one job of τ_i that has been released before the beginning of a given time window of length x and executes within the window. If no such job exists, τ_i is referred to as a *non-carry-in* task (NC).

[2]This is different when compared to scheduling traditional RT tasks since the RT task parameters (*e.g.,* periods) are often derived from physical system properties and cannot be adjusted due to control/application requirements.

[3]This is the maximal continuous time interval $[t_1, t_2)$ until τ_s^k finishes where all the cores are executing either higher priority tasks or τ_s^k itself.

[4]The workload $W_i(w)$ of a task τ_i in a window of length w represents the accumulated execution time of τ_i within this time interval [20].

To calculate the number of carry-in tasks, we extend the busy period of τ_s^k from its arrival time (denoted by a_s) to an earlier time instance t_0 (see Fig. 2) such that during any time instance $t \in [t_0, a_s)$ all cores are busy executing tasks with higher priority than τ_s [20]. Note that by definition, this implies that there was at least one free core (*i.e.,* not executing higher priority tasks) at time $t_0 - 1$.

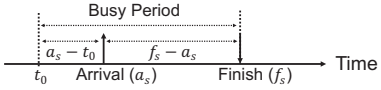


Fig. 2. Busy period extension.

Lemma 2. *At most $M-1$ higher priority tasks can have carry-in at time t_0.*

Proof. The maximum number of higher priority tasks that can have carry-in at t_0 is $M-1$ since by definition there have to be strictly less than M higher priority tasks active at time $t_0 - 1$ (otherwise they will occupy all the cores). □

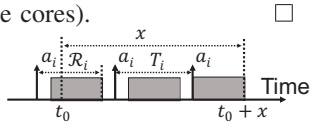


Fig. 3. Illustration of carry-in task for a window of size x.

Since Lemma 2 holds for all tasks with higher priority than τ_s, an immediate corollary is that the number of security tasks with carry-in at t_0 also cannot be larger than $M-1$. If a security task τ_i does not have carry-in, its workload is maximized when the task is released at the beginning of the busy interval. Hence, we can calculate the workload bound $W_i^{S_{NC}}(x)$ for the interval x as follows: $W_i^{S_{NC}}(x) = \left\lfloor \frac{x}{T_i} \right\rfloor C_i + \min(x \bmod T_i, C_i)$. Likewise, the workload bound for a carry-in security task τ_i in an interval of length x starting at t_0 is given by (see Fig. 3): $W_i^{S_{CI}}(x) = W_i^{S_{NC}}(\max(x - \bar{x}_i, 0)) + \min(x, C_i - 1)$, where $\bar{x}_i = C_i - 1 + T_i - \mathcal{R}_i$. We can bound the workload of the first carry-in job to $C_i - 1$ because the job must have started executing at the latest at $t_0 - 1$ (given that not all cores are busy). Finally, using the same argument as in §III-A1, the interference of τ_i can be bounded as follows: $I_{\tau_s \leftarrow \tau_i}(x, W_i(x)) = \min\left(W_i(x), x - C_s + 1\right)$, where $W_i(x)$ is either $W_i^{S_{NC}}(x)$ or $W_i^{S_{CI}}(x)$. Notice that the WCRT and periods of security task in the carry-in workload function is actually an unknown parameter. However, we follow an iterative scheme (§III-B) that allows us to calculate the period and WCRT of all higher priority security tasks before we calculate the interference for task τ_s.

3) Response Time Calculation: Let $hp_S(\tau_s)$ denote the set of security tasks with a higher priority than τ_s. Note that we do not know which (at most) $M-1$ security tasks in $hp_S(\tau_s)$ have carry-in. In order to derive the WCRT of τ_s, let us define $\mathcal{Z}_{\tau_s} \subset \Gamma \times \Gamma$ as the set of all partitions of $hp_S(\tau_s)$ into two subsets Γ_s^{NC} and Γ_s^{CI} (*i.e.,* the non overlapping set of carry-in and non-carry-in tasks) such that: $\Gamma_s^{NC} \cap \Gamma_s^{CI} = \emptyset, \Gamma_s^{NC} \cup \Gamma_s^{CI} = hp_S(\tau_s)$, and $|\Gamma_s^{CI}| \leq M-1$.

For a given carry-in and non-carry-in set (*i.e.,* Γ_s^{NC} and Γ_s^{CI}), we can calculate the total interference experienced by τ_s as follows: $\Omega_s(x, \Gamma_s^{NC}, \Gamma_s^{CI}) = \sum_{\pi_m \in \mathcal{M}} I_{\tau_s \leftarrow \Gamma_R^{\pi_m}}\Big(x, \sum_{\tau_i \in \Gamma_R^{\pi_m}} W_i^R(x)\Big) + \sum_{\tau_i \in \Gamma_s^{NC}} I_{\tau_s \leftarrow \tau_i}\Big(x, W_i^{S_{NC}}(x)\Big) + \sum_{\tau_i \in \Gamma_s^{CI}} I_{\tau_s \leftarrow \tau_i}\Big(x, W_i^{S_{CI}}(x)\Big)$. The response time $\mathcal{R}_{s|(\Gamma_s^{NC}, \Gamma_s^{CI})}$ then will be the minimal solution of the following iteration[5] [20]: $x = \left\lfloor \frac{\Omega_s(x, \Gamma_s^{NC}, \Gamma_s^{CI})}{M} \right\rfloor + C_s$. We can solve this using an iterative fixed-point search with the initial condition $x^{(0)} = C_s$. The search terminates if there exists a solution (*i.e.,* $x = x^{(l)} = x^{(l-1)}$ for some iteration l) or when $x^{(l)} > T_s^{max}$ for any iteration l since τ_s becomes trivially unschedulable for WCRT greater than T_s^{max}. Finally we can calculate the WCRT of τ_s as follows: $\mathcal{R}_s = \max_{(\Gamma_s^{NC}, \Gamma_s^{CI}) \in \mathcal{Z}_{\tau_s}} \mathcal{R}_{s|(\Gamma_s^{NC}, \Gamma_s^{CI})}$.

Algorithm 1 Period Selection

Input: Set of real-time and security tasks $\Gamma = \Gamma_R \cup \Gamma_S$
Output: Periods of the security tasks, $\mathbf{T}$ (if the security tasks are schedulable); Unschedulable otherwise

1: Set $T_s := T_s^{max}$ and calculate $\mathcal{R}_s$ for $\forall \tau_s \in \Gamma_S$
2: **if** $\exists \tau_s$ such that $\mathcal{R}_s > T_s^{max}$ **then**
3: **return** Unschedulable
4: **end if**
5: **for each** security task $\tau_s \in \Gamma_S$ (from higher to lower priority) **do**
6: */* Find period for which all lower priority tasks are schedulable */*
7: Find minimum $T_s^* \in [\mathcal{R}_s, T_s^{max}]$ using logarithmic search such that all low priority task τ_j remains schedulable (*i.e.,* $\mathcal{R}_j \leq T_j^{max}, \forall \tau_j$)
8: **end for**
9: **return** $\mathbf{T} := [T_s^*]_{\forall \tau_s \in \Gamma_S}$ */* return the periods */*

B. Algorithm

The security task τ_s remains schedulable with any period $T_s \in [\mathcal{R}_s, T_s^{max}]$. However as mentioned earlier, the calculation of $\mathcal{R}_s$ requires us to know the period and response time of other high priority tasks $\tau_h \in hp_S(\tau_s)$. Also if we arbitrarily set $T_s = \mathcal{R}_s$ (since this allows us to execute security tasks more frequently) it may negatively affect the schedulability of other tasks that are at a lower priority than τ_s because of a high degree of interference from τ_s. Hence, we developed an iterative algorithm that trades-off between schedulability and monitoring frequency.

Our proposed solution (Algorithm 1) works as follows. We first fix the period of each security task T_s^{max} and calculate the response time $\mathcal{R}_s$ (Line 1). If there exists a task τ_j such that $\mathcal{R}_j > T_j^{max}$ we report the taskset as unschedulable (Line 3) since it is not possible to find a period for the security tasks within the designer provided bounds – this unschedulability result will help the designer in modifying the requirements (and perhaps RT tasks' parameters, if possible) accordingly to integrate monitoring tasks for the target system. If the taskset is schedulable with T_s^{max}, we then optimize the periods from higher to lower priority order (Lines 5-8) and return the period (Line 9). To be specific, for each task $\tau_s \in \Gamma_S$ we perform a logarithmic search and find the minimum period T_s^* within the range $[R_s, T_s^{max}]$ such that all low priority tasks (denoted as $lp(\tau_s)$) remain schedulable, *e.g.,* $\forall \tau_j \in lp(\tau_s) : \mathcal{R}_j \leq T_j^{max}$ (Line 7) and repeat the search for next security task.

IV. Evaluation

We evaluate HYDRA-C on two fronts[6]: *(i)* a proof-of-concept implementation on an ARM-based rover platform with security applications – to demonstrate the viability of our scheme in a realistic setup (§IV-A); and *(ii)* with synthetically generated workloads for broader design-space exploration (§IV-B).

[5]Note that the worst-case is when the job arrives at t_0 (*i.e.,* $a_s = t_0$).

[6]Our implementation is available in a public, open-sourced repository [21].

TABLE II
SUMMARY OF THE EVALUATION PLATFORM

Artifact	Configuration/Tools
Platform	1.2 GHz 64-bit Broadcom BCM2837 (Raspberry Pi 3)
CPU	ARM Cortex-A53
Memory	1 Gigabyte
Operating System	Debian Linux (Raspbian Stretch Lite)
Kernel version	Linux Kernel 4.9
Real-time patch	PREEMPT_RT 4.9.80-rt62-v7+
Kernel flags	`CONFIG_PREEMPT_RT_FULL` enabled
Boot parameters	`maxcpus=2`, `force_turbo=1`, `arm_freq=700`, `arm_freq_min=700`
WCET measurement	ARM cycle counter registers
Task partition	Linux `taskset`

A. Experiment with an Embedded Platform

We implemented our ideas on a rover platform (Fig. 4). The rover peripherals (*e.g.,* wheel, motor, servo, sensor) are controlled by a Raspberry Pi [22] single board computer. We used Linux kernel 4.9 and enabled RT capabilities by applying the PREEMPT_RT patch [23] (version 4.9.80-rt62-v7+). Our experiments were performed on a dual-core setup – this was done by setting the flag `maxcpus=2` in the boot command file `/boot/cmdline.txt`. In our experiments the rover moved around autonomously and periodically captured and stored images. We assumed implicit deadlines for RT tasks and considered two RT tasks: *(a)* a navigation task – that avoids obstacles using an infrared sensor and navigates (*e.g.,* both driving and path-planning) the rover and *(b)* a camera task that captures and stores still images. Parameters for the navigation and camera tasks were (C_r, T_r): $(240, 500)$ ms and $(1120, 5000)$ ms, respectively (*i.e.,* total RT task utilization was 0.7040).

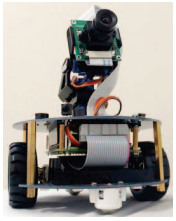


Fig. 4. Rover used in our experiments.

We introduced two security tasks: *(a)* an open-source security application, Tripwire [6], that checks intrusions in the image data-store and *(b)* our custom security task that checks current kernel modules (for detecting rootkits) and compares with an expected profile of modules. We modified Tripwire configurations (`/etc/tripwire/twpol.txt`) and retrofitted it into periodic execution model. The WCET of the security tasks were 5342 ms and 223 ms, respectively and the maximum periods[7] of security tasks were assumed to be 10000 ms (*e.g.,* total system utilization is at least $0.7040 + 0.5565 = 1.2605$). The system configurations and tools used in our experiments are summarized in Table II.

We compared the performance of HYDRA-C with our prior work (HYDRA) [16] where we proposed to statically partition the security tasks among the multiple cores – to the best of our knowledge that paper is the state-of-the-art mechanism for integrating security in legacy multicore-based RT platforms. The key idea in our prior work was to allocate security tasks using a greedy best-fit strategy: for each task, allocate it to a core that gives shorter period without violating schedulability constraints of already allocated tasks.

[7]We picked this maximum period value by trial and error so that the taskset became schedulable for demonstration purposes.

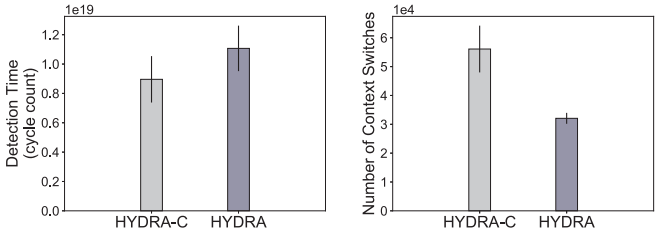


Fig. 5. Experiments with rover platform: *(a)* time (cycle counts) to detect intrusions; *(b)* average number of context switches. On average our scheme can detect the intrusions faster without impacting the performance of RT tasks.

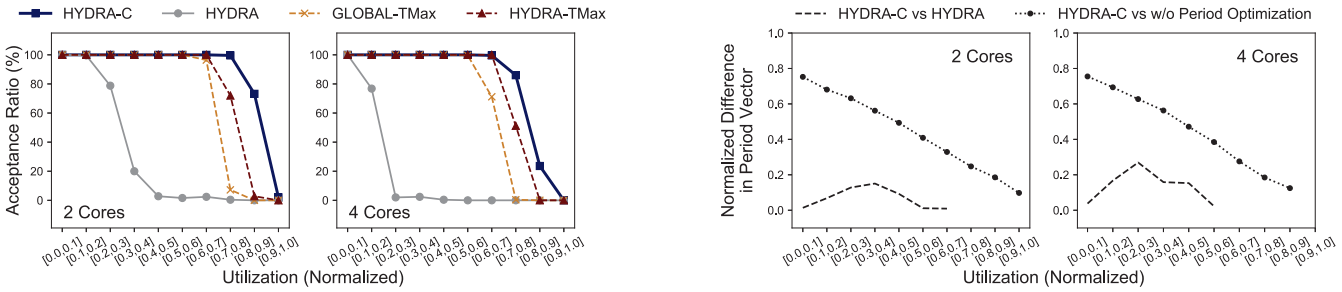
Experience and Evaluation: We observed the performance of HYDRA-C by *analyzing how quickly an intrusion can be detected*. We considered the following two realistic attacks[8]: *(i)* an ARM shellcode [24] that allows the attacker to modify the contents of the image data-store – this attack can be detected by Tripwire; *(ii)* a rootkit [25] that intercepts all the `read()` system calls – our custom security task can detect the presence of the malicious kernel module. In Fig. 5a we show the average time to detect both the intrusions (in terms of cycle counts, collected from 35 trials) for HYDRA-C and HYDRA schemes. From our experiments we found that, on average, our scheme can detect intrusions 19.05% faster compared to the HYDRA approach (Fig. 5a). Since our scheme allows security tasks to migrate across cores, it has shorter periods and that leads to faster detection times.

We next measured the overhead of our security integration approach in terms of number of context switches (CS). For each of the trials we observed the schedule for 45 seconds and counted the number of CS using the Linux `perf` tool [11]. In Fig. 5b we show the number of CS (y-axis) for HYDRA-C and HYDRA schemes (for 35 trials). As shown in the figure, our approach increases the number of CS (since we permit migration across cores) From our experiments we found that, on average, our scheme increases CS by 1.75 times. However, this increased CS overhead *does not impact the deadlines of RT tasks* (since the security tasks always execute with a priority lower than the RT tasks) and thus may be acceptable for many RT applications.

B. Experiment with Synthetic Tasksets

We also conducted experiments with randomly generated workloads for broader design-space exploration. We considered $M \in \{2, 4\}$ cores and each taskset instance contained $[3 \times M, 10 \times M]$ RT and $[2 \times M, 5 \times M]$ security tasks. We only considered schedulable RT tasksets. Each RT task had periods between $[10, 1000]$ ms and the maximum periods for security tasks were selected from $[1500, 3000]$ ms. We assumed that RT tasks were partitioned using a best-fit [15] strategy. The utilization of individual tasks were generated using Randfixedsum algorithm [26] and total utilization of the security tasks was at least 30% of the system utilization.

[8]Note: our focus here is on the integration of any given security mechanisms rather the detection of any particular class of intrusions. Hence we assumed that there were no zero-day attacks and the security tasks were able the detect the corresponding attacks correctly.



(a) (b)

Fig. 6. Impact on schedulability and security. *(a)* The acceptance ratio vs taskset utilizations for 2 and 4 core platforms: our scheme outperforms HYDRA and GLOBAL-TMax approaches for higher utilizations. *(b)* Difference in period vectors for our approach and reference schemes (*e.g.,* HYDRA, GLOBAL-TMax, HYDRA-TMax): the non-negative distance (y-axis in the figure) implies that HYDRA-C finds shorter periods than other schemes.

Impact on Schedulability and Security Trade-off: While in this work we consider a legacy system (where RT tasks are partitioned to cores), for comparison purposes we considered the following two schemes (in addition to HYDRA) that do not consider any period adaptation for security tasks.

• GLOBAL-TMax: In this scheme all the RT and security tasks are scheduled using a global fixed-priority multicore scheduling scheme[9] [1]. Since our focus here is on schedulability we set $T_s = T_s^{max},\ \forall \tau_s \in \Gamma_S$. This scheme allows us to observe the performance impacts of binding RT tasks to the cores (due to legacy compatibility).

• HYDRA-TMax: This is similar to the HYDRA approach introduced in §IV-A but instead of minimizing periods here we set $T_s = T_s^{max}, \forall \tau_s$. This allows us to observe the trade-offs between schedulability and security in a fully-partitioned system.

In Fig. 6a we compare the performance of HYDRA-C with the HYDRA, GLOBAL-TMax and HYDRA-TMax strategies in terms of *acceptance ratio* (y-axis) defined as the number of schedulable tasksets over the generated ones. As we can see from the figure, HYDRA-C outperforms HYDRA when the normalized utilization $\frac{\sum C_i/T_i}{M}$ (x-axis) increases. Our scheme allows security tasks to execute in parallel across cores and also allocate periods considering the schedulability constrains of all low priority tasks – this results in a smaller response times and can find more tasksets that satisfy the designer specified bound. In contrast, HYDRA uses a greedy approach that minimizes the periods of higher priority tasks first without considering the global state. Also HYDRA statically binds the security task to the core and hence suffers interference from the higher priority tasks assigned to that core – this leads to lower acceptance ratios. For higher utilizations HYDRA-C can find schedulable tasksets that can not be easily partitioned by using the HYDRA-TMax scheme. The acceptance ratio of our method and the HYDRA-TMax scheme is equal when utilization less than 0.7 since some lower priority security tasks experience less interference due to longer periods and specific core assignment. While we bind the RT tasks to cores (due to legacy compatibility), it does not affect the schedulability since RT tasks are already schedulable when partitioned and our analysis reduces the interference that RT tasks have on security ones. We also highlight that while our approach results in better schedulability, HYDRA-C/HYDRA-TMax and GLOBAL-TMax schemes are incomparable in general (*e.g.,* there exists tasksets that may be schedulable by task partitioning but not in global scheme and vice-versa) – we allow security tasks to migrate due to security requirements (*e.g.,* to achieve faster intrusion detection – as we explain in the next experiments, see Fig. 6b).

In the final set of experiments (Fig. 6b) we compare the achievable periods (in terms of Euclidean distance) for our approach and the other schemes. The x-axis in the Fig. 6b shows the normalized utilizations and the y-axis represents the average difference between the following period vectors $\mathbf{T^*} = [T_s^*]_{\forall \tau_s \in \Gamma_S}$: *(a)* HYDRA-C and HYDRA (dashed line); *(b)* HYDRA-C and other strategies (*e.g.,* GLOBAL-TMax and HYDRA-TMax) that do not consider period minimization (dotted marker). Higher distance values imply that the periods calculated by HYDRA-C are smaller (*i.e.,* leads to faster detection time) and our approach outperforms the other scheme. For low to medium utilizations HYDRA-C performs better when compared to HYDRA. In situations with higher utilizations, the lesser availability of slack time results in HYDRA-C and HYDRA performing in a similar manner. Our experiments show that HYDRA-C achieves better continuous monitoring when compared with both a fully-partitioned approach (HYDRA, HYDRA-TMax) and a global scheduling approach (GLOBAL-TMax) while providing the same or better schedulability.

V. Related Work

The closest line of research is HYDRA [16] where we proposed to statically partition security tasks to the cores. As we observed (see §IV), our prior work results in a poor acceptance ratio for larger utilizations and suffers interference from other high priority tasks leading to slower detection of intrusions (*i.e.,* less effective). While it was not done in the context of RT security, the scheduling approach presented in this paper can be considered as a special case of prior work [28] where each task can bind to any arbitrary number of

[9]We note that there exists recent work [27] that aims to reduce pessimism of multicore schedulability analysis by dividing task WCET into two virtual partitions and then calculating response times by enumerating all possible partitions. Given the workload of RT and security tasks, our interference calculations (§III-A) can be adopted to such a two-partitions method. However, from our experiments we found that this extra complexity (*e.g.,* enumerating all WCET partitions) does not improve the schedulability any further.

available cores. For a given period, that approach is pessimistic for our model in the sense that it over-approximates carry-in interference from the RT tasks and hence results in lower schedulability (*i.e.,* identical to the GLOBAL-TMax scheme in Fig. 6a). Researchers also studied schedulability for dynamic priority and FIFO systems [29] while our focus here is on fixed-priory RTS. There exists other work [30], [31] that considers the problem of period selection, however, they are designed for single core systems only.

Researchers proposed various mechanisms to provide security guarantees into RTS in several directions, *viz.,* integration of security mechanisms [18], [19], authenticating/encrypting communication channels [32], [33], side-channel defence techniques [34], [35] and hardware/software-based frameworks [14], [36]. Majority of those solutions are designed for single core platforms and often require system-level modifications and thus are not suitable for legacy systems. To our knowledge this is the first work that aims to achieve continuous monitoring for multicore-based legacy RT platforms.

VI. Conclusion

Threats to safety-critical RTS are growing and there is a need for developing layered defense mechanisms to secure such critical systems. In this paper we study mechanisms to integrate security monitoring for legacy multicore-based RTS. By using our framework, systems engineers can improve the security posture of RTS. This additional security guarantee also enhances safety – which is the main goal for such systems.

References

[1] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 43, no. 4, pp. 35:1–35:44, 2011.

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

[3] N. Falliere, L. O. Murchu, and E. Chien, "W32. stuxnet dossier," *White paper, Symantec Corp., Security Response*, vol. 5, p. 6, 2011.

[4] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, T. Kohno *et al.*, "Comprehensive experimental analyses of automotive attack surfaces," in *USENIX Sec. Symp.*, 2011.

[5] S. S. Clark and K. Fu, "Recent results in computer security for medical devices," in *MobiHealth*, 2011, pp. 111–118.

[6] "Tripwire," https://github.com/Tripwire/tripwire-open-source.

[7] "AIDE," http://aide.sourceforge.net/.

[8] "The Bro network security monitor," https://www.bro.org.

[9] M. Roesch, "Snort - lightweight intrusion detection for networks," in *USENIX Conf. on Sys. Admin.*, 1999, pp. 229–238.

[10] L. L. Woo, M. Zwolinski, and B. Halak, "Early detection of system-level anomalous behaviour using hardware performance counters," in *DATE*, 2018, pp. 485–490.

[11] V. M. Weaver, "Linux perf_event features and overhead," in *IEEE FastPath*, 2013.

[12] "OProfile," http://oprofile.sourceforge.net/.

[13] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM CSUR*, vol. 41, no. 3, p. 15, 2009.

[14] 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 *IEEE RTAS*, 2013, pp. 21–32.

[15] J. Chen, "Partitioned multiprocessor fixed-priority scheduling of sporadic real-time tasks," in *Euromicro ECRTS*, 2016, pp. 251–261.

[16] M. Hasan, S. Mohan, R. Pellizzoni, and R. B. Bobba, "A design-space exploration for allocating security tasks in multicore real-time systems," in *DATE*, 2018, pp. 225–230.

[17] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *JACM*, vol. 20, no. 1, pp. 46–61, 1973.

[18] M. Hasan, S. Mohan, R. B. Bobba, and R. Pellizzoni, "Exploring opportunistic execution for integrating security into legacy hard real-time systems," in *IEEE RTSS*, 2016, pp. 123–134.

[19] M. Hasan, S. Mohan, R. Pellizzoni, and R. B. Bobba, "Contego: An adaptive framework for integrating security tasks in real-time systems," in *Euromicro ECRTS*, 2017, pp. 23:1–23:22.

[20] N. Guan, M. Stigge, W. Yi, and G. Yu, "New response time bounds for fixed priority multiprocessor scheduling," in *IEEE RTSS*, 2009, pp. 387–397.

[21] "Implementation codes of the security integration framework," https://github.com/mnwrhsn/multicore-continuous-security-monitoring.

[22] "Raspberry Pi," https://tinyurl.com/rpi3modelb.

[23] L. Fu and R. Schwebel, "Real-time Linux wiki," https://rt.wiki.kernel.org/index.php/rt_preempt_howto, [Online].

[24] "Linux ARM shellcode," https://www.exploit-db.com/exploits/21253/.

[25] "Linux rootkit," https://github.com/crudbug/simple-rootkit.

[26] P. Emberson, R. Stafford, and R. I. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *WATERS*, 2010, pp. 6–11.

[27] Y. Sun and M. Di Natale, "Assessing the pessimism of current multicore global fixed-priority schedulability analysis," in *ACM SAC*, 2018, pp. 575–583.

[28] A. Gujarati, F. Cerqueira, and B. B. Brandenburg, "Schedulability analysis of the Linux push and pull scheduler with arbitrary processor affinities," in *Euromicro ECRTS*, 2013, pp. 69–79.

[29] A. Biondi and Y. Sun, "On the ineffectiveness of 1/m-based interference bounds in the analysis of global EDF and FIFO scheduling," *RTS*, vol. 54, no. 3, pp. 515–536, 2018.

[30] E. Bini and A. Cervin, "Delay-aware period assignment in control systems," in *IEEE RTSS*, 2008, pp. 291–300.

[31] A. Davare, Q. Zhu, M. Di Natale, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli, "Period optimization for hard real-time distributed automotive systems," in *ACM DAC*, 2007, pp. 278–283.

[32] V. Lesi, I. Jovanov, and M. Pajic, "Network scheduling for secure cyber-physical systems," in *IEEE RTSS*, 2017, pp. 45–55.

[33] M. Lin, L. Xu, L. T. Yang, X. Qin, N. Zheng, Z. Wu, and M. Qiu, "Static security optimization for real-time systems," *IEEE Trans. on Indust. Info.*, vol. 5, no. 1, pp. 22–37, 2009.

[34] S. Mohan, M.-K. Yoon, R. Pellizzoni, and R. B. Bobba, "Integrating security constraints into fixed priority real-time schedulers," *RTS Journal*, vol. 52, no. 5, pp. 644–674, 2016.

[35] M.-K. Yoon, S. Mohan, C.-Y. Chen, and L. Sha, "TaskShuffler: A schedule randomization protocol for obfuscation against timing inference attacks in real-time systems," in *IEEE RTAS*, 2016, pp. 1–12.

[36] 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 *ACM/IEEE ICCPS*, 2018, pp. 10–21.

Appendix

Proof of Lemma 1: Since RT tasks are partitioned and they have higher priorities than security tasks, the schedule of RT tasks executed on π_m does not depend on any other task in the system. Now consider any interval $[t, t+x)$ of length x. We show that we can obtain an interval $[t', t'+x)$ where all tasks are released at t', such that the workload of RT tasks on π_m is higher in $[t', t'+x)$ compared to $[t, t+x)$.

First step: let t' be the earliest time such that π_m continuously executes RT tasks in $[t', t)$; if such time does not exist, then let $t' = t$. By definition, π_m does not execute RT tasks at time $t'-1$. Also since RT tasks continuously execute in $[t', t)$, the workload of RT tasks in $[t', t'+x)$ cannot be smaller than the workload in $[t, t+x)$.

Second step: since π_m is idle at $t'-1$, no job of RT tasks on π_m released before t' can contribute to the workload in $[t', t)$. Hence, the workload can be maximized by anticipating the release of each RT task τ_r so that it corresponds with t'. This concludes the proof.