# A Server Model to Integrate Security Tasks into Fixed-Priority Real-Time Systems

Monowar Hasan*, Sibin Mohan*, Rakesh B. Bobba† and Rodolfo Pellizzoni‡
*Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, USA
†School of Electrical Engineering and Computer Science, Oregon State University, Corvallis, OR, USA
‡Dept. of Electrical and Computer Engineering, University of Waterloo, Ontario, Canada
Email: {*mhasan11, *sibin}@illinois.edu, †rakesh.bobba@oregonstate.edu, ‡rodolfo.pellizzoni@uwaterloo.ca

***Abstract*—Modern embedded real-time systems (RTS) are increasingly facing security threats than the past. In this paper, we develop a *unified* framework by using the concept of *server* and propose a *metric* to integrate security tasks into RTS that will allow system designers to improve the security posture without affecting temporal constraints of the existing real-time tasks. We demonstrate our framework using a proof-of-concept implementation on an ARM-based embedded platform and real-time Linux OS.**

## I. Introduction

Due to the use of component-based subsystems from different vendors as well as increased connectivity through unreliable media such as the Internet, modern real-time systems (RTS) are exposed to unknown security flaws. A successful attack on systems with real-time properties can have disastrous effects from loss of human life to damage to the equipment and/or the environment and thus necessitate the development of effective protection mechanisms that foil attacks on such systems. It is not straightforward to retrofit RTS with security mechanisms that were developed for more general purpose computing scenarios. This is because security mechanisms have to *a*) *co-exist* with the real-time tasks in the system and *b*) operate *without* impacting the *timing* and *safety* constraints of the control logic. This creates an apparent tension, especially for the *legacy* systems, *viz.,* the real-time control systems where the perturbation of existing constraints for constituent tasks (such as run-times, period, task execution order, *etc.*) is not allowed/feasible. Thus there is a trade-off between security requirements (*e.g.,* having enough cycles for effective detection) and the timing/safety requirements. Besides, security mechanisms have to be designed so that an adversary cannot easily evade them.

Our goal is to improve the RTS security posture by integrating security mechanisms *without* violating real-time constraints. The security mechanism could be any one of protection, detection or response mechanisms, depending on the system requirements. For example, a sensor measurement correlation task may be added for detecting sensor manipulation or a change detection task may be added to detect changes to, or intrusions into the system. The addition of such tasks may necessitate changing the schedule of real-time tasks. For instance, the authors in literature [1], [2] proposed to add a state cleansing task to deal with stealthy adversaries gleaning sensitive information through side channels. In contrast, we focus on situations where added security tasks are *not* allowed to impact the schedule of existing real-time tasks when integrating security mechanisms in the system.

While integrating security mechanisms into a practical system, performance criteria such as *frequency of monitoring* and *responsiveness* need to be considered. To provide best protection, security tasks may need to be executed quite often. If the interval between consecutive monitoring events is too large then an attacker may remain undetected and cause harm to the system between two invocations of the security task. In contrast, if the security tasks are executed very frequently, it may impact the schedulability of the real-time tasks. Besides, in some circumstances a security task may need to complete with less interference (*e.g.,* better responsiveness) from higher-priority tasks. As an example, let us consider the scenario where a security breach is suspected and a security task may be required to perform more fine-grained checking instead of waiting for its next execution slot. This may require some low-priority non-critical real-time tasks to finish *after* their deadlines within an *acceptable* bound. However, the scheduling policy needs to ensure that the system remains secure without violating real-time constraints for critical, high-priority control tasks.

In this paper we mainly focus on the *monitoring frequency* criterion. However, we do discuss extensibility of our proposed framework to incorporate better *responsiveness* against integrity violation. Specifically, we consider incorporating security mechanisms by implementing them as separate *sporadic tasks*. This brings up the challenge of determining the *right periods* (*viz.,* minimum inter-execution time) for the security tasks. For instance, some critical security routines may be required to execute more frequently than others. However, if the period is too short (*e.g.,* the security task repeats too often) then it will use too much of the processor time and eventually lower the overall system utilization. As a result, the security mechanism itself might prove to be a hindrance to the system and reduce the overall functionality or worse, safety. In contrast, if the period is too long, the security task may not always detect violations since attacks could be launched between two instances of the security task. Besides, if the security tasks execute with lower priority, they suffer more interference (*i.e.,* preemption from high-priority control tasks)

and the longer detection time (due to poor response time) will make the security mechanisms less effective.

The framework developed in this paper is based on our earlier work [3] where we proposed to execute security routines only when real-time tasks are *not* running. Our earlier approach is suitable for integrating security mechanisms into legacy systems since the execution order and schedulability of the real-time tasks are not affected. However, as illustrated in Section IV, the cost of not perturbing all of the real-time task schedule is the reduced speed of detecting intrusions. In contrast to our earlier work, in this paper we propose a *generic* framework that allows scheduling of security tasks along with real-time tasks that allows best possible periods for security routines while ensuring *all* the tasks in the system remain schedulable.

Specifically, in this paper we make following contributions:

- We introduce an extensible framework by using the concept of *server* [4] (Section III-B). The proposed method allows the security tasks to execute with *minimal* perturbation of the scheduling order of the real-time tasks while guaranteeing their timing constraints (Section III-C).
- We propose a *metric* to measure the security posture of the system in terms of frequency of periodic execution (Sections III-A-III-B).
- We evaluate the proposed approach for schedulability and security as well as a prototype implementation on an ARM-based development board and real-time Linux (Section IV).

## II. System and Security Model

### A. Real-Time Tasks

Let us consider a uni-processor system comprising $m$ fixed-priority sporadic real-time tasks $\Gamma_R = \{\tau_1, \tau_2, \cdots, \tau_m\}$. Each real-time task $\tau_i \in \Gamma_R$ is characterized by $(C_i, T_i, D_i)$, where $C_i$ is the worst-case execution time (WCET), $T_i$ is the minimum inter-arrival time (or period) between successive releases and $D_i$ is the relative deadline. We assume that priorities are distinct and assigned according to the Rate Monotonic (RM) [5] order. We also assume that tasks have implicit deadlines, *e.g.,* $D_i = T_i$ for $\forall i \in \Gamma_R$.

The processor utilization of $\tau_i$ is defined as $U_i = \frac{C_i}{T_i}$. Let $hp_R(\tau_i)$ and $lp_R(\tau_i)$ denote the set of tasks that have higher and lower priority than $\tau_i$, respectively. We assume that the real-time task-set $\Gamma_R$ is *schedulable* by a fixed-priority preemptive scheduling algorithm. Therefore, the worst-case response time $w_i$ is less than or equal to the deadline $D_i$ and the following inequality is satisfied for all tasks $\tau_i \in \Gamma_R$ [6]:

$$w_i = C_i + \sum_{\tau_h \in hp_R(\tau_i)} \left\lceil \frac{w_i^k}{T_h} \right\rceil C_h \leq D_i \tag{1}$$

where $\sum_{\tau_h \in hp_R(\tau_i)} \left\lceil \frac{w_i^k}{T_h} \right\rceil C_h$ is the worst-case interference to $\tau_i$ due to preemption by the tasks with higher priority than $\tau_i$ (*e.g.,* $hp_R(\tau_i)$). The recurrence starts with $w_i^0 = C_i$ and will have a solution if $w_i = w_i^{k+1} = w_i^k$ for some $k$.

### B. Attack Model

RTS face threats in various forms depending on the system and the goals of an adversary. For example, adversaries may insert, eavesdrop on or modify messages exchanged by system components. Besides, attackers may manipulate the processing of sensor inputs and actuator commands, could try to modify the control flow of the system as well as glean sensitive information through side channels [1], [2], [7]. Other than trying to aggressively crash the system, the intruder may utilize side-channels to monitor the system behavior and infer certain degree of system information (*e.g.,* hardware/-software architecture, user tasks and thermal profiles, *etc.*) that eventually leads to the attacker actively taking control, manipulating and/or crashing the system. This is different from the earlier work [1], [2] where adding security policies impact the schedulability. In this work, we focus on incorporating security mechanisms into legacy systems where added security tasks are not allowed to violate the temporal requirements and also have minimal impact on the schedule of existing real-time tasks.

### C. Security Tasks

Our goal is to ensure security of the system *with minimal perturbation* of the scheduling order and timing constraints (*e.g.,* Eq. (1)) of the real-time tasks. With a view to integrating security into the system, let us add additional $n$ fixed-priority sporadic security tasks denoted by the set $\Gamma_S = \{\tau_1, \tau_2, \cdots, \tau_n\}$. We assume that security tasks follow RM priority order. Each security task $\tau_i \in \Gamma_S$ is characterized by $(C_i, T_i^{des}, T_i^{max}, \omega_i)$, where $C_i$ is the WCET, $T_i^{des}$ is the most desired period between successive releases (hence $F_i^{des} = \frac{1}{T_i^{des}}$ is the desired execution frequency of a security routine) and $T_i^{max}$ is the maximum allowable period beyond which security checking by $\tau_i$ may not be effective. The parameter $\omega_i > 0$ is a designer provided weighting factor, that may reflect the criticality of the security task $\tau_i$. More critical security tasks would have larger $w_i$.

In order to provide a trade-off between security and system performance, we assume that security tasks are allowed to execute with a priority higher than *certain* low priority real-time tasks. For example, the entertainment subsystem of an avionics control system may have lower timing criticality compared to the navigation subsystem. Thus security tasks could be prioritized over such lower priority real-time tasks to maintain system security and in turn, its *safety*. Since the task priorities are distinct, there are $m$ priority-levels for real-time tasks (indexed from 0 to $m-1$ where level 0 is the highest priority). Among the $m$ priority-levels, we assume that security tasks can execute up to priority-level $l_S$ $(0 < l_S \leq m),\ l_S \in \mathbb{Z}$. Notice that $l_S = m$ implies that the security tasks execute with lowest priority and allowed to run *only* when other real-time tasks are not running.

One fundamental problem in integrating security routines is to determine *which* security tasks will be running *when*. Although any period $T_i$ within the range $T_i^{des} \leq T_i \leq T_i^{max}$ and priority-level $l \in [l_S, m]$ would be acceptable, the actual period $T_i$ and priority-level $l$ however, is not known a priori. Therefore our goal is to find the *suitable period as well as the priority-level for the security tasks* without violating the real-time constraints that achieve the best trade-off between schedulability and defense against security breaches.

## III. Period Adaptation

As mentioned earlier, the actual period as well as the priority-levels of the security tasks are unknown and we need to *adapt* the periods and priority-levels to be within acceptable ranges. We measure the security of the system by means of the *achievable periodic monitoring*. Let $T_i$ be the period of the security task $\tau_i \in \Gamma_S$ that needs to be determined. Since our goal is to minimize the perturbation between the achievable period $T_i$ and the desired period $T_i^{des}$, we define the following metric:

$$\eta_i = \frac{T_i^{des}}{T_i}, \tag{2}$$

that denotes the *tightness* of the frequency of periodic monitoring for the security task $\tau_i$. Thus $\eta = \sum_{\tau_i \in \Gamma_S} \omega_i \eta_i$ denotes the *cumulative tightness* of the achievable periodic monitoring. This monitoring frequency metric, for instance, provides one way to trade-off security with schedulability. As mentioned earlier, if the interval between consecutive monitoring events is too large, the adversary may remain undetected and harm the system between two invocations of the security task. On the other hand, a very frequent execution of security tasks may impact the schedulability of the real-time tasks. This metric $\eta$ will allow us to execute the security routines with a frequency closer to the desired one while respecting the temporal constraints of the other real-time control tasks.

### A. Problem Overview

A simple approach to integrating security tasks is to execute them at a lower priority than more critical, high-priority real-time tasks. Hence, the security routines will only be executing when other real-time tasks with priority-level higher than $l_S$ are not running. For instance, when both real-time and security tasks follow RM priority order, we can formulate a optimization problem with the following constraints that maximizes the cumulative tightness of the frequency of periodic monitoring:

$$(\mathbf{P1}) \qquad \max_{\mathbf{T}} \sum_{\tau_i \in \Gamma_S} \omega_i \frac{T_i^{des}}{T_i}$$

$$\text{Subject to: } \sum_{\tau_i \in \Gamma_S} \frac{C_i}{T_i} \leq (m+n)(2^{\frac{1}{m+n}} - 1) - \sum_{\tau_j \in \Gamma_R} \frac{C_j}{T_j} \tag{3a}$$

$$T_i \geq \max_{\tau_j \in \Gamma_{R_{hp(l_S)}}} T_j \quad \forall \tau_i \in \Gamma_S \tag{3b}$$

$$T_i^{des} \leq T_i \leq T_i^{max} \quad \forall \tau_i \in \Gamma_S \tag{3c}$$

where $\Gamma_{R_{hp(l_S)}}$ represents the set of real-time tasks that are higher priority than level $l_S$ and $\mathbf{T} = [T_1, T_2, \cdots, T_n]^\mathsf{T}$ is the optimization variable that needs to be determined. The constraint in Eq. (3a) ensures that the utilization of the security tasks are within the remaining RM utilization bound [5]. The RM priority order for real-time and security tasks are ensured by the constraints in Eq. (3b) and the constraints in Eq. (3c) fulfill the restrictions on periodic monitoring.

One of the limitations of the above approach is that the overall system utilization is limited by RM bound and also the security task's periods need to satisfy the constraint in Eq. (3b) to follow RM priority order. Besides, rather than only focusing on optimizing the periods of the security tasks, we aim to design a *unified* framework that can achieve other security aspects (*e.g.,* responsiveness). Thus we follow an alternative approach similar to the one presented in our earlier research [3]. Specifically, we had proposed to use a *server*[1] in order to execute the security tasks. By utilizing this server-based approach, for instance, if better responsiveness is desired from security mechanisms, we could increase the priority of the server and allow the server to execute until the security task finishes its desired checking. This server abstraction allows us to provide better isolation between real-time and security tasks and provides us to integrate additional security properties. For instance, security tasks can be operated in different *modes* (*viz.,* passive monitoring or exhaustive checking) and can switch from one mode to other when security breaches are suspected[2]. In what follows we briefly discuss our proposed server-based model for integrating security tasks in the system.

### B. The Security Server

The security server $\mathcal{S}(Q, P)$ is characterized by the *capacity* $Q$ and *replenishment period* $P$ and schedules the security tasks according to preemptive fixed-priority order that we consider RM priority order in this work. The server can execute with any allowable priority-level[3] within the range $[l_S, m]$.

*1) Reformulation of the Period Adaptation Problem using Server:* When security tasks are executing within the server, we need to modify the RM utilization bound in Eq. (3a) and (3b) considering the server parameters ($Q$ and $P$). Let us denote $UB_{\mathcal{S}(Q,P),\Gamma_S}$ as the utilization bound for the set of security task $\Gamma_S$ executing within the server. With the inclusion of the server, we can modify the constraints in Eqs. (3a) and (3b) with the utilization bound $UB_{\mathcal{S}(Q,P),\Gamma_S}$ using the concept similar to that discussed in literature [8]. Specifically, when the smallest period of the security task is greater than or equal to $3P - 2Q$, the upper bound of the utilization factor for the security tasks is given by $UB_{\mathcal{S}(Q,P),\Gamma_S} = n \left[ \left( \frac{3 - \frac{Q}{P}}{3 - 2\frac{Q}{P}} \right)^{\frac{1}{n}} - 1 \right]$

[1]The security server is an abstraction that provides execution time to the security tasks, according to a predefined scheduling policy. For an overview of using the server for security tasks we refer the readers to literature [3].

[2]This aspect is discussed further in Section V.

[3]Calculation of the server priority-level is described in Section III-C.

where $n$ is number of security tasks in the set $\Gamma_S$. Therefore, the constraints in Eqs. (3a) and (3b) can be rewritten as follows

$$\sum_{\tau_i \in \Gamma_S} \frac{C_i}{T_i} \leq n \left[ \left( \frac{3-\frac{Q}{P}}{3-2\frac{Q}{P}} \right)^{\frac{1}{n}} - 1 \right] \tag{4a}$$

$$T_i \geq 3P - 2Q \quad \forall \tau_i \in \Gamma_S. \tag{4b}$$

Therefore, the period optimization problem with presence of the server can be represented as follows

$$(\mathbf{P2}) \ \max_{\mathbf{T}} \sum_{\tau_i \in \Gamma_S} \omega_i \frac{T_i^{des}}{T_i}, \quad \text{Subject to: (4a), (4b), (3c).}$$

*2) Selection of the Server Parameters:* The period adaptation problem given by $\mathbf{P2}$ is derived based on a given server parameter $\mathcal{S}(Q,P)$ *e.g.,* the utilization bound $UB_{\mathcal{S}(Q,P),\Gamma_S}$. However, one fundamental problem is to find a suitable pair of server capacity $Q$ and replenishment period $P$ that respects the real-time constraints of the tasks in the system. Since the security tasks are executing within the server, we need to ensure the following: *i*) the server is schedulable (*e.g.,* server's capacity and interference from higher priority real-time tasks is less than the replenishment period); *ii*) the security tasks are schedulable (*e.g.,* minimum supply by the server to the security tasks is greater than the worst-case workload generated by the security tasks); and *iii*) the real-time tasks with lower priority than the server are schedulable (*e.g.,* interference from the server and other higher priority real-time tasks do not violate the deadline). Note that, the schedulability of the higher priority real-time tasks (*e.g.,* $\forall \tau_j \in hp_R(\tau_S)$) is already ensured by definition (*viz.,* Eq. (1)). Based on these requirements, we illustrate how to determine the server parameters by formulating it as a *constraint optimization problem*.

The security server is referred to as schedulable if worst-case response time of the server does not exceed its replenishment period [4]. Thus following an approach similar to earlier work [3], the *server schdulability constraint* can be represented as follows:

$$Q + \Delta_{\mathcal{S}} \leq P \tag{6}$$

where $\Delta_{\mathcal{S}} = \sum_{\tau_h \in hp_R(\tau_S)} \left( \frac{P}{T_h} + 1 \right) C_h$ is the worst-case interference experienced by the server when preempted by the higher priority real-time tasks. In the above equation, the set of real-time tasks with higher priority than the server (*i.e.,* $hp_R(\tau_S)$) is constant for a given priority-level.

Let us denote $hp_S(\tau_i)$ the set of security task that are higher priority than $\tau_i \in \Gamma_S$. To ensure schedulability of the security tasks, we can derive the *minimum supply* of the server delivered to the security tasks by using the periodic resource model introduced in literature [9]. In particular, the constraints on the server supply to ensure *schedulability of the security tasks* [3] can be expressed as:

$$\frac{Q}{P} \left[ T_i - (P - Q) - \Delta_{\mathcal{S}} \right] \geq I_i \quad \forall \tau_i \in \Gamma_S \tag{7}$$

where $I_i = C_i + \sum_{\tau_h \in hp_S(\tau_i)} \left\lceil \frac{T_i}{T_h} \right\rceil C_h$ is the worst-case workload generated by the security task $\tau_i$ and $hp_S(\tau_i)$ during the time interval of $T_i^{des}$. This workload is a constant for a given input.

As mentioned earlier, the security server can be executed on any priority within the range $[l_S, m]$. Thus we need to ensure the schedulability of the real-time tasks that are executing with a priority lower than the server. For a given priority-level, let us denote $lp(\tau_{\mathcal{S}})$ as the set of real-time tasks that are with lower priority than the server. Hence we define the following constraints to ensure the *schedulability of the low-priority real-time tasks*

$$C_j + \sum_{\tau_h \in hp_R(\tau_j)} \left\lceil \frac{D_j}{T_h} \right\rceil C_h + \left( \frac{D_j}{P} + 1 \right) Q \leq D_j \quad \forall \tau_j \in lp(\tau_{\mathcal{S}}) \tag{8}$$

where $\sum_{\tau_h \in hp_R(\tau_j)} \left\lceil \frac{D_j}{T_h} \right\rceil C_h$ is the interference from other real-time tasks to $\tau_j$ and $\left( \frac{D_j}{P} + 1 \right) Q$ is the worst-case interference caused to $\tau_j$ by the server. Notice that, for a given priority-level the set of tasks $lp(\tau_{\mathcal{S}})$ is predefined. Thus the only variables for the constraints in Eq. (8) are the server capacity $Q$ and replenishment period $P$.

Since we need to ensure maximal processor utilization for the security tasks without violating the real-time constraints of the system, we define the following objective function

$$\max_{Q,P} \frac{Q}{P}. \tag{9}$$

With the objective function in Eq. (9) and the constraints in Eqs. (6)-(8), the server parameter selection problem can be formulated as follows

$$(\mathbf{P3}) \qquad \max_{Q,P} \frac{Q}{P}, \quad \text{Subject to: (6), (7), (8)}$$

where server parameters, $Q$ and $P$ are the optimization variables.

**Remark 1.** *The formulation of the period adaptation problem (*$\mathbf{P3}$ *and* $\mathbf{P2}$*) is a non-linear constraint optimization problem and non-trivial to solve in its current form. However, this problem can be transformed into a Geometric Programming (GP) problem. In addition, it is also possible to reformulate the non-convex GP representation into equivalent convex form that can be solvable using known algorithms such as Interior Point method. For details of this reformulation, we refer the readers to our earlier work [3].*

It is worth mentioning that the concept of *opportunistic execution* of the security tasks (*viz.,* security tasks are allowed to execute only when other real-time control tasks are *not* running) presented in earlier work [3] can be considered as a special case of the proposed method. For instance, if it is required to integrate security in the system without perturbing execution order of the *any* of the real-time tasks (which is especially required for many legacy systems), one can set

$l_S = m$[4]. In contrast to our earlier work we therefore provide a *generic* approach to integrate security tasks in the RTS.

### C. Algorithm Development

We develop a simple iterative scheme to obtain the security task's period and priority-level. The overall algorithm, as presented in **Algorithm 1** works as follows. We initialize the period vector with desired periods (*e.g.,* $\mathbf{T}(l')_{\forall l' \in [l_S,m]} = [T_i^{des}]^{\mathsf{T}}_{\forall \tau_i \in \Gamma_S}$) and solve the optimization problems (*e.g.,* $\mathbf{P3}$ and $\mathbf{P2}$) to obtain server parameters and security task's periods for each of the allowable priority-levels (Line 2-13). If there exists a solution for any priority level $l' \in [l_S, m]$, we store the solutions in a candidate solution list (Line 9).

If the candidate solution list is non-empty, the algorithm then finds the best priority-level (say $l^*$) from the candidate list that maximizes the cumulative tightness of periodic monitoring and returns the solution, *viz.,* the tuple $\{l^*, \mathbf{T}(l^*), Q(l^*), P(l^*)\}$, *e.g.,* server priority-level, periods of the security tasks and the server parameters (Line 15-18). In the event there are no candidate solutions found (*e.g.,* the flag Schedulable = **false**), the task-set is reported as unschedulable (Line 20) since it is not possible to integrate security tasks with desired requirements.

**Algorithm 1** Feasibility Checking and Parameter Selection

**Input:** Set of real-time and security tasks, $\Gamma_R$ and $\Gamma_S$, respectively, and allowable priority ranges $[l_S, m]$
**Output:** The tuple $\{l^*, \mathbf{T}(l^*), Q(l^*), P(l^*)\}$, *e.g.,* server priority-level, periods of the security tasks and the server parameters if the task-set is schedulable; Unschedulable otherwise

1: Schedulable := **false**
2: Initialize security task's period $\mathbf{T}(l')_{\forall l' \in [l_S,m]} := [T_i^{des}]^{\mathsf{T}}_{\forall \tau_i \in \Gamma_S}$
3: **for each** priority level $l' \in [l_S, m]$ **do**
4: Solve $\mathbf{P3}$ to obtain server parameters
5: **if** SoulutionFound **then**
6: Solve $\mathbf{P2}$ to obtain security periods
7: **if** SoulutionFound **then**
8: */* store the parameters for priority level $l'$ where $Q^*$, $P^*$ and $\mathbf{T}^*$ are the solutions obtained by $\mathbf{P3}$ and $\mathbf{P2}$, respectively */*
9: $Q(l') := Q^*, P(l') := P^*, \mathbf{T}(l') := \mathbf{T}^*$
10: Schedulable := **true**
11: **end if**
12: **end if**
13: **end for**
14: */* Obtain the parameters that provide best security metric */*
15: **if** Schedulable **then**
16: Find the priority-level $l^*$ from $\mathbf{T}(l')_{\forall l' \in [l_S,m] |}$ tasks at $l'$ is schedulable that gives the maximum cumulative tightness $\eta = \sum_{\tau_i \in \Gamma_S} \eta_i$
17: */* return the parameters */*
18: **return** $l^*$, $\mathbf{T}(l^*)$, $Q(l^*)$, $P(l^*)$
19: **else**
20: **return** Unschedulable */* not possible to integrate security tasks */*
21: **end if**

## IV. Evaluation

We evaluate the proposed scheme with randomly generated synthetic workloads(Section IV-A) as well as a proof-of-concept implementation on an ARM-based embedded development board and real-time Linux OS (Section IV-B).

[4]When $l_S = m$, the constraints in Eq. (8) are no longer necessary.

### A. Experiment with Synthetic Task-sets

*1) Experimental Setup:* In order to generate task-sets with an even distribution of tasks, the real-time and security task-sets are grouped by base-utilization from $[0.01 + 0.1 \cdot i, 0.1 + 0.1 \cdot i]$ where $i \in \mathbb{Z} \wedge 0 \leq i \leq 9$. Each utilization group contains 500 task-sets. In other words, total 5000 task-set tested for each of the experiments. The utilization of the real-time and security tasks are generated by the UUniFast [10] algorithm and we use GGPLAB [11] to solve the optimization problems (*i.e.,* Line 3 and 5 in **Algorithm 1**).

We use the parameters similar to that used in earlier research [1], [3]. In particular, each task-set instance contains $[3, 10]$ real-time and $[2, 5]$ security tasks. Each real-time task $\tau_j \in \Gamma_R$ has a period $T_j \in [10ms, 100ms]$ and we assume $l_S = \lceil 0.3m \rceil$. The desired periods for the security tasks are selected from $[1000ms, 3000ms]$ and the maximum allowable period is assumed to be $T_i^{max} = 10T_i^{des}$, $\forall \tau_i \in \Gamma_S$. We consider $\omega_i = 1$, $\forall \tau_i \in \Gamma_S$ and the total utilization of the security tasks are assumed to be no more than 30% of the real-time tasks.

*2) Results:* We compare the performance of the proposed scheme with the approach described in our earlier work [3] where the security routines are allowed to run only when the real-time tasks are *not* executing (referred to as *opportunistic execution*).

*a) Improved Performance:* In Fig. 1 we measure the difference in tightness of periodic monitoring (*e.g.,* $\eta$) obtained by proposed scheme and the opportunistic execution approach. The positive values in the y-axis of Fig. 1 implies that the proposed scheme performs better than the scheme presented in our previous work [3]. The figure shows that our approach can achieve better cumulative tightness, and performs comparatively better in low to medium utilization. For higher utilization difference is close to zero, however, this does not attribute that the proposed scheme performs inferior than the other approach. Instead, as utilization increases, schedulability inefficiencies (*e.g.,* less number of schedulable task-set in base-utilization groups) are making both schemes look more similar.

*b) Effectiveness of Security:* The parameter $\eta$ is given by the total number of the security tasks and provides insights on cumulative measure of security. However, in this experiment we like to measure the effectiveness of security of the system by observing whether *each* of the security tasks can achieve the execution frequency closer to the desired one. Hence we use the following metric: $\xi = \frac{\|\mathbf{T}^* - \mathbf{T^{des}}\|_2}{\|\mathbf{T^{max}} - \mathbf{T^{des}}\|_2}$ where $\mathbf{T}^*$ is the solution obtained from **Algorithm 1**, $\mathbf{T^{des}} = [T_i^{des}]^{\mathsf{T}}_{\forall \tau_i \in \Gamma_S}$ and $\mathbf{T^{max}} = [T_i^{max}]^{\mathsf{T}}_{\forall \tau_i \in \Gamma_S}$ are the desired and maximum period vector (refer to Appendix IV-A1), respectively, and $\|\cdot\|_2$ denotes the Euclidean norm. The closer the value of $\xi$ to 0, the nearer the period of each of the security task is to the desired period. As the total utilization increases, the feasible set of of the period adaptation problem that respects all constraints in $\mathbf{P3}$ and $\mathbf{P2}$ becomes more restrictive. As a result, we see the degradation in effectiveness (in terms of $\xi$) for the task-sets with higher utilization. However, from our experiments

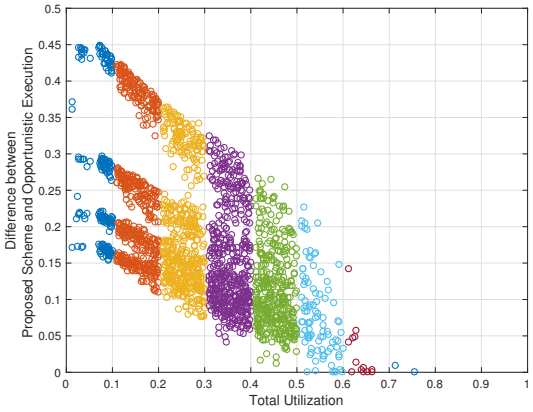


Fig. 1. Proposed scheme vs opportunistic execution: difference in cumulative tightness of achievable periodic monitoring, $\eta$. Non-zero difference indicates that the proposed schemes performs better than previous work [3]. Each of the data points represents schedulable task-sets.

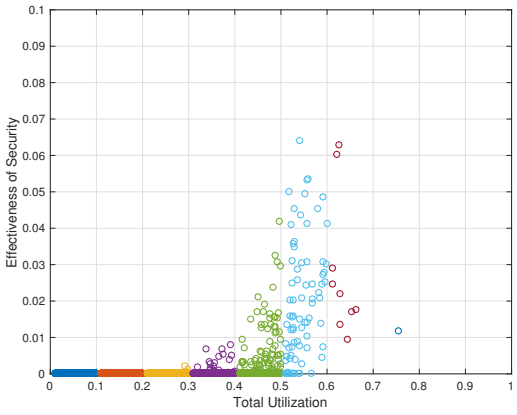


Fig. 2. The effectiveness of security vs total utilization of the system. The closer the y-axis values to 0, the nearer the period of each of the security task is to the desired period.

we find that the proposed scheme can achieve the periods that is *within* 7% *of the desired periods*.

### *B. Experiment with a Security Application in an Embedded Platform*

To observe the performance of the proposed scheme in a practical setup, we develop a proof of concept implementation on an embedded platform. For this, we use a BeagleBone Black (BBB) development board[5] (1 GHz ARM Cortex-A8 single-core processor, 512 MB RAM) as our experimental platform. We select an embedded Debian GNU/Linux console image with Xenomai[6] 2.6.3 real-time patch (kernel version 3.8.13-r72) as the OS. Our prototype implementation is developed in C and uses a fixed-priority scheduler powered by the Xenomai real-time patch. Sporadic (real-time and security) tasks in the system are defined by Xenomai `rt_task_create()` function and suspended after the completion of corresponding instances using the `rt_task_wait_period()` function.

*1) Real-time Tasks:* For a real-time application, we consider a UAV control system [12]. It includes following real-time tasks[7]: *Guidance*, *Slow navigation*, *Fast navigation*, *Controller*, *Missile control* and *Reconnaissance*. Since the actual control code of these tasks are not publicly available, we demonstrate tasks behavior by developing a custom busy-wait loop using `rt_timer_spin()` function.

*2) Security Tasks:* To integrate security in the aforesaid control system, we include additional security tasks. For the security application, we consider a lightweight open source intrusion detection mechanism, Tripwire[8], that detects integrity violations in the system. Tripwire stores the clean system state during initialization and uses it later to detect intrusions by comparing the current system state against the stored clean values. As Table II shows (refer to Appendix), the default configuration of Tripwire contains several tasks, *viz., protecting its own binary files*, *protecting system binary and library files*, *ensuring kernel and process integrity*, *etc.* By using the WCET values of the Tripwire application tasks obtained from the experiments (see Table II), we set the desired and maximum allowable periods of the security tasks such that total utilization of the security tasks are not exceeding 0.25.

*3) Experience and Evaluation:* We assume that attacker can hijack[9] one of the low-priority real-time tasks (refer to as victim task) and is able to insert malicious codes that can execute with a privilege similar to that of legitimate tasks (*viz.,* `root`). Malware (such as LRK, tOrn, Adore, *etc.*) in general-purpose Linux environment causes damage to the system by modifying or overwriting the system binary [13, Ch. 5]. Thus we follow a similar approach to illustrate malicious behavior, *viz.,* we override the victim task's code and launch the attack by modifying the contents in the file-system binary.

We obtain the periods of the security tasks by solving the period adaptation problem (**Algorithm 1**) and set it as the period of Tripwire tasks (using Xenomai `rt_task_set_periodic()` function). For each of the experiments, we start with a clean (*e.g.,* uncompromised) system state, launch an attack at any random point of the program execution and log the time required by Tripwire to detect the attack. We control the experiment environment so that the results are not affected by false positive/negative errors. We measure the detection time using ARM cycle counter registers (*e.g.,* CCNT) that gives us nanosecond-level precision. For accuracy of the detection time measurement, we disable all the frequency scaling feature in the kernel (using `cpufrequtils` utility) and allow the

[5]https://beagleboard.org/black.

[6]https://xenomai.org/.

[7]The task parameters of the UAV control system is summarized in Table I (see Appendix).

[8]http://www.tripwire.com/.

[9]One way to take over a task could be using the approach similar to that presented in literature [7] that exploits the deterministic behavior of the real-time scheduling.

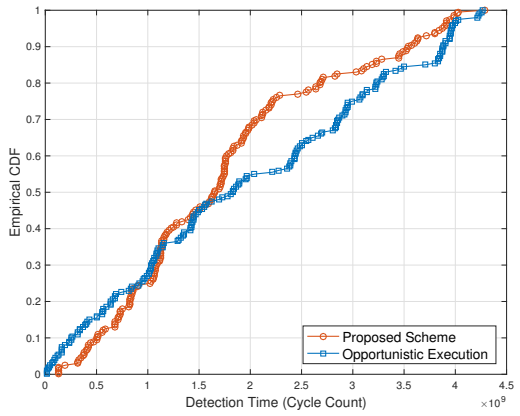


Fig. 3. The empirical distribution of time to detect an intrusion by proposed scheme and opportunistic execution. We use ARM cycle counter registers to measure the detection time. Total 100 individual experiment instances are examined to obtain the timing traces.

platform to execute with a constant frequency (*e.g.,* 1 GHz, the maximum frequency of BBB).

We compare the performance of proposed approach with earlier work [3] by measuring the time to detect the attack by both of these schemes and plot the empirical cumulative distribution function (CDF) of the detection time in Fig. 3. The x-axis in Fig. 3 represents the detection time (in cycle count) and y-axis represents the the probability that the attack being detected by that time. The empirical CDF is defined as $\widehat{F}_{\alpha}(\jmath) = \frac{1}{\alpha}\sum_{i=1}^{\alpha}\mathbb{I}_{[\zeta_i \leq \jmath]}$, where $\alpha$ is the total number of experimental observations, $\zeta_i$ is time to detect the attack in at the $i$-th experimental observation, and $\jmath$ represents the $x$-axis values (*viz.,* detection time in cycle count) in Fig. 3. The indicator function $\mathbb{I}_{[\cdot]}$ outputs 1 if the condition $[\cdot]$ is satisfied and 0 otherwise. As we can see from the figure, the proposed scheme provides better detection time (e.g., lesser cycle count required to detect the intrusion). Although with small probability, we find that there exist some experiment instances (specially the cases when detection time is less than $10^9$ cycle counts) the reference scheme outperforms the proposed approach. This is because the time to detect an attack depends on the actual point of intrusion and corresponding scheduling instance of the security tasks. However, from our experiments we find that *on average* the proposed scheme detects the attack $10.49\%$ faster (with $0.174$ standard deviation) than the reference scheme. Since the approach in literature [3] allows the security tasks to run only when other real-time tasks are not running, that causes more interference (*e.g.,* higher response time) and thus leads to poorer detection rate in general for most of the experiments.

## V. Discussion

Despite the fact that we are stepping towards an approach to integrate security policies in RTS, this work can be extended in several directions to develop a comprehensive security-aware framework. Our proposed approach allow security routines to be executed over some of the lower priority real-time tasks. However this may not be possible for many legacy systems especially where altering the execution order of the real-time tasks is not straightforward due to control requirements. In such cases, a useful solution could be allow the security routines to executed in different *modes* (*viz.,* passive monitoring by opportunistic execution as well as exhaustive checking with higher priority and allowing acceptable deadline tolerance on real-time tasks).

By using this approach, for instance, security routines can execute *opportunistically* when the system is deemed to be clean (*i.e.,* not compromised). However if any anomaly or unusual behavior is suspected, the security policy may switch to *fine-grained checking mode* and execute with *higher priority instead of waiting for its next sporadic slot*. The security routines may go back to normal (*e.g.,* opportunistic) mode if: *i*) no anomalous activity is found; or *ii*) the intrusion is detected and malicious entities are removed (or an alarm may be triggered if human intervention is required). However, the scheduling policy needs to ensure that the system remains *functional* (possibly with some tolerance on deadline for the low priority real-time tasks) *even with these mode changes*.

In addition, depending on the actual implementations of the security routines, the scheduling framework may need to follow certain *precedence constraints*. For example, in order to ensure that the security application itself has not been compromised, the security application's own binary may need to be examined first before checking the system binary files. These aspects will be explored in our future work.

## VI. Related Work

In our earlier work [3], we proposed to execute security tasks with the lowest priority relative to real-time tasks. Although this approach may be useful for existing systems since the schedulability of the real-time tasks remain unaffected, this leads to longer response time for the security tasks and thus may increase the detection time of an attack.

Although not in the context of security in RTS, there exists other work [14] in which the authors statically assign the periods for multiple independent control tasks considering control delay as a cost metric and estimate the delay using an approximate response time analysis. In contrast, our goal is to ensure security without violating timing constraints of the real-time tasks. Hence, instead of minimizing response time, our goal is to assign best possible periods and priority-levels, so that we can minimize the perturbation between achievable period and desired period for all the security tasks.

A state cleanup mechanism is introduced in literature [1] and further generalized [2] where the authors modify the fixed priority scheduling algorithm to mitigate information leakage through shared resources. However, this leakage prevention comes at a cost of reduced schedulability. In comparison, we propose to ensure security policies *without* violating temporal constrains and schedulability of the real-time control tasks.

With a view to hardening security mechanisms by minimizing predictability of deterministic RTS scheduler, researchers proposed a schedule obfuscation method [15] aimed at randomizing the task schedule while providing the necessary real-time guarantees. Different from our scheme that works at the scheduler-level, recent work [16], [17] on dual-core based hardware/software architectural frameworks aim to protect RTS against security vulnerabilities. It is not inconceivable that the architectural frameworks [16], [17] as well as the randomization protocols [15] can the employed on top of proposed scheme to harden security mechanisms in future RTS.

## VII. Conclusion

Threats to RTS are growing, both in number as well as sophistication, as demonstrated by recent attacks on UAVs [18], automobiles [19] as well as industrial control systems [20]. In this paper we are stepping towards developing a *generalized* framework to integrating security mechanisms and provide a glimpse of *security design metrics* for RTS. Designers of RTS are now able to improve their security posture, this will also improve overall *safety* – which is essentially the main goal for such systems.

## References


[1] S. Mohan, M.-K. Yoon, R. Pellizzoni, and R. B. Bobba, "Real-time systems security through scheduler constraints," in *IEEE ECRTS*, 2014, pp. 129–140.

[2] ——, "Integrating security constraints into fixed priority real-time schedulers," *RTS Journal*, vol. 52, no. 5, pp. 644–674, 2016.

[3] M. Hasan, S. Mohan, R. B. Bobba, and R. Pellizzoni, "Exploring opportunistic execution for integrating security into legacy hard real-time systems," *arXiv preprint*, 2016, http://arxiv.org/abs/1608.07872 [Online].

[4] R. Davis and A. Burns, "An investigation into server parameter selection for hierarchical fixed priority pre-emptive systems," in *IEEE RTNS*, 2008.

[5] 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.

[6] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *SE Journal*, vol. 8, no. 5, pp. 284–292, 1993.

[7] C.-Y. Chen, R. B. Bobba, and S. Mohan, "Schedule-based side-channel attack in fixed-priority real-time systems," University of Illinois, http://hdl.handle.net/2142/88344, Tech. Rep., 2015, [Online].

[8] S. Saewong, R. R. Rajkumar, J. P. Lehoczky, and M. H. Klein, "Analysis of hierarchical fixed-priority scheduling," in *IEEE ECRTS*, 2002, pp. 173–181.

[9] M.-K. Yoon, J.-E. Kim, R. Bradford, and L. Sha, "Holistic design parameter optimization of multiple periodic resources in hierarchical scheduling," in *DATE*, 2013, pp. 1313–1318.

[10] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *RTS Journal*, vol. 30, no. 1-2, pp. 129–154, 2005.

[11] A. Mutapcic, K. Koh, S. Kim, L. Vandenberghe, and S. Boyd, "GGPLAB: a simple Matlab toolbox for geometric programming," 2006. [Online]. Available: https://stanford.edu/~boyd/ggplab/

[12] T. Atdelzater, E. M. Atkins, and K. G. Shin, "Qos negotiation in real-time systems and its application to automated flight control," *IEEE TC*, vol. 49, no. 11, pp. 1170–1183, 2000.

[13] *Ethical Hacking and Countermeasures: Secure Network Operating Systems and Infrastructures*, 2nd ed. EC-Council, 2017.

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

[15] 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.

[16] D. Lo, M. Ismail, T. Chen, and G. E. Suh, "Slack-aware opportunistic monitoring for real-time systems," in *IEEE RTAS*, 2014, pp. 203–214.

[17] 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.

[18] D. P. Shepard, J. A. Bhatti, T. E. Humphreys, and A. A. Fansler, "Evaluation of smart grid and civilian UAV vulnerability to GPS spoofing attacks," in *Proc. of the ION GNSS Meeting*, vol. 3, 2012.

[19] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham *et al.*, "Experimental security analysis of a modern automobile," in *IEEE S&P*, 2010, pp. 447–462.

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


## Appendix
## Task Parameters used in the Experiments

TABLE I
Real-time task parameters for the UAV control system [12]

| Task | Function | Period (ms) | WCET (ms) |
|---|---|---|---|
| Guidance | Select the reference trajectory (*i.e.,* altitude and heading) | 1000 | 100 |
| Controller | Execute closed-loop control functions (*e.g.,* actuator commands) | 5000 | 80 |
| Slow Navigation | Read sensor values according to the required update frequency | 1000 | 100 |
| Fast Navigation | | 200 | 60 |
| Missile Control | Read radar and fire missiles | 10000 | 500 |
| Reconnaissance | Read radar/camera data, collect sensitive information and send data to the base control station | 10000 | 200 |

TABLE II
Security tasks used in the experiments*

| Task | Function | WCET (ms) |
|---|---|---|
| Check own binary of the security routine | Scan (*viz.,* compare hash value) files in the following locations: `/usr/sbin/siggen`, `/usr/sbin/tripwire`, `/usr/sbin/twadmin`, `/usr/sbin/twprint` | 3640 |
| Check critical executables | Scan file-system binary (`/bin`, `/sbin`) | 4031 |
| Check critical libraries | Scan file-system library (`/lib`) | 2670 |
| Check device and kernel | Scan peripherals and kernel information in the `/dev` and `/proc` directory | 4004 |
| Check configuration files | Scan changes in the configuration files (`/etc`) | 3845 |

*We measure the WCET of the tasks by using the ARM cycle counter registers (CCNT). Total 2500 execution traces were examined to obtain these timing traces.