# A Design-Space Exploration for Allocating Security Tasks in Multicore Real-Time Systems

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

***Abstract*—The increased capabilities of modern real-time systems (RTS) expose them to various security threats. Recently, frameworks that integrate security tasks without perturbing the real-time tasks have been proposed, but they only target single core systems. However, modern RTS are migrating towards multicore platforms. This makes the problem of integrating security mechanisms more complex, as designers now have multiple choices for where to allocate the security tasks. In this paper we propose *HYDRA*, a design space exploration algorithm that finds an allocation of security tasks for multicore RTS using the concept of *opportunistic execution*. HYDRA allows security tasks to operate with existing real-time tasks *without* perturbing system parameters or normal execution patterns, while still meeting the desired monitoring frequency for intrusion detection. Our evaluation uses a representative real-time control system (along with synthetic task sets for a broader exploration) to illustrate the efficacy of HYDRA.**

## I. Introduction

Real-time systems (RTS) rely on a variety of inputs for their correct operation and have to meet stringent *safety* and *timing* requirements. The drive towards remote monitoring and control, increased connectivity through unreliable media such as the Internet and the use of component-based subsystems from different vendors are exposing modern RTS a multitude of threats. A successful attack on systems with real-time properties can have disastrous effects – from loss of human life to damage to the environment and/or hard to replace equipment. A number of high-profile attacks on real systems, (*e.g.,* denial-of-service (DoS) attacks from Internet-of-Things devices [1], Stuxnet [2], BlackEnergy [3], *etc.*) have shown that the threat is real. Hence it is essential to retrofit existing critical RTS with detection, survival and recovery mechanisms.

As the use of multicore platforms in safety-critical real-time systems is increasingly becoming common, the focus of this work is on integrating or retrofitting security mechanisms into *multicore RTS*. It is not straightforward to retrofit RTS with security mechanisms that were developed for more general purpose computing scenarios since, 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. Further, it *may not be feasible to adjust the parameters (such as run-times, period, and execution order, etc.) of real-time tasks to accommodate security tasks.* This creates an apparent tension between security and real-time requirements. Unlike single core systems, integrating security into multicore platforms is more challenging since designers have multiple choices across cores to retrofit security mechanisms. For instance, is it better to *dedicate a core* to all the security tasks or is it better to *spread them out* (in conjunction with the real-time tasks) and if so, to *which* cores?

The material in this paper is based upon work supported in part by the National Science Foundation (NSF) grant numbers NSF CNS 14-23334 and NSF SaTC 1718952, and by the Natural Sciences and Engineering Research Council of Canada (NSERC).

Our goal is to improve the security posture of multicore RTS by integrating security mechanisms *without* violating real-time constraints. Security mechanism could include protection, detection or response mechanisms, depending on the system requirements – for instance, a sensor correlation task (to detect sensor manipulation) or an intrusion detection task. As an illustrative example, consider the open source intrusion detection mechanisms Tripwire [4] and Bro [5] that detect integrity violations in the host and at the network level, respectively[1]. The default configurations of Tripwire and Bro contain several tasks (see Table I).

TABLE I
Illustration of Security Tasks*

*The corresponding application for each of the security tasks is specified in the parenthesis – TR: Tripwire, BR: Bro.

| Task | Function |
|---|---|
| Check own binary of the security routine (TR) | Compare the hash value of the application binary (*e.g.,* `/usr/sbin/tripwire`, `/usr/local/bro/bin`, *etc.*) |
| Check executables (TR) | Check hash of the file-system binary (`/bin`, `/sbin`) |
| Check critical libraries (TR) | Check library hashes (`/lib`) |
| Check device and kernel (TR) | Check hash of peripherals and kernel information in `/dev` and `/proc` |
| Check config files (TR) | Check configuration hashes (`/etc`) |
| Monitor network traffic (BR) | Scan network interface (*e.g.,* `en0`) |

We propose to incorporate security mechanisms into a multicore setup by implementing them as separate *sporadic tasks*. Unlike periodic real-time/control tasks, security tasks may not have strict period/deadline requirements. A metric of success for such security tasks could be, for instance, how quickly they can detect security violations (*e.g.,* an intrusion). This is in contrast with the measure of "control loop performance" for real-time tasks. The challenge then, is to determine the *right periods* (*viz.,* minimum inter-execution time) and *core assignment* for the security tasks. It is not trivial to determine the execution frequency and core assignment of security tasks (*i.e.,* what security tasks will execute on *which core* and with *what frequency*). For instance, some critical security routines may be required to execute more frequently than others. However, if the frequency of execution is too high (*e.g.,* shorter period) then it will use too much of the processor time and lower the system utilization for real-time

[1]We use Tripwire and Bro as examples of security applications to be integrated into multicore RTS – the ideas presented here apply more broadly.

tasks. Hence, the security mechanism itself might prove to be a hindrance to the system and reduce the overall functionality or worse, negatively impact the system 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.

Our focus here is to integrate security in an existing (*e.g.,* legacy) system where it is harder to *(a)* modify the micro-architecture (say inclusion of extra hardware/processor cores) or *(b)* change real-time task parameters (such as execution time and/or period). Existing work that integrate security in RTS either focuses on single core systems [6]–[11] and/or require modification of system parameters [6]–[9], [11], [12] and thus are not applicable for systems where it is harder to change the real-time task parameters.

In this paper we introduce HYDRA[2], a scheme for multicore RTS that finds a suitable assignment of security tasks in order to ensure that they can execute with a frequency close to what a designer expects. The main contributions of this work can be summarized as follows:

- Integrating security mechanisms in a multicore setup where changing existing real-time task parameters is not an option.
- A mathematical model to jointly obtain the assignment of security tasks to respective cores with execution frequency close to the desired values (Section III-A).
- An iterative scheme, HYDRA, that jointly finds the assignment and period of the security tasks (Section III-B).
- Comparisons of HYDRA with *(i)* assigning all security tasks to a single dedicated core and *(ii)* an 'optimal' multicore allocation scheme (Section IV).

We evaluate HYDRA with synthetic workloads as well as a representative real-time control system and security applications (Section IV).

## II. System and Security Model

### A. Real-time Tasks

Let us consider a multicore platform comprised of $M$ identical cores denoted by the set $\mathcal{M} = \{\pi_1, \pi_2, \cdots, \pi_M\}$ where we schedule a set $\Gamma_R = \{\tau_1, \tau_2, \cdots, \tau_{N_R}\}$ of $N_R$ independent sporadic real-time tasks. Each real-time task $\tau_r \in \Gamma_R$ is characterized by the tuple $(C_r, T_r, D_r)$ where $C_r$ is the worst-case execution time (WCET), $T_r$ is the minimum separation (*e.g.,* period) between two successive invocations and $D_r$ is the relative deadline. In this work, we consider *partitioned fixed-priority preemptive scheduling* [13] since *(a)* it does not introduce task migration costs and *(b)* it is widely supported in many commercial and open-source real-time OSs (*e.g.,* QNX, OKL4, real-time Linux, *etc.*). We assume that real-time task priorities are distinct and assigned according to rate monotonic (RM) [14] order. We also assume that tasks have implicit deadline, *e.g.,* $D_r = T_r, \forall \tau_r \in \Gamma_R$.

We assume that real-time tasks are *schedulable* and assigned to the cores using existing multicore task partitioning algorithm [13]. Since the taskset is schedulable, the following necessary condition will hold [15]:

$$\sum_{\tau_r \in \Gamma_R} \mathsf{DBF}(\tau_r, t) \leq Mt, \quad \forall t > 0 \qquad (1)$$

where the *demand bound function* $\mathsf{DBF}(\tau_r, t)$ computes the cumulative maximum execution requirements of the real-time task $\tau_r$ and it is defined as follows: $\mathsf{DBF}(\tau_r, t) = \max\left(0, \left(\left\lfloor \frac{t-D_r}{T_r} \right\rfloor + 1\right) C_r\right)$.

### B. Threat Model

In this work we consider a generic threat model where a malicious adversary can use various techniques to attack the RTS. For example, the adversary might intercept the information over the communication channel, forge messages or prevent normal requests from being processed. The adversary can also attack services within the OS, say, could compromise the file system resulting in corrupted information or could delay the delivery of control commands that may cause some tasks to miss deadline. Other than trying to aggressively crash the system, the adversary may utilize side-channels to monitor the system behavior and infer certain system information (*e.g.,* user tasks, thermal profiles, cache information, *etc.*) that eventually leads to the attacker actively taking control of the system. Our focus is on threats that can be dealt with by integrating additional security tasks. The addition of such tasks may necessitate changing the schedule of real-time tasks as was the case in earlier work [6]–[9], [11]. In this work we focus on situations where added security tasks are not allowed to impact the schedule of existing real-time tasks as is often the case when integrating security into existing multicore systems.

### C. Security Tasks

Since our goal is to *ensure security without any modification of real-time task parameters*, we propose to integrate security tasks as independent sporadic tasks. Let us consider additional $N_S$ security tasks denoted by the set $\Gamma_S = \{\tau_1, \tau_2, \cdots, \tau_{N_S}\}$. We follow the sporadic security task model [10] and characterize each security task $\tau_s$ by the tuple $(C_s, T_s^{des}, T_s^{max})$ where $C_s$ is the WCET, $T_s^{des}$ is the best period (minimum inter-arrival time) between successive releases (*i.e.,* $F_s^{des} = \frac{1}{T_s^{des}}$ is the desired frequency for $\tau_s$ effective security monitoring and/or intrusion detection) and $T_i^{max}$ is the maximum period beyond which security monitoring will not be effective. We assume that periods for security tasks are assigned based on the desired monitoring frequency[3]. Hence $pri(\tau_{s_1}) > pri(\tau_{s_2})$ if $T_{s_1}^{max} < T_{s_2}^{max}$ where $pri(\tau_i)$ denotes the priority of $\tau_i$. Security tasks also have implicit deadlines (*e.g.,* they are required to complete execution before its period).

One fundamental problem while integrating security mechanisms is to determine *which security tasks will be assigned to which core and executed when*. Although security tasks can execute in any of the $M$ available cores and any period $T_s^{des} \leq T_s \leq T_s^{max}$ is acceptable, the actual task-to-core assignment and the periods of the security tasks are not known apriori. The goal of HYDRA therefore is to jointly find the core-to-task assignment and suitable periods for security tasks.

## III. Assignment of Security Tasks with Period Adaptation

One way to integrate security mechanisms into existing systems without perturbing real-time task behavior is to execute security tasks with the *lowest priority* as compared

[2]In Greek mythology Hydra is a serpent with multiple heads. We refer to our scheme as HYDRA since we are trying to maximize the potential across multiple 'heads' (cores).

[3]For the purpose of this work we consider that the priorities are assigned based on the desired monitoring frequency. Such a frequency-based priority assignment, however, is not a requirement – the proposed scheme will work with any given priority assignment policy.

to the real-time tasks [10]. Thus security tasks will execute *opportunistically* in the slack time (*e.g.,* when other real-time tasks are not running). As mentioned earlier, actual periods of the security tasks are not known and we need to *adapt* the periods within acceptable ranges to optimize the trade-offs between schedulability and defense against intrusions. We measure the security of the system by means of the *achievable periodic monitoring* and our goal is to minimize the perturbation between the achievable (unknown) period $T_s$ and the given desired period $T_s^{des}$ for all security tasks $\tau_s \in \Gamma_S$. Therefore we consider the following metric [10]:

$$\eta_s = \frac{T_s^{des}}{T_s} \tag{2}$$

that denotes the *tightness* of periodic monitoring (*e.g.,* how close the period of the security task is to the desired period) and bounded by $\frac{T_s^{des}}{T_s^{max}} \leq \eta_s \leq 1$. 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, very frequent execution of security tasks may impact the schedulability of the system (due to higher utilization). The metric in Eq. (2) allows us to measure how close the security tasks are able to get to their desired monitoring frequencies.

Note that arbitrarily setting $T_s = T_s^{des}$ for all (or some) security tasks $\tau_s \in \Gamma_S$ may lead to the system becoming *unschedulable* since low-priority security tasks may miss deadlines due to interference from higher priority tasks. Also exhaustively finding all possible acceptable periods for the security tasks for all available cores is not feasible. It will cause an exponential blow-up as numbers of tasks and cores increase. For instance for a given taskset $\Gamma_S$, there is a total of $|\mathcal{M} \times \Gamma_s|$ assignments possible[4] (where $A \times B = \{(a,b) \mid a \in A \wedge b \in B\}$ and $|\cdot|$ denotes set cardinality) and for each combination the period for each security task $\tau_s \in \Gamma_S$ can be any value within the range $[T_s^{des}, T_s^{max}]$. In order to address this combinatorial problem we obtain the periods of the security tasks by framing it as an optimization problem.

## A. Formulation as an Optimization Problem

*1) Objective Function and Bounds on Period:* Let us consider the vector $\mathbf{X} = [x_s^m]^{\mathrm{T}}_{\forall \tau_s \in \Gamma_S, \forall \pi_m \in \mathcal{M}}$ where $x_s^m = 1$ if $\tau_s$ is assigned to $\pi_m$ and 0 otherwise. Recall that our goal is to find a task assignment that minimizes the difference between achievable and desired periods (*e.g.,* maximize the tightness) for all the security tasks. Hence we define the following objective function:

$$\max_{\mathbf{X},\mathbf{T}} \sum_{\pi_m \in \mathcal{M}} \sum_{\tau_s \in \Gamma_S} x_s^m \omega_s \eta_s = \sum_{\pi_m \in \mathcal{M}} \sum_{\tau_s \in \Gamma_S} x_s^m \omega_s \frac{T_s^{des}}{T_s} \tag{3}$$

where $\mathbf{T} = [T_s]^{\mathrm{T}}_{\forall \tau_s \in \Gamma_s}$ is the (unknown) period vector that needs to be determined and $\omega_s$ reflects the priority (higher priority tasks would have large $\omega_s$). Besides, in order to satisfy the frequency of periodic monitoring, the security task needs to satisfy the following constraint:

$$T_s^{des} \leq T_s \leq T_s^{max}, \ \forall \tau_s \in \Gamma_s. \tag{4}$$

Finally, each security task must be assigned to exactly one core: $\sum\limits_{\pi_m \in \mathcal{M}} x_s^m = 1, \quad \forall \tau_s \in \Gamma_s$.

[4]For instance, when $M = 8$ cores and $N_S = 10$ tasks there is a total of $3.518437208883 \times 10^{13}$ possible assignments.

*2) Schedulability Constraint:* Since the security tasks are executed with a priority lower that all real-time tasks, they will suffer interference from all real-time and high priority security tasks executing in the same core. Let $hp_S(\tau_s) \subset \Gamma_S$ denote the set of security tasks with a higher priority than $\tau_s$. The worst-case release pattern of $\tau_s$ occurs when $\tau_s$ and all high-priority tasks are released simultaneously [16]. Using response time analysis [17] we can determine an upper bound to the interference experienced by $\tau_s$ for a given core $\pi_m$ as follows:

$$I_s^m = \sum_{\tau_r \in \Gamma_R} \mathbb{I}_r^m \left(1 + \frac{T_s}{T_r}\right) C_r + \sum_{\tau_h \in hp_S(\tau_s)} x_h^m \left(1 + \frac{T_s}{T_h}\right) C_h \tag{5}$$

where $\mathbb{I}_r^m = 1$ if the real-time task $\tau_r$ is partitioned to core $\pi_m$ and 0 otherwise.

The first and second term in Eq. (5) represent the amount of interference from real-time and high-priority security tasks, respectively. Note that the assignment of real-time tasks to cores is known by assumption. In order to ensure that each security task $\tau_s$ will complete its execution before its deadline on its assigned core, the following constraint needs to be satisfied:

$$C_s + I_s^m \leq T_s, \quad \forall \tau_s \in \Gamma_s, \forall \pi_m \in \mathcal{M} : x_s^m = 1. \tag{6}$$

The variables $\mathbf{X}$ and $\mathbf{T}$ in the above formulation turn the problem into a *non-linear combinatorial optimization problem* that is NP-hard. We therefore propose an iterative algorithm HYDRA that jointly finds the security tasks' period and core assignment.

## B. Algorithm Development

As mentioned earlier, jointly finding the security task assignment and periods is an NP-hard problem. Even for fixed periods, finding the assignment for security tasks turns the problem to a bin-packing problem that is known to be NP-hard [15]. Existing partitioning heuristics (*e.g.,* first-fit, best-fit, *etc.*) [13] are *not* directly applicable in our context since the real-time requirements (*e.g.,* minimize the number of cores so that all real-time tasks can meet deadlines) are often different from the security requirements (*e.g.,* execute security tasks more often to improve intrusion detection rate without violating real-time constraints).

For a given task $\tau_s$ and allocation vector $\mathbf{X}$, let us rewrite the optimization problem as follows:

$$\max_{T_s} \eta_s, \text{ Subject to: } T_s^{des} \leq T_s \leq T_s^{max}, \ C_s + I_s^m \leq T_s. \tag{7}$$

Notice that for a given assignment $\mathbf{X}$ (see Algorithm 1), the period $T_s$ is the only variable (when the $T_h, \forall \tau_h \in hp_S(\tau_s)$ is known) in $I_s^m$ (see Eq. (5)). Although the period adaptation problem in Eq. (7) is a constraint optimization problem it can be transformed into a convex optimization problem (that is solvable in polynomial time). For details of this reformulation we refer the readers to the Appendix.

The proposed HYDRA algorithm (summarized in Algorithm 1) works as follows. We start with the highest priority security task $\tau_s$ and try to obtain the best period for each available core $\pi_m \in \mathcal{M}$ by solving the period adaptation problem introduced in Eq. (7) (Line 4). If there exists a set of cores $\mathcal{M}'_s \subseteq \mathcal{M}$ for which the optimization problem is feasible (*e.g.,* an optimal period is obtained satisfying the real-time constraints) we pick the core $\pi_{m^*} \in \mathcal{M}'_s$ that gives the

**Algorithm 1** HYDRA: Task Allocation and Period Adaptation

```
Input: Input taskset $\Gamma = \{\Gamma_R \cup \Gamma_S\}$ and the partition of real-time tasks $\mathbb{I} = [\mathbb{I}_r^m]^{\mathrm{T}}_{\forall \tau_r \in \Gamma_R, \forall \pi_m \in \mathcal{M}}$
Output: The security task allocation $\mathbf{X} = [x_s^m]^{\mathrm{T}}_{\forall \tau_s \in \Gamma_S, \forall \pi_m \in \mathcal{M}}$ and periods $\mathbf{T} = [T_s]^{\mathrm{T}}_{\forall \tau_s \in \Gamma_S}$, if the taskset is schedulable, Unschedulable otherwise.
1:  Initialize $x_s^m := 0,\ \forall \tau_s \in \Gamma_S,\ \forall \pi_m \in \mathcal{M}$
2:  for each security task $\tau_s \in \Gamma_S$ (from higher to lower priority) do
3:      for each core $\pi_m \in \mathcal{M}$ do
4:          Solve the optimization problem in Eq. (7)
5:      end for
6:      Let $\mathcal{M}'_s \subseteq \mathcal{M}$ is the set of core(s) for which the optimization problem is feasible
7:      if $\mathcal{M}'_s = \emptyset$ then
8:          /* Unable to find suitable period for $\tau_s$ */
9:          return Unschedulable
10:     end if
11:     Find the core $\pi_{m^*} = \underset{\pi_m \in \mathcal{M}'_s}{\operatorname{argmax}}\ \eta_s^m$ where $\eta_s^m$ is the tightness of $\tau_s$ obtained for $\pi_m$
12:     Set $x_s^{m^*} := 1$   /* Assign $\tau_s$ to $\pi_{m^*}$ */
13:     Update period $T_s := T_s^{m^*}$ where $T_s^{m^*}$ is the period obtained by solving optimization for $\pi_{m^*}$
14: end for
15: return $(\mathbf{X}, \mathbf{T})$   /* Return the allocation vector and periods */
```

maximum tightness (Line 11) and allocate the security task to core $\pi_{m^*}$ (Line 12). This will ensure that the more critical security tasks will get a period close to the desired one. We repeat this process for all security tasks to jointly obtain the assignment and periods. If for any security task $\tau_j$ the set of available cores $\mathcal{M}'_j$ is empty (*e.g.,* the optimization problem is infeasible) we return the taskset as *unschedulable* (Line 9) since it is not possible to find any suitable core with given taskset parameters. This unschedulability result will provide hints to the designers to update the parameters of security tasks (and/or the real-time tasks, if possible) in order to integrate security for the target system.

## IV. Evaluation

We evaluated HYDRA along two fronts: *(i)* on parameters derived from a real UAV control system (Section IV-A) and *(ii)* synthetically generated tasksets to explore the design space (Section IV-B). Recall from Section I that our goal is to explore the possible ways in which security could be integrated in multicore-based real-time platforms. The HYDRA mechanism presented in this paper assumes that the real time tasks are distributed across *all* available cores. Another design choice available is to allocate a dedicated core for security while the real-time tasks are assigned to the remaining cores. In this Section, we compare HYDRA to this alternate mechanism for security task allocation – that we refer to henceforth as the "SingleCore" allocation mechanism. Given the taskset is schedulable, one of the benefits of the SingleCore scheme is that there is no requirement for assigning security tasks. While evaluating SingleCore, all the real-time tasks are partitioned into $M-1$ cores leaving the other core for security tasks. Notice that in the SingleCore scheme security tasks do not suffer any interference from real-time tasks (*e.g.,* the first term in Eq. (5) is zero). For a given assigned core $\pi_m$, the decision variable $x_s^m$ is known for all $\tau_s$ and the optimization problem can be solved using an approach similar to the one described in the Appendix.

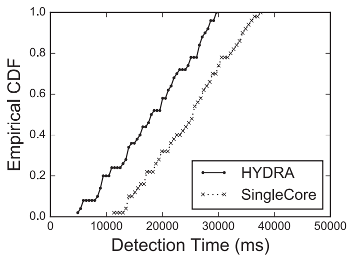

Fig. 1. HYDRA vs. SingleCore: empirical CDF of intrusion detection time. The empirical CDF is defined as $\widehat{F}_\alpha(\jmath) = \frac{1}{\alpha}\sum_{i=1}^{\alpha} \mathbb{I}_{[\zeta_i \leq \jmath]}$, where $\alpha$ is the total number of experimental observations, $\zeta_i$ is time to detect the attack in at the $i$-th experimental observation and $\jmath$ represents the $x$-axis values (*e.g.,* detection time). The indicator function $\mathbb{I}_{[\cdot]}$ outputs 1 if the condition $[\cdot]$ is satisfied and 0 otherwise.

### A. Case-study with a UAV Control System and Security Applications

The goal of this experiment was to observe the runtime behavior of HYDRA. For a real-time application, we considered a UAV control system [18]. It includes following real-time tasks (refer to earlier work [18, Tab. 1] for the task parameters): *Guidance* (selects the reference trajectory), *Slow and Fast navigation* (read sensor values according to the required update frequency), *Controller* (executes the closed-loop control functions), *Missile control* (fires missile) and *Reconnaissance* (collects sensitive information and send data to the control center). For the security application, we considered Tripwire [4] and Bro [5] that detects integrity violations in the system both at host and network level, respectively (refer to Table I). For this experiment we considered a quad-core system (*e.g.,* $M = 4$). We executed the security tasks on an 1 GHz ARM Cortex-A8 processor with Xenomai 2.6 [19] patched real-time Linux kernel (version 3.8.13-r72) and used ARM cycle counter registers (*e.g.,* CCNT) to obtain the timing parameters (*e.g.,* WCET). We used `GPkit` [20] library and `CVXOPT` [21] solver to obtain the periods.

The work-flow of our experiment was as follows. For each of the trials, we observed the schedule for $500$s and during any random time of execution we triggered synthetic attacks[5] (*e.g.,* that corrupts the file system and network packets). We assumed that the intrusions were correctly detected by the security tasks (*e.g.,* there is no false positive/negative errors) and measured the empirical CDF of worst-case detection time. From Fig. 1 we can observe that paralleling security tasks across cores leads to faster intrusion detection time for HYDRA (*e.g.,* higher empirical CDF). From our experiment we found that *on average* HYDRA can provide $27.23\%$ faster detection rate for a quad-core system. While SingleCore scheme does not experience any interference from real-time tasks, however, low priority security tasks can still suffer inference from high priority security tasks. Therefore running security tasks in a single core leads to higher periods and consequently poorer detection time.

### B. Experiment with Synthetic Tasksets

We used parameters similar to those in related work [10], [22]. We performed experiments for $M = \{2, 4, 8\}$ cores. Each taskset instance contained $[3M, 10M]$ real-time and $[2M, 5M]$

[5]Our goal here is to analyze the security from the scheduling perspective. Thus instead of assuming any specific intrusion (or detection capabilities of security tasks), HYDRA allows designer to integrate any security mechanism required to defend targeted attack surfaces.

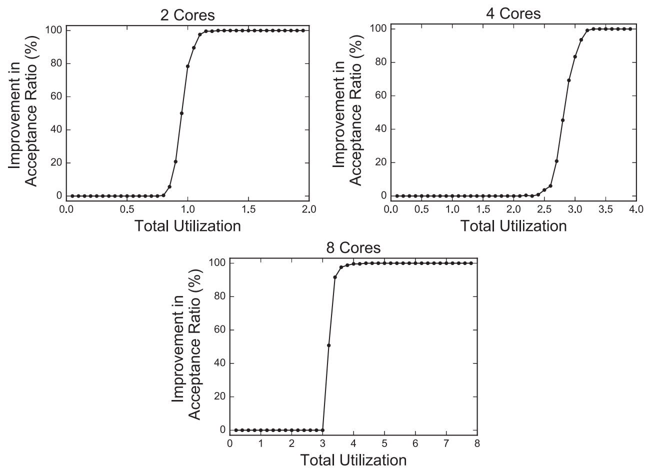

Fig. 2. The improvement in acceptance ratio for $2, 4$, and $8$ core system. The improvement is given by $\frac{\delta_{\text{SingleCore}} - \delta_{\text{HYDRA}}}{\delta_{\text{SingleCore}}} \times 100\%$ where $\delta_{(\cdot)}$ is the acceptance ratio of scheme $(\cdot)$.

security tasks. Each real-time task had periods between $[10 \text{ ms}, 1000 \text{ ms}]$. The desired periods for the security tasks were selected from $[1000 \text{ ms}, 3000 \text{ ms}]$ and the maximum allowable period was assumed to be $T_s^{max} = 10 T_s^{des}$, $\forall \tau_s$. The real-time tasks were partitioned across multiple cores using a best-fit [13] strategy.

In each experiment, the total taskset utilization was varied from $0.025M$ to $0.975M$ with step size $0.025M$. For a given number of tasks and total system utilization, the utilization of individual tasks were generated from an unbiased set of utilization values using the Randfixedsum algorithm [23]. Total utilization of the security tasks were set to be no more than $30\%$ of the real-time tasks. For each utilization value, we randomly generated 250 tasksets. In other words, for each core configuration a total of $39 \times 250 = 9750$ tasksets were tested. We only considered tasksets that satisfied the necessary condition in Eq. (1), as any taskset that fails the condition is trivially unschedulable.

*1. Experiment with Core Assignment Schemes:* We compared HYDRA with SingleCore in terms of *acceptance ratio*. The acceptance ratio is given by the number of *schedulable* tasksets (*e.g.,* that satisfy all real-time constraints) over the generated ones. The x-axis in Fig. 2 represents the total system utilization (*e.g.,* utilization of both real-time and security tasks). The y-axis represents improvement in acceptance ratio comparing HYDRA with SingleCore for different values of $M$. For lower utilization values both schemes have similar performance (*e.g.,* improvement is zero) since the system has enough slack to execute security tasks. However as we see from the figure, for higher utilization values HYDRA outperforms SingleCore – when all security tasks share a core, it causes more interference and reduces the overall schedulability (*e.g.,* unable to find any solution that respects all the real-time constraints[6]).

*2. Comparing with Optimal Multicore Assignment:* The result of an empirical comparison of HYDRA with an optimal solution (*e.g.,* a solution of the formulation described in Section III-A that finds the variables $\mathbf{X}$ and $\mathbf{T}$) is presented in Fig. 3 where we exhaustively searched for all possible combinations for a small setup with $M = 2$ cores and up to $N_S = 6$ security tasks. To find the optimal solution, we examined each of the $M^{N_S}$ possible assignments of security tasks to cores. For each assignment, we then determined the value of the period vector $\mathbf{T}$ that maximized the cumulative tightness by solving a convex optimization problem (see Appendix).

[6]Note that security tasks also have real-time constraints.

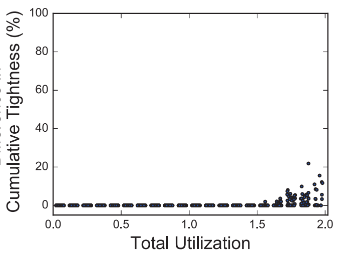

Fig. 3. Comparing HYDRA with optimal solution: we consider $M = 2$ and $N_S \in [2, 6]$ with other parameters similar to that mentioned in Section IV-B.

The x-axis of Fig. 3 represents total system utilization and y-axis is the difference in cumulative tightness (*e.g.,* $\Delta_\eta = \frac{\eta_{\text{OPT}} - \eta_{\text{HYDRA}}}{\eta_{\text{OPT}}} \times 100\%$) for HYDRA and the optimal solutions. As shown in the figure, for low to medium utilization cases, HYDRA's performance is similar to the optimal solution (*e.g.,* the difference is zero). However for higher utilizations performance degrades. This is because HYDRA follows an iterative best-fit strategy to find the periods (and assignment). Hence for higher utilization values the lower priority tasks may not get periods close to the desired values (and the cumulative tightness degrades). As we see from the figure, the degradation (in cumulative tightness) is no more than $22\%$ and that may be acceptable given the exponential computational complexity of finding an optimal solution.

## V. Discussion

While we take a step towards developing a model for integrating security mechanisms into multicore RTS, our initial attempt can be extended into several directions. HYDRA statically partition the tasks to the cores. However in practice security tasks can also move across multiple cores if there is available slack at runtime (for faster detection and better schedulability). While there exists methods for global scheduling policy [13] where tasks can migrate across cores, casting real-time scheduling problems into RTS security domain requires further research.

In this work we consider security tasks are independent and preemptive. However some critical security task may require non-preemptive execution to perform desired checking. 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

There has been some work [6]–[9] on reconciling the addition of security mechanisms into RTS that considered periodic task scheduling where each task requires a security service whose overhead varies according to the quantifiable level of the service. The issues regarding information leakage through storage channels also addressed in prior research [11]. All of the aforementioned work, however, only consider single core system and require modification of system parameters. In early work [10] we used the concept of hierarchical scheduling and proposed to execute the security mechanisms with a lower priority than the real-time tasks for a single core system. Unlike prior work here we focus on integrating security in multicore platforms.

Although not in the context of security in RTS, there exists other work [24], [25] in which the authors statically assign the periods for control tasks. While this previous work focused on single core systems and optimizing period of *all* the tasks, our goal is to ensure security without violating timing constraints of the real-time tasks in a multicore setup.

In contrast to proposed scheduler-level solution, recent work [12], [26], [27] on hardware/software architectural frameworks aim to protect multicore RTS against security vulnerabilities. Compared to our scheme that works for any $m$-core system, these preceding frameworks mainly focus on dual core architecture and require architectural modifications that may not be suitable for existing RTS.

## VII. Conclusion

This paper presents an evaluation of a good heuristic mechanism (HYDRA) for assigning security tasks into a multicore RTS. Engineers can now evaluate the design choices of such assignments to improve the overall security (and hence, safety) of systems with real-time requirements. Since we provide comparisons of our solution with two extremes – an 'optimal' assignment strategy and isolating all security tasks to a single core – we are able to provide valuable hints to designers on how to build security into such systems.

## References

[1] J. Westling, "Future of the Internet of things in mission critical applications," 2016.
[2] N. Falliere, L. O. Murchu, and E. Chien, "W32. stuxnet dossier," *White paper, Symantec Corp., Security Response*, vol. 5, p. 6, 2011.
[3] R. M. Lee, M. J. Assante, and T. Conway, "Analysis of the cyber attack on the ukrainian power grid," *SANS Industrial Control Systems*, 2016.
[4] "Tripwire," https://github.com/Tripwire/tripwire-open-source.
[5] "The Bro network security monitor," https://www.bro.org.
[6] T. Xie and X. Qin, "Improving security for periodic tasks in embedded systems through scheduling," *ACM TECS*, vol. 6, no. 3, p. 20, 2007.
[7] 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.
[8] X. Zhang, J. Zhan, W. Jiang, Y. Ma, and K. Jiang, "Design optimization of security-sensitive mixed-criticality real-time embedded systems," in *IEEE ReTiMiCS*, 2013.
[9] K. Jiang, P. Eles, and Z. Peng, "Optimization of secure embedded systems with dynamic task sets," in *DATE*, 2013, pp. 1765–1770.
[10] 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.
[11] 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.
[12] 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.
[13] 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.
[14] 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.
[15] S. Baruah and N. Fisher, "The partitioned multiprocessor scheduling of sporadic task systems," in *IEEE RTSS*, 2005.
[16] 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.
[17] 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.
[18] 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.
[19] "Xenomai – real-time framework for Linux," https://xenomai.org.
[20] E. Burnell and W. Hoburg, "GPkit software for geometric programming," 2017. [Online]. Available: https://github.com/hoburg/gpkit
[21] L. Vandenberghe, "The CVXOPT linear and quadratic cone program solvers," 2010. [Online]. Available: http://cvxopt.org/documentation/coneprog.pdf
[22] R. I. Davis, A. Burns, J. Marinho, V. Nelis, S. M. Petters, and M. Bertogna, "Global and partitioned multiprocessor fixed priority scheduling with deferred preemption," *ACM TECS*, vol. 14, no. 3, pp. 47:1–47:28, 2015.
[23] P. Emberson, R. Stafford, and R. I. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *WATERS*, 2010, pp. 6–11.
[24] E. Bini and A. Cervin, "Delay-aware period assignment in control systems," in *IEEE RTSS*, 2008, pp. 291–300.
[25] 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.
[26] 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.
[27] F. Abdi, M. Hasan, S. Mohan, D. Agarwal, and M. Caccamo, "ReSecure: A restart-based security protocol for tightly actuated hard real-time systems," in *IEEE CERTS*, 2016, pp. 47–54.
[28] S. Boyd, S.-J. Kim, L. Vandenberghe, and A. Hassibi, "A tutorial on geometric programming," *Opt. & Eng.*, vol. 8, no. 1, pp. 67–127, 2007.
[29] S. Boyd and L. Vandenberghe, *Convex optimization*, 2004.

## Appendix
## Solution to the Period Adaptation Problem

The period adaptation problem given in Section III-A is a constrained optimization problem and not very straightforward to solve. Therefore we reformulate the optimization problems as a geometric program (GP) [28].

A nonlinear optimization problem can be solved by GP if the problem is formulated as follows [28]:

$$\min_{\mathbf{Y}} f_0(\mathbf{y}), \text{ Subject to: } f_i(\mathbf{y}) \leq 1, \quad i = 1, \cdots, z_p, \quad \text{and}$$
$$g_i(\mathbf{y}) = 1, \quad i = 1, \cdots, z_m$$

where $\mathbf{y} = [y_1, y_2, \cdots, y_z]^{\mathrm{T}}$ denotes the vector of $z$ optimization variables. The functions $f_0(\mathbf{x}), f_1(\mathbf{y}), \cdots, f_{z_p}(\mathbf{y})$ are *posynomial* and $g_1(\mathbf{y}), \cdots, g_{z_m}(\mathbf{y})$ are *monomial* functions, respectively. A monomial function is expressed as $g_i(\mathbf{y}) = c_i \prod_{l=1}^{L_i} y_l^{a_l}$, where $c_i \in \mathbb{R}^+$ and $a_l \in \mathbb{R}$. A posynomial function (*i.e.,* the sum of the monomials) can be represented as $f_i(\mathbf{y}) = \sum_{l=1}^{L_i} c_l y_1^{a_{1l}} y_2^{a_{2l}} \cdots y_z^{a_{1l}}$, where $c_l \in \mathbb{R}^+$ and $a_{jl} \in \mathbb{R}$. We can maximize a non-zero posynomial objective function by minimizing its inverse. In addition, we can express the constraint $f(\cdot) < g(\cdot)$ as $\frac{f(\cdot)}{g(\cdot)} \leq 1$.

Based on above discussion we can rearrange the objective function as $\min_{T_s} \left(T_s^{des}\right)^{-1}$. Likewise period bound constraint in Eq. (4) can be represented as $T_s^{des} {T_s}^{-1} \leq 1$ and $(T_s^{max})^{-1} T_s \leq 1$, respectively. In addition, the schedulability constraint in Eq. (6) can be rewritten as: $(C_s + I_s^m) T_s^{-1} \leq 1$ where $I_s^m = \sum_{\tau_r \in \Gamma_R} \mathbb{I}_r^m (T_r + T_s) T_r^{-1} C_r + \sum_{\tau_h \in hp_S(\tau_s)} x_h^m (T_h + T_s) T_h^{-1} C_h$.

The above GP reformulation is not a convex optimization problem since the posynomials are not convex functions [28]. However, by using logarithmic transformations (*e.g.,* representing $\tilde{T}_s = \log T_s$ and hence $T_s = e^{\tilde{T}_s}$, and replacing inequality constraints of the form $f_i(\cdot) \leq 1$ with $\log f_i(\cdot) \leq 0$), we can convert the above formulation into a convex optimization problem that can be solved using standard algorithms, such as *interior-point* method in polynomial time [29, Ch. 11].