Real-Time Systems Security Through Scheduler Constraints

Sibin Mohan*, Man-Ki Yoon†, Rodolfo Pellizzoni‡ and Rakesh Bobba*
*Information Trust Institute, University of Illinois at Urbana-Champaign, Urbana, IL 61801
†Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801
‡Dept. of Electrical and Computer Engineering, University of Waterloo, Ontario Canada
Email: {*sibin, †mkyoon, *rbobba}@illinois.edu, ‡rodolfo.pellizzoni@uwaterloo.ca

***Abstract*—**

Real-time systems (RTS) were typically considered to be invulnerable to external attacks, mainly due to their use of proprietary hardware and protocols, as well as physical isolation. As a result, RTS and security have traditionally been separate domains. These assumptions are being challenged by a series of recent events that highlight the vulnerabilities in RTS. In this paper we focus on integrating security as a first class principle in the design of RTS: we show that certain security requirements can be specified as real-time scheduling constraints. Using information leakage as a motivating problem, we illustrate our techniques with fixed-priority (FP) real-time schedulers. We evaluate our approach and discuss tradeoffs. Our evaluation shows that many real-time task sets can be scheduled under the proposed constraints without significant performance impact.

I. Introduction

Embedded real-time systems (RTS) are used for the monitoring and control of physical systems and processes in varied domains such as aircraft, submarines, other vehicles (both autonomous as well as manual), spacecraft, critical infrastructures (*e.g.,* power grid and water systems) and industrial plants to name but a few. The next-generation of real-time control systems will need to support multiple, interconnected, complex functions including mission control, data acquisition as well as processing and communication [24]. To reduce cost, power consumption and weight embedded designers are progressively moving towards integrated architectures where all such functionalities are implemented as separate, yet inter-dependent, real-time tasks on individual processing nodes, possibly using commercial-off-the-shelf (COTS) technology.

Until recently, most RTS were considered to be invulnerable against software security breaches because real-time systems were *(a)* physically isolated from the outside world, *(b)* used specialized protocols and *(c)* executed on dedicated hardware. These systems are increasingly being connected together, sometimes through the use of unsecured networks such as the Internet. Furthermore, malware developers and sophisticated adversaries are able to overcome air-gaps. This is evident from recent successful attacks on industrial control systems [6], malicious code injection into the telematics units of modern automobiles [3], [15], demonstration of potential vulnerabilities in avionics systems [32] and attacks on UAVs [25].

Given the time and resource constraints under which RTS operate, vulnerabilities in RTS differ considerably from those of traditional enterprise systems. Threats faced by real-time systems could vary in scope and effect; from the leakage of critical data [27] to hostile actions due to lack of authentication [3], [15], [32]. Arguably some of the aforementioned attacks succeeded because such systems were not designed to be secure against attacks. However, simply adding security mechanisms that provide confidentiality (*e.g.,* encryption), integrity protection (*e.g.,* message authentication) and availability (*e.g.,* replication) without considering the real-time and embedded nature of such systems will not be effective. Recognizing this, researchers have proposed adding security as a new dimension to be considered during embedded system design [13]. There has been some work on reconciling the addition of security mechanisms with real-time properties [16], [30], [35]. Specifically, researchers proposed changes to EDF scheduling [16], [35] to optimize the level of security achieved (measured in terms of strength of security keys and primitives) while ensuring that real-time deadlines are met.

In this paper, we consider the issue of information leakage between real-time tasks with different security levels. It is well understood that the use of shared resources can lead to information leakage between tasks without the use of explicit communication (*e.g.,* [14], [21]). The issue of covert timing channels between tasks of different security levels in the rate monotonic (RM) scheduler was previously considered [27]. In contrast, we focus on information leakage due to the sharing of resources[1] such as the cache, DRAMs, and I/O bus. In particular, every time there is a switch between tasks belonging to different security levels there is a possibility of information leakage through shared resources.

We propose to reduce this potential for information leakage via shared resources by integrating security at the design phase of RTS in the form of intelligent scheduling constraints. We discuss various methods of integrating such constraints into scheduling policies for real-time systems (Sections III, IV and V) and derive some analytical bounds for the same (Section IV).

This work is supported in part by a grant from ONR (N00014-13-1-0707 and N00014-12-1-0046). Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of sponsors.

[1]Other than the processor core.

Specifically, we focus on the Fixed Priority (FP) scheduling algorithms [18] that cover a large class of real-time systems.

As part of the high-level contributions of this paper, we:

1) demonstrate the use of constraints on real-time scheduling policies as a means of enforcing security properties (mitigating information leakage through storage channels over implicitly shared resources[2] in this case) as discussed in Section III;
2) discuss enhancements to the FP scheduler for the integration of such constraints (Sections III, IV and V) and provide analysis bounds for an instance of the problem (non-preemptive FP algorithm) as seen in Section IV; and
3) show additional ideas for the integration of such constraints for other instances of the problem; they aim to improve performance compared to non-preemptive FP (Section V).

We now present the adversary and system model in Section II.

II. Adversary and System Model

The issue of information leakage arises in systems with multiple levels of security where tasks at different security levels share the computing platform. In complex real-time systems this situation can arise in multiple scenarios such as when modules sourced from different providers are integrated together. For example, consider an avionics system designed as per the DO-178B model [5]. The navigation system which is less critical can be sourced from a less trustworthy vendor and can be placed at a lower confidentiality level than the flight control system to which it provides information. In such a case even if the navigation system were to be compromised it will not be privy to the critical data from the latter [32].

Another example is of unmanned aerial vehicles (UAVs) where: *(a)* a set of real-time tasks/components, $\{R\}$, controls the UAV and *(b)* another set of tasks, $\{I\}$, gathers, processes and communicates information back to the base station. $\{R\}$ could include tasks to calculate the flight path and control code to manage the engines. $\{I\}$ includes software components that control a camera to capture images, one or more tasks to process the images and another component to communicate the processed/raw images back to the command center. Now, the components in $\{R\}$ will have a higher criticality in terms of the real-time requirements while those in $\{I\}$ will have higher security requirements in terms of confidentiality, *i.e.,* the information that the UAV captures should not leak out. Security levels could also vary within each set of tasks. For instance, the information being processed by the task that calculates the flight path of the UAV must be more secure than the information being processed by the control task (essentially the low level sensory/actuation information).

Another scenario: legacy applications are moved over to modern computing platforms due to the obsolescence of older processor architectures, *e.g.,* the "RePLACE" system from Northrup Grumman [8], [9], [23]. Such solutions take legacy applications and execute them on modern processors by providing emulation frameworks so that the application still believes it executes on the original platform. However, due to lack of physical isolation that they relied on before, leakage of information across applications due to the underlying shared resources is a real possibility. This could be exacerbated if the original applications belonged to different security levels.

A. Adversary Model

We assume that an adversary can either insert new tasks (that respect the real-time guarantees of the system to avoid immediate detection) or compromise one or more existing tasks. The main objective of this attacker is to passively glean secure information by observation of shared resource usage. Also, while the adversary can observe the usage of shared resources (like caches), it cannot snoop on the RAM contents of other tasks (due to the existence of virtual memory and/or memory controllers). Active adversaries that can tamper with the system operation are out of scope for this work.

B. System Model

We consider a uniprocessor system following the Liu and Layland task model [17] that contains a set of sporadic tasks, $\{\tau\}$ where each task $\tau_i \in \{\tau\}$ has the parameters: $\{p_i, c_i, d_i\}$, where p_i is the period, c_i is the worst-case execution time and d_i is the deadline, with $d_i \leq p_i$. We also assume that (for the FP scheduling policy), the set of real-time priorities, $\{Pri\}$ are fixed such that, $\forall \tau_i, \tau_j \in \{\tau\}$ and $pri_{\tau_i}, pri_{\tau_j} \in \{Pri\}$ then either $pri_{\tau_i} \prec pri_{\tau_j}$ if τ_i has a higher priority than τ_j or $pri_{\tau_i} \succ pri_{\tau_j}$ if τ_j has a higher priority than τ_i. Of course, it is also possible that pri_{τ_i} and pri_{τ_j} share the same priority level. For the sake of simplicity, we write pri_{τ_i} as pri_i. Let hep_i be the set of tasks with a higher or equal priority than a given task τ_i (excluding τ_i itself) and lp_i be the set of tasks with lower priority than τ_i. We also assume that time is measured in *integral quantities* and not as a continuous value.

We assume that the set of security levels for tasks, S, forms a *total order*. Hence, any two tasks (τ_i, τ_j) in the system may have one of the following two relationships when considering their security levels, $s_i, s_j \in S$: *(i)* $s_i \prec s_j$, meaning that τ_i has higher security level than τ_j or *(ii)* $s_j \prec s_i$. We plan to generalize our system model to consider a partial ordering [4] among security levels in an extended version of this paper. Note that, while it might be the case that the tasks with higher real-time priorities may also have the higher security levels, we do not limit ourselves to this assumption. While we do present cases where there exists a relationship between the priority levels and security levels, our techniques are more general than these special cases imply.

III. Security and Scheduling

As stated in Section I, we propose to mitigate the problem of information leakage via shared resources among tasks of varying security levels, by placing constraints on scheduling algorithms. While there may be multiple ways to mitigate information leakage (*e.g.,* hardware-supported cache partitions), the approach of modifying or constraining scheduling algorithms is appealing because, *(a)* it is a software based approach and hence easier to deploy compared to hardware based approaches; *(b)* it allows for reconciling the security requirements with real-time or schedulability requirements; and *(c)* availability of extensive analytical tools for real-time systems allows for better evaluation and understanding of the trade-offs between the security and schedulability requirements.

[2]Sometimes referred to as "storage channels with timing exploitation".

We now present a simple example that is used to illustrate our ideas in this context. Consider the case of a simple RTS with just two periodic tasks, a high priority task $H\{p_H, c_H, d_H\}$ and a low priority task $L\{p_L, c_L, d_L\}$. Let us assume that the security levels for this system match the real-time priorities, *i.e.,* $s_H \prec s_L$; hence, information from H must not leak to L. Now, these tasks must be scheduled on a single processor, P, so that both deadlines (d_H, d_L) are satisfied. Let the tasks be scheduled according to a Fixed Priority (FP) scheduling policy. If L (or any part thereof) executes immediately after (any part) or all of H, then there is the potential for information leakage (*i.e.,* L could inspect the shared resource contents that were recently used by H). Many attacks only need information about which cache lines were requested/evicted and the timing for such actions to be able to gather information about the data that the higher security level tasks used [21]. *The main intuition is that a penalty must be paid for each shared resource in the system, every time tasks switch between security levels.* In this case, *the cache must be flushed before a new task is scheduled*[3].

The solutions suggested in this paper are not specific to shared caches; we believe that the similar concepts are applicable to other shared resources as well. For instance, DRAMs or even the I/O bus could be shared resources that attackers may target to gather information (*e.g.,* it has been shown that the I/O bus can carry traffic related to a previously executing task even after a new task has been scheduled [20]). Hence, flushing mechanisms could be used for these other resources as well; *e.g.,* in the case of I/O buses, the "flushing mechanism" could just introduce a delay so that outstanding requests have time to complete before the new task starts up; in the case of a disk, the flushing task could reset the seek heads to some initial position. In general, there exist many resources in the system that are 'stateful'; hence, what one task does will have a timing effect on the next task. There exist ways to use this timing data to extract information about the previous executing task(s). Our solution is to add an operation (that has a fixed overhead) to "reset" the state of the device.

One point to note is that the flush task is not meant to immediately clear out the contents of the cache; the easiest way would have been to just supply a null voltage to the cache for a short duration. In fact, the flush task needs to ensure that the *system is left in a consistent state*. Any outstanding writebacks, bus write requests, data transfer, *etc.* need to complete before the contents of the cache are wiped. Hence, a finite amount of time must be spent in performing all of these operations.

A. Scheduling Constraints

An initial analysis, then, yields the following constraints that can be applied to the real-time scheduler:

C_1 No instance of L can be scheduled right after any instance of H

C_2 If an instance of L is preempted by a job of H and then resumes when H completes then there is still a potential for information leakage – hence such a situation must be avoided.

Constraint C_2 is a special instance of C_1. Further analysis presents a few options for integrating these constraints into the scheduler:

[**A**] Since the shared resource is the offending party, we can ensure that it is always flushed/cleaned out whenever we see transitions of the type $H \rightarrow L$;

[**B**] Ensure that all jobs of one kind (H or L) complete before transitioning to the other; or

[**C**] prevent L from being preempted by H once it has been scheduled.

The first method, A, can be implemented by the use of a synthetic 'flush task' (FT) that is *always* executed once L is scheduled after a job of H – the job of the FT is to flush the contents of the shared resource; this ensures that any following task will not be able to access the contents of the resource. The problem with this method, of course, is that the synthetic task incurs an overhead (that may be small, yet still constant).

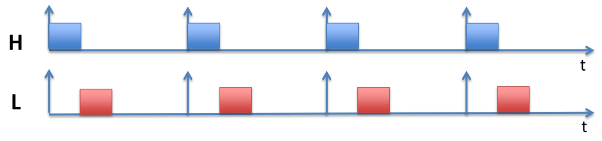


Fig. 1. Task Switches for a Two-Task System

Of course, the problem with the use of an additional task to cleanse out the cache is that the overheads scale (often linearly) with the number of switches of the type $H \rightarrow L$ (Figure 1 where the horizontal axis is time; upward pointing arrows are new instances of tasks – also serve as deadlines for the previous invocations). Hence, the number of FT instances, $N_{ft} = N_H$, *i.e.,* the number of jobs for H. If we also assume that the FT is invoked on $L \rightarrow H$ switches (to prevent acknowledgements in covert channels[4]) then the number of FT instances is: $N_{ft} = N_H + N_L$, *i.e.,* the *total number of jobs*.

The second and third methods (B, C) will have to be used in conjunction with the first (A), since it is impossible to avoid the $H \rightarrow L$ transitions completely. These two techniques though, could significantly reduce the number of calls to the FT. Method (B) results in the least number of switches between tasks of H and L – just *one*. The problem, though, is obvious – one of the two tasks will very likely miss its deadline. Hence, for all practical purposes, we cannot use this method.

The final technique (C) is an additional constraint that is placed on the scheduling algorithm to handle constraint C_2. While this avoids situations where a preempted instance of L resumes execution immediately after the completion of H (thus reducing the number of preemptions and executions of FT), it could suffer from the problem of *priority inversion* where a high priority task is forced to wait for a low priority task to complete [22]. This technique could also result in lower utilization for some task sets (often resulting in task sets becoming unschedulable) since we are often forcing higher priority (critical) jobs to wait until an *entire lower priority job completes its execution*.

For the rest of this paper, we will refer to technique (A) as "`PreFlush`" (PF), *i.e.,* the basic scheduling algorithm with preemptions and FT, while technique (C) will be referred to as "`ConstrainedPreFlush`" (CPF), *i.e.,* the scheduling algorithm with a limited set of non-preemptions and (where necessary) FT invocations.

[3]We will discuss techniques to avoid an inordinate number of cache flushes later on in the paper.

[4]We will relax this assumption later in the paper to obtain tighter bounds.

B. PF and CPF for a Partial Order of Security Levels

It is now easy to extrapolate PF and CPF to multiple tasks and security levels, S. The rules for **PF** are:

1) for every pair of tasks, $\tau_i, \tau_j |\ s_i \prec s_j$, invoke the FT on every transition of the type, $\tau_i \rightarrow \tau_j$;
2) also invoke the FT on every transition of the type, $\tau_j \rightarrow \tau_i$, *i.e.,* either on completion of τ_j or if τ_i preempts τ_j.

The second rule ensures that the lower security task is not able to 'respond' (with acknowledgements) in case a covert channel is setup [4], [7], [27]. While this does cut down the efficiency of any such channels (and significantly reduces the possibility of information leakage), we can still get good protection and increased utilization by not enforcing this rule on most systems. If we prevent the flow of information between tasks by invoking the flush task on $\tau_i \rightarrow \tau_j$ transitions (where $s_i \prec s_j$), then even if a compromised τ_j is able to send back acknowledgements, it will not be effective. Hence, we present a simpler version of the PF mechanism called **Half-PF** *i.e.,* for every pair of tasks, $\tau_i, \tau_j |\ s_i \prec s_j$, invoke the FT on every transition of the type, $\tau_i \rightarrow \tau_j$ only. For the remainder of this paper, we focus on Half-PF rather than PF, unless explicitly mentioned.

The rules, then, for **CPF** are as follows:

1) for every pair of tasks, $\tau_i, \tau_j |\ pri_i \prec pri_j\ \&\ s_i \prec s_j$, *prevent* τ_i from preempting τ_j; τ_i executes on the completion of τ_j if it is the highest priority task that is ready to execute at that point in time
2) for every pair of tasks, $\tau_i, \tau_j |\ pri_i \prec pri_j\ \&\ s_i \succ s_j$, *allow* τ_i to preempt τ_j; this preemption is fine since the FT would have been invoked once anyways when τ_j completes execution

For the first rule, if there exist one or more tasks, τ_k, such that $pri_i \prec pri_k\ \&\ s_i \prec s_k$ then τ_i is still allowed to execute after τ_j (even though we could further reduce FT invocations by not allowing it to). The reason is to avoid the situation where τ_i faces an inordinate priority inversion scenario. Hence, we are only concerned with direct preemptions and not indirect ones.

IV. FP and Security

Fixed Priority (FP) schedulers [18] form a well known class of static scheduling algorithms. In this section we discuss how to combine the non-preemptive FP scheduler with our security-related constraint, Half-PF. We start with the non-preemptive FP scheduler because it is one of the easier algorithms to analyze and implement. We believe that the techniques in this paper will not only *(a)* provide insights into how such security-related constraints can be integrated into real-time schedulers but also *(b)* demonstrate how a worst-case response-time analysis can be carried out for such situations. We discuss CPF and other variations of FP later on.

For the remainder of this section, let τ_i be the task under analysis and c_{ft} be the execution time of one FT. We assume that each FT is executed together with the task that requires it: *i.e.,* if a job of task τ_j follows the execution of a job of task τ_i, with $s_i \prec s_j$, a FT is invoked when the job of task τ_j is scheduled for execution, effectively increasing its execution time to $c_j + c_{ft}$. In other words, FT are also executed non-preemptively. Our analysis strategy is: we use standard response time analysis for non-preemptive FP to compute, at each iteration, the number of higher or equal priority jobs that interfere with τ_i. Then we determine the maximum number of FT invocations required by such jobs and we correspondingly increase the computed response times. As usual, we iterate until convergence is achieved.

Hence, the worst-case response time $R_i(k+1)$ of task τ_i at iteration k can be computed [2] as:

$$R_i(k+1) = B_i + N_{ft}(S, \{I_j | \tau_j \in hep_i\}) c_{ft} + \sum_{\forall j \in hep_i} (I_j c_j) + c_i, \tag{1}$$

where B_i represents the maximum blocking time induced by lower priority tasks and their FT, N_{ft} is the worst-case number of FT required by either interfering higher or equal priority tasks or by the task under analysis and I_j is the number of instances of a higher or equal priority task τ_j that interfere with τ_i, which for non-preemptive FP is:

$$I_j = \left\lfloor \frac{R_i(k) - c_i}{p_j} + 1 \right\rfloor. \tag{2}$$

The maximum blocking time B_i can be computed as:

$$B_i = \max_{\forall \tau_j \in lp_i} \bar{c_j} - 1, \tag{3}$$

where $\bar{c_j} = c_j + c_{ft}$ if there exists a task $\tau_k \in S$ such that $s_k \prec s_j$ and $\bar{c_j} = c_j$ otherwise; *i.e.,* if there is any task that can cause a lower priority job of τ_j to suffer a FT then we need to add c_{ft} to the blocking time generated by τ_j. The -1 term accounts for the fact that the lower priority blocking task must arrive at least one time unit before the activation of τ_i.

Note that we derive $N_{ft}(S, \{I_j | \tau_j \in hep_i\})$ based only on the ordering of security levels and the number of interfering jobs of each task in hep_i; in other words, we make no assumption on the arrival time or other timing parameters of higher or equal priority jobs within the busy interval. In the remaining of this section, we will show how to compute a bound to N_{ft} in polynomial time in $\sum_{\tau_j \in hep_i} I_j$. We begin with some definitions. Intuitively, the concept of a *valid job sequence* captures all valid schedules based on the available information $S, \{I_j | \tau_j \in hep_i\}$, *i.e.,* all sequences of jobs that can invoke a FT during the busy interval for τ_i.

Definition 1 (Valid Job Sequence). *A valid job sequence ψ for $S, \{I_j | \tau_j \in hep_i\}$ is a sequence of $\sum_{\tau_j \in hep_i} I_j + 2$ jobs in S such that: (a) the first job is a job of any task of S; (b) the last job is a job of τ_i; (c) the sequence of $\sum_{\tau_j \in hep_i} I_j$ intermediate jobs is any permutation of the union of I_j jobs for each task τ_j in hep_i. Let $\Psi(S, \{I_j | \tau_j \in hep_i\})$ be the set of all valid job sequences for $S, \{I_j | \tau_j \in hep_i\}$.*

Definition 2 (Number of FT for ψ). *$N(\psi)$ is the number of FT required by jobs of valid job sequence ψ, with the exclusion of the first job of the sequence; i.e., for any two successive jobs of any tasks τ_j, τ_k in the sequence, a FT is required for τ_k if and only if $s_j \prec s_k$ in S.*

Let t_0 be the time at which a job of task τ_i arrives. Then a job of a task in lp_i could be executing at t_0, but after this job completes, only higher or equal priority tasks can possibly execute before τ_i. Furthermore, the processor could also be idle at t_0, in which case a job of any task (either higher priority, lower priority or τ_i itself) could have finished executing last before t_0. Hence, when considering the job sequence, we need to consider a job of any task as the first job in the sequence. Since we make no assumption on the arrival time of tasks in the busy interval, we then need to consider any possible permutation of the $\{I_j | \tau_j \in hep_i\}$ higher or equal priority jobs; finally, the job of τ_i must execute.

Since valid job sequences corresponds to valid schedules[5] for $S, \{I_j | \tau_j \in hep_i\}$, we can obtain the desired FT value as:

$$N_{ft}(S, \{I_j | \tau_j \in hep_i\}) = \max_{\psi \in \Psi(S, \{I_j | \tau_j \in hep_i\})} N(\psi). \quad (4)$$

Unfortunately, enumerating all possible permutations of higher or equal priority jobs would take factorial time. Hence, we now show how to transform the problem of computing the maximum $N(\psi)$ over all valid job sequences into a max flow problem on a graph derived from S and $\{I_j | \tau_j \in hep_i\}$. Intuitively, we construct the graph by using "sender" and "receiver" nodes corresponding to jobs in any valid sequence. We add an edge with capacity 1 between a sender and a receiver corresponding to tasks τ_j, τ_k respectively to represent the fact that an FT is required if a job of τ_j is followed by a job of τ_k in the sequence.

Definition 3 (FT Graph)**.** *The FT Graph for $S, \{I_j | \tau_j \in hep_i\}$ is a flow graph (V, E) with the following set of vertexes V:*

1) a source *vertex and a* sink *vertex;*
2) a sender *vertex $SendF$ and a* receiver *vertex $RecvL$;*
3) for each $\tau_j \in hep_i$, I_j sender *vertexes $\{Send_{j,1}, \ldots, Send_{j,I_j}\}$ and I_j* receiver *vertexes $\{Recv_{j,1}, \ldots, Recv_{j,I_j}\}$;*

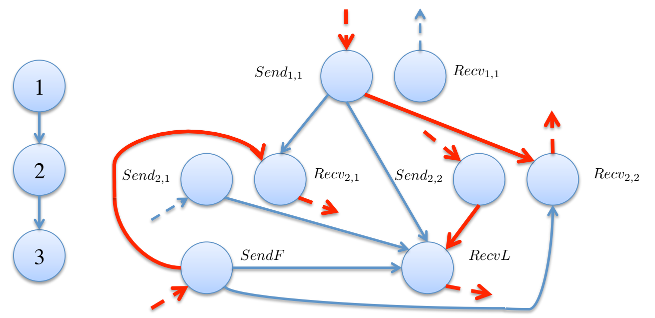
and the following set of directed edges E, where all edges have a capacity of 1:

1) an edge from the source to every sender vertex (including $SendF$);
2) an edge from every receiver vertex (including $RecvL$) to the sink;
3) if there exists a task $\tau_k \in S$, $s_k \prec s_i$, an edge from $SendF$ to $RecvL$;
4) for every task $\tau_j \in hep_i$, if there exists a task $\tau_k \in S$, $s_k \prec s_j$, an edge from $SendF$ to every receiver $Recv_{j,q}, 1 \le q \le I_j$;
5) for every task $\tau_j \in hep_i$ such that $s_j \prec s_i$, an edge from every sender $Send_{j,q}, 1 \le q \le I_j$, to $RecvL$;
6) for every pair of tasks $\tau_j, \tau_k \in hep_i$ such that $s_j \prec s_k$, an edge from every sender $Send_{j,q}, 1 \le q \le I_j$, to every receiver $Recv_{k,l}, 1 \le l \le I_k$.

In the following, let $f(e)$ denote the flow assigned to an edge $e \in E$ in a given flow assignment and $F = \sum_{e \in E} f(e)$ be the total flow value for that assignment. A flow assignment is valid if each flow $f(e)$ is between 0 and the edge capacity (always 1 for the FT graph) and the flow conservation constraint is obeyed at all vertices except the source and sink. Hence, the sum of the flows on incoming edges to a vertex must be equal to the sum of flows on outgoing edges from that vertex. We use the notation $v \to v'$ to denote an edge from vertex v to v'.

An an example, consider the security ordering in Figure 2(a). Let τ_3 be the task under analysis and $I_1 = 1, I_2 = 2$. The

[5]Note that when considering the timing constraints of the jobs in hep_i, there are valid job sequences that are not valid schedules; however, we still compute a safe upper bound on the number of FT since every possible schedule is captured by a sequence ψ.



(a) S (b) Corresponding FT Graph

Fig. 2. Example: FT graph creation and max flow. τ_3 is the task under analysis, with $I_1 = 1$ and $I_2 = 2$. Red arrows represent a valid max flow assignment.

equivalent FT Graph is shown in Figure 2(b). Note that for the sake of clarity we do not represent the source and sink vertices; all dotted lines directed towards a sender vertex represent edges originating from the source (edges of Type 1 in Definition 3) and all dotted lines going out of a receiver vertex represent edges ending in the sink (Type 2). Each higher or equal priority job is represented by two vertices, a sender and a receiver. An edge is added between $Send_{j,q}$ and $Recv_{k,l}$ if executing τ_j followed by τ_k would result in a FT (Type 6). $SendF$ represents the first job in a valid job sequence; it only has a sender vertex since it is not preceded by another job in the sequence, and based on Definition 1, it can represent any task in the system. Similarly, $RecvL$ represents the last job in a valid job sequence, *i.e.,* the job of the task under analysis. Edges of Types 3-5 represent FT required by $RecvL$ or by the job following $SendF$ (possibly $RecvL$ itself, Type 3). Note that each sender vertex can receive at most one unit of flow from the source and the sender can provide at most one unit of flow to other receivers; intuitively, this is because the job represented by the sender vertex is followed by one other job in a valid job sequence. Similarly, each receiver vertex can receive at most one unit of flow from sender vertexes.

A valid max flow assignment is shown by the bolded, red arrows in Figure 2(b). One unit of flow is sent on the following edges (and zero everywhere else): $source \to SendF \to Recv_{2,1} \to sink$, $source \to Send_{1,1} \to Recv_{2,2} \to sink$, $source \to Send_{2,2} \to RecvL \to sink$, for a flow value $F = 3$. This flow assignment corresponds to the valid job sequence $\psi = \{\tau_1, (\tau_2, \tau_1, \tau_2), \tau_3\}$, which similarly results in $N(\psi) = 3$, since FT are required in the transitions from τ_1 to τ_2 and τ_2 to τ_3. Note that the sequence can be constructed by following the sequence of sender and receiver nodes with non-zero flow, starting from $SendF$ and ending with $RecvL$. Finally, note that the max flow is strictly dependent on the number I_j of jobs of each higher priority task τ_j.

The following theorem formally proves that the maximum flow on the FT graph represents an upper bound to the number of FT for any valid job sequence. Furthermore, while the bound is not always tight, it is at most one higher than the maximum number of FT. Intuitively, the bound is one FT higher because there exist some integer flow assignments that do not correspond to a valid job sequence. In particular, an assignment might have flow both on the edge between $SendF$ and $RecvL$ and on other edges. This cannot result in a valid sequence, since it requires that the first job in the sequence is immediately followed by the job of task under analysis τ_i, but at the same time, there must be other FT caused by intermediate higher priority jobs. However, we can show that we can always obtain a valid sequence by removing the flow on at most one edge.

Theorem 1. *Let $\bar{F}$ be the max flow value for the FT graph (V, E) for $S, \{I_j | \tau_j \in hep_i\}$. Then:*

$$\bar{F} - 1 \leq \max_{\psi \in \Psi(S, \{I_j | \tau_j \in hep_i\})} N(\psi) \leq \bar{F} \tag{5}$$

Proof: Since all capacities in the FT graph are integers, there must exist an integer flow assignment with max flow value $\bar{F}$. We first show (Part A of the proof) that for any valid job sequence $\psi \in \Psi(S, \{I_j | \tau_j \in hep_i\})$, there exists a valid flow assignment with flow value of $F = N(\psi)$; thus $\max_{\psi \in \Psi(S, \{I_j | \tau_j \in hep_i\})} N(\psi) \leq \bar{F}$. We then show (Part B of the proof) that for any valid integer flow assignment with flow value F, there exists a valid job sequence with $N(\psi) \geq F - 1$; thus $\max_{\psi \in \Psi(S, \{I_j | \tau_j \in hep_i\})} N(\psi) \geq \bar{F} - 1$. This concludes the proof.

Part A: Given job sequence ψ and for ease of notation, let us number the jobs in ψ of higher or equal priority task τ_j (with the possible exception of the first job in the sequence, if it is a job of a task in hep_i) as $\tau_{j,1}, \ldots, \tau_{j,I_j}$, based on the order in which they appear in ψ. As in Definition 1, we refer to such jobs as intermediate jobs in the sequence. Also assume that the first job in ψ is a job of task τ_k (based on Definition 1, τ_k can be any job in S). We show how to construct a flow f such that $F = N(\psi)$. We consider two cases.

Case A.1: there are no tasks in hep_i. Then ψ is composed of only two jobs, of τ_k and τ_i respectively. If $s_k \prec s_i$, we set of flow of 1 on edges $source \rightarrow SendF \rightarrow RecvL \rightarrow sink$, for a flow value $F = 1$ equal to $N(\psi)$; note that the flow is valid since there must be an edge $SendF \rightarrow RecvL$ in the FT Graph according to Definition 3. If instead it does not hold $s_k \prec s_i$, then we can simply set all flows to zero since $N(\psi) = 0$.

Case A.2: there is at least at task in hep_i. In this case, there must be at least one intermediate job of the task in hep_i in ψ. We thus construct the flow in the following manner: A.2.1) Let $\tau_{j,1}$ be the first intermediate job in the sequence. Then if $s_k \prec s_j$, we set of flow of 1 on edges $source \rightarrow SendF \rightarrow Recv_{j,1} \rightarrow sink$. A.2.2) For any two successive intermediate jobs $\tau_{j,q}$ and $\tau_{l,v}$, if $s_j \prec s_l$ we set of flow of 1 on edges $source \rightarrow Send_{j,q} \rightarrow Recv_{l,v} \rightarrow sink$. A.2.3) Let τ_{l,I_l} be the last intermediate job in the sequence. Then if $s_l \prec s_i$, we set of flow of 1 on edges $source \rightarrow Send_{l,I_l} \rightarrow RecvL \rightarrow sink$. It is easy to see that the flow is valid, since we only send flow on edges that exist in the FT Graph according to Definition 3, and furthermore incoming/outgoing flow is added at most once to each node with the exception of $source$ and $sink$. Furthermore, an outgoing flow of 1 from the source (and incoming flow of 1 to the sink) is added every time for two successive jobs of τ_j, τ_l in the sequence it holds $s_j \prec s_l$; hence, $F = N(\psi)$ concluding this part of the proof.

Part B: We need to show that given an integer flow f, we can construct a sequence ψ such that $N(\psi) \geq F - 1$. If $F = 0$, the proof is trivial, so assume $F > 0$. Since the flow is integer, either $f(e) = 0$ or $f(e) = 1$ for any edge in the FT Graph. We say that a sender node is *activated* if the incoming/outgoing flow through the node is 1 (note it cannot be greater than one); similarly for a receiver node. We now define the concept of a vertex sequence for f as follows: B.1) a vertex sequence is an alternation of activated sender and receiver vertexes; B.2) the sequence starts with a sender and ends with a receiver; B.3) if a sender is followed by a receiver, there must be a flow of 1 on the edge between that sender and receiver; B.4) if a receiver is followed by a sender, both vertexes must be generated from the same task (i.e., $Recv_{j,q}$ and $Send_{j,l}$ for some task τ_j and $q, l \in [1, I_j]$). A maximal vertex sequence is a vertex sequence that cannot be extended by adding other vertexes to either the beginning or end of the sequence. Since each sender node can only have one outgoing edge with a flow of 1 and each receiver can only have one incoming edge with a flow of 1, it is easy to see that each vertex can appear only once in each vertex sequence; furthermore, a vertex cannot appear in two maximal sequences. Finally, there must exist at least one maximal sequence since $F > 0$, and the total value of F must be equal to half the sum of the number of vertexes in all maximal vertex sequences (for each flow of 1 from $source$ to $sink$ in the graph, we have a pair of sender and receiver in a sequence).

We can then construct ψ as follows: B.5) if there is any maximal vertex sequence that starts with $SendF$, pick any task τ_k such that $s_k \prec s_j$, assuming that $Recv_{j,q}$ is the second vertex in the sequence, and start ψ with a job of τ_k; note that there must be at least one such task τ_k, otherwise according to Definition 3 there would be no edge between $SendF$ and $Recv_{j,q}$. Then add intermediate jobs to ψ based on the vertex sequence (i.e., for each successive receiver and sender vertexes $Recv_{j,q}$ and $Send_{j,l}$, add one job of τ_j, plus one job for the final receiver in the sequence). If there is no maximal vertex sequence that starts with $SendF$, start ψ with a job of any task in S. B.6) for any maximal vertex sequence that does not start with $SendF$ or ends with $RecvL$, add intermediate jobs to ψ based on the vertex sequence as in the previous point. B.7) Add to ψ any intermediate jobs that is needed to complete the job sequence and have not been already added or will be added in the next point. B.8) If there is any maximal vertex sequence that ends with $RecvL$, add intermediate jobs to ψ based on the vertex sequence. Otherwise, simply add a job of τ_i as the last job in the sequence. Since each vertex appears in either one maximal vertex sequence or none at all, it is easy to see that ψ is a valid job sequence. Furthermore, for each maximal vertex sequence with $2K$ vertexes, by construction we have added K job transitions of the form $s_j \prec s_l$ in ψ. Hence, $N(\psi)$ must be at least equal to F, satisfying the $N(\psi) \geq F - 1$. However, we need to cover one final case: B.9) the FT graph might have at least two maximal vertex sequences, including one that starts with $SendF$ and ends with $RecvL$. In this case, we cannot construct a full sequence ψ based on both maximal vertex sequences. However, we can still construct a sequence ψ based on B.5 - B.8 by removing $SendF$ and $Recv_{j,q}$ from the vertex sequence, where $Recv_{j,q}$ is the second vertex in the sequence started with $SendF$. Using the same reasoning as above, this results in $N(\psi) \geq F - 1$, concluding the proof. ■

The max flow value can be computed by any max-flow algorithm. For instance the original Ford-Fulkerson algorithm has a complexity of $O\big(|E|\bar{F}\big)$. Since the number of edges in the FT graph is $O(|V|^2)$ and the max flow value is $O(V)$ (given that all capacities are 1), it follows that N_{ft} can be derived in $O\big((\sum_{\tau_j \in hep_i} I_j)^3\big)$ time.

Note: whether it is possible to derive exact bounds on the number of FT in polynomial time in the number of higher priority jobs is left as an open question. We plan to extend our analysis to preemptive FP as part of future work. While we believe that a similar graph theoretical approach could be used to bound the number of FT, the analysis is likely to be significantly more complex since each preempted job would appear multiple times in a valid job sequence.

V. Further Considerations for Scheduling

Designers of real-time systems must work towards optimizing a large number of parameters to ensure the (functional & temporal) correctness of the system. Security constraints are additional parameters that they must now be considered as part of this process. This begets the following questions:

1) What is the best ordering of security levels?
2) In fact, is there such a thing as the "best ordering" for security levels in RTS?
3) Is this "best ordering" related, in any way, to the real-time priorities of the task sets?

The most obvious answer, of course is *depends* – this is entirely up to the particulars of the system (what tasks are in the system, their properties, functionality, *etc.*). But perhaps we can provide some hints to designers so that if they have a choice, then they could tune their system to improve not just real-time performance but also security. We present two examples to highlight some of these issues. While our analysis (Section IV) has focused on non-preemptive scheduling algorithms, in the examples and discussion presented in this section, we will also talk about issues related to preemptive scheduling methods.

Example 1: Consider a security ordering for three tasks, $s_1 \prec s_2 \prec s_3$ where the real-time priorities are: $pri_1 \prec pri_2 \prec pri_3$; we also assume that all tasks are released at the same time (at least for this example). Now, consider the Half-PF and CPF constraints:

Half-PF: Under this constraint, every time there is a change of the type $pri_i \rightarrow pri_j$ (where $pri_i \prec pri_j$) there will a FT invocation. Hence, for the current set of ready jobs, *every time* a task completes and a new one is scheduled, it will result in a FT invocation. The only time that a FT will not execute is when a new instance of a higher priority task preempts a lower priority task. Also, it is often the case that higher priority tasks have shorter periods (*e.g.,* in RM [17] scheduling) – hence, the jobs of the higher priority tasks will often preempt the lower priority jobs and every time the latter resume their executions, a FT invocation will take place. It is also typical that lower priority jobs, while having the longer periods, also have longer execution times. Hence, the chances of such preemptions, followed by FT invocations, will be high.

CPF: even though tasks cannot be preempted by higher priority tasks for this example (see the definition of CPF in Section III-B), it still suffers from the overheads of many FT invocations. Every time a higher priority task is scheduled and then is followed by a lower priority task the FT must execute. This seems to result in the most number of FT executions for a set of ready tasks at any point in time as we see in Section VI. We refer to this ordering (where priorities and security levels are ordered along the same direction) as *forward ordering*:

Definition 4. *For any two tasks, τ_i, τ_j with priority ordering $pri_i \preceq pri_j$, a* **forward ordering** *states that the security levels will be $s_i \prec s_j$.*

Example 2: Now, let us change the security ordering of tasks from Example 1 to be $s_1 \succ s_2 \succ s_3$ while still preserving the real-time priorities as before, *i.e.,* $pri_1 \prec pri_2 \prec pri_3$; we also assume, as before, that all tasks are released at the same time. Now, consider Half-PF and CPF for this updated example:

Half-PF: In this case, since all the tasks are released at the same time, the highest priority task τ_1 will execute first, followed by τ_2 and then τ_3 will execute; during this sequence of executions, *the FT was never invoked*. The reason is simple – though the higher priority tasks have a lower security level, the number of transitions of the type $s_i \rightarrow s_j$ where $s_i \prec s_j$ is reduced (reduced to zero for these invocations in Example 2). The only instances when we need to execute the FT is when a preemption takes place or if a high priority task follows a low priority task. As mentioned above, most of the time (in a typical RTS) low priority tasks execute and though preemptions can occur, they only result in one additional FT.

CPF: the CPF constraint is able to do better than Half-PF at reducing the number of FT executions (for this example), since lower priority tasks cannot be preempted once they start. Hence, higher priority job can cause only *one* FT invocation (and that too only if it immediately follows a task with lower priority). Hence, this particular ordering of tasks seems to bring out the best behavior in the scheduling algorithms when the constraints Half-PF and CPF are applied (more in Section VI). We call this a *backward ordering*, where priorities and security levels are in the seemingly opposite directions; it is defined as:

Definition 5. *For any two tasks, τ_i, τ_j with priority ordering $pri_i \preceq pri_j$, a* **backward ordering** *states that the security levels will be $s_i \succ s_j$.*

We also introduce the notion of "random ordering" where two tasks, τ_i, τ_j with priority ordering $pri_i \preceq pri_j$ could exhibit any one of the following two behaviors regarding security level ordering: $s_i \prec s_j$, or $s_i \succ s_j$. Considering the varied nature of RTS and security constraints, it may often be the case that task sets will fall into this particular ordering.

Definition 6. **Random ordering***: Given a set of tasks $\{\tau\}$, it not possible to establish an exact relationship between their priority ordering and their security ordering.*

Note: By "random ordering" we do not mean that the ordering is random. It is a convenient phrase to describe the fact that we do not assign a specific order, ahead of time, to the tasks.

Definitions 4, 5 and 6 provide insights into the what could possible be the worst, best and average case situations for real-time systems with security constraints.

VI. Evaluation

We now evaluate the constraint-driven FP schedulers introduced in the previous sections. We first present our experimental setup, discuss the evaluation of the response time-based analyses (from Section IV) and present a set of simulations for other combinations of FP schedulers and security constraints. We also discuss some limitations for our work.

A. Experimental Setup

Part of the process of integrating the Half-PF and CPF into FP is to gain an understanding of the behavior of the modified algorithm(s). For this purpose, we set up simulation and analysis engines and analyzed thousands of task sets. Table I summarizes the parameters used for the generation of task sets used in our evaluation. We generated 2000 random, synthetic, task sets evenly from ten base utilization groups, $[0.02 + 0.1 \cdot i, 0.08 + 0.1 \cdot i]$ for $i = 0, \ldots, 9$, *i.e.,* 200 instances per group. The base utilization of an instance is defined as the total sum of the task utilizations. Each input instance consists of $[3, 10]$ tasks, each τ_i of which has a period $p_i \in [50, 100, \ldots, 950, 1000]$ and an execution time $c_i \in [3, 30]$. The deadline of each task is set to be equal to its period, *i.e.,* $d_i = p_i$. Since deadlines are equal to periods, we decided to assign task priorities according to the Rate Monotonic (RM) algorithm [17]. While we acknowledge that RM is not an optimal priority assignment algorithm for our task model, we point out that optimizing the priority assignment is outside the scope of this work.

TABLE I. EXPERIMENTAL PARAMETERS.

Parameter	Value
Number of tasks, N	$[3, 10]$
Task period, p_i	$[50, 100, \ldots, 950, 1000]$
Task execution time, e_i	$[3, 30]$
FT overhead	$\{1, 5, 10\}$

The overheads for the FT instances would depend on the actual resources, *e.g.,* in the case of a cache[6], it will be: $\frac{sizeof(cache)}{cache\_refill\_bandwidth}$; hence we use the values in the table ($[1, 5, 10]$) as placeholders. The values of c_{ft} (relative to the task execution times) might seem a little high but it depends on the system under consideration. Sure, if the highest value for execution time (30) is equated to say, 10 ms, then c_{ft} ranges from 0.3–3.3ms. This value does seem inordinately high considering that, for many architectures, the values lie in the 100's of microseconds range for cache flushes[7]. Of course things change if we were to assume that say, $30 = 1ms$ for the task execution times. Then the values of c_{ft} range from $33\mu s$ to $333.33\mu s$; admittedly the latter is a little high (relative to the task execution times), but the purpose of choosing this value is to show what happens if we end up using shared resources that have high associated costs for cleaning out their state. Typically we would see c_{ft} values closer to the lower end of the spectrum (*i.e.,* between $33 - 167\mu s$). The techniques presented in this paper would still be valid with other values for FT overheads.

For each task set instance, an ordering of security levels (S) is constructed: for each task τ_i, a task $\tau_j | s_i \prec s_j$, where $i < j$, is added to S with a probability of 0.5. Thus, in the resulting set, any τ_i can never have a lower security level than τ_j. This prevents the formation of cyclical security relationships.

We use the same generated task sets for both, *(a)* the evaluation of the analysis bounds from Section IV and *(b)* the simulation-based evaluation of the other techniques (in Section V). All tasks are released at time $t = 0$ and each simulation executes for the duration of the hyper-period HP of the given task set. The system keeps track of the response time of each job, denoted by r_i^k, that is calculated by the time duration that the k^{th} job of τ_i took to complete its execution e_i. The analysis engine, on the other hand, computes the worst-case response time based on the iteration from Section IV. Upon completion of a job, the simulation/analysis engine checks whether the response time exceeds its deadline. A task set is said to be *unschedulable* if there exists any τ_i such that $r_i^k > d_i$ for $k = 1, \ldots, \frac{HP}{p_i}$.

B. Evaluation of Response-Time Based Analysis

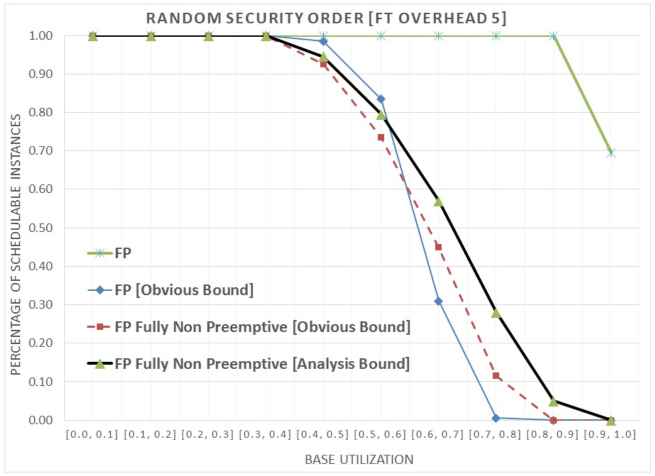


Fig. 3. Analysis-based Results [Random Ordering] [FT=5]

We first evaluate the Non-Preemptive FP+Half-PF combination (analyzed in general for FP in Section IV). We focus on the random ordering of security levels (Definition 6) since it represents the general case. Consider Figure 3 that shows results for an FT execution time of 5. The X-axis plots the utilization "bins" (or ranges) for the experiments while the Y-axis represents the total percentage of schedulable task sets for each bin. The various plots represent the FP variants: *(a)* the (vanilla) FP (thick green line); *(b)* FP with the obvious bounds (blue solid line) and two versions of the Non-Preemptive FP algorithm, *(c)* one based on the obvious/worst-case bound (red dotted line) and *(d)* one based on the worst-case response time analysis from Section IV and Equations 1 and 2 (thick black line). In the case of the "obvious bound" for both vanilla FP and non-preemptive FP, we refer to time taken by the worst-case number of FT invocations, N_{ft}. For the former, $N_{ft} = 2 * N_{hep_i} + 1$ and for the latter, $N_{ft} = N_{hep_i} + 1$ (N_{hep_i} is the number of higher or equal priority jobs for each task τ_i[8]). **Note**: vanilla FP does not suffer from any FT overheads – it does not implement the security constraints.

The random ordering of security levels provides insights into the typical performance for the scheduling algorithms. As expected, `FP` (the vanilla version) performs the best, simply because it does not implement any of the constraints from Section III-B. This is evident from the graphs where the bar for FP is at the top and schedules most task sets. Of course, the ability to schedule task sets drops as we reach the higher

[6]Essentially to flush and refill the cache.

[7]For instance, in the core i7 we have an 8 MB Level 3 cache and up to 21 GB/s memory bandwidth (more or less, since it is always hard to have an exact number). Hence, $c_{ft} = 380\mu s$. For a Tegra 3 device (quad core AFP A9) we have a 1 MB L2 and 6.4 GB/s memory bandwidth – that puts the c_{ft} at $156\mu s$; this is assuming you have to load from memory to flush (rather than having a flush instructions).

[8]We get these bounds based on the upper bounds on the number of preemptions for basic and non-preemptive FP algorithms.

utilization values (above 90%[9]).

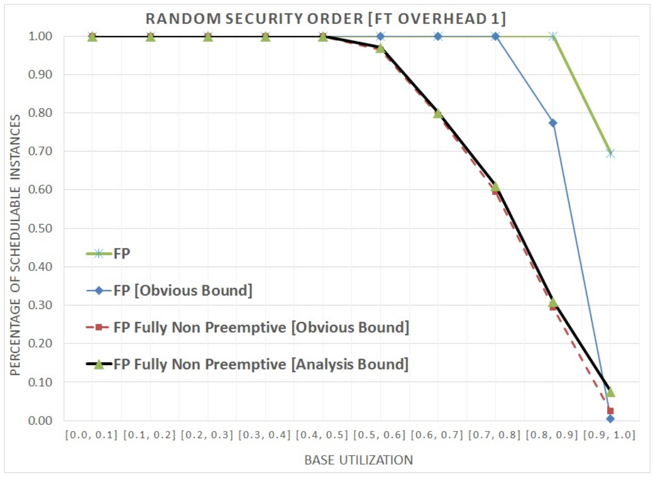


Fig. 4. Analysis-based Results [Random] [FT=1]

From this graph we observe that our analysis (black line) is able to obtain tighter bounds than the naive (obvious) bounds. Hence, following the max-flow based graph algorithm presented in Section IV, we are able to schedule more task instances. While the performance may not be as good as vanilla FP, designers of such systems can now choose to increase the security, albeit at reduced schedulability levels (for task sets with higher utilizations), of real-time systems.

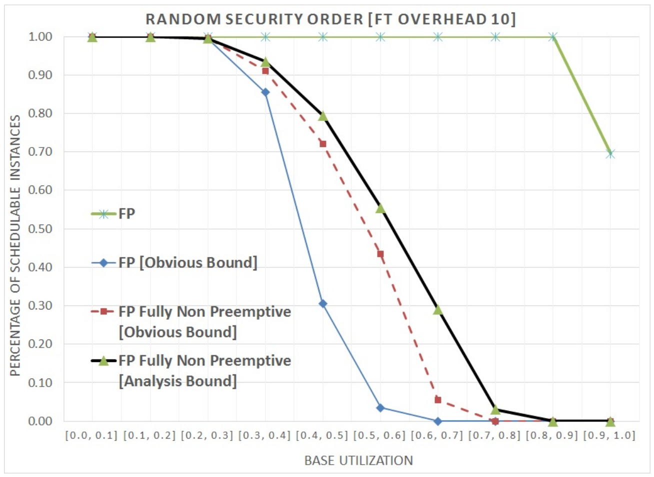


Fig. 5. Analysis-based Results [Random] [FT=10]

We also analyzed the effects of variability in the execution time of FTs on the schedulability of task sets (Figures 4 and 5). As expected, the overall performance (relative to basic FP) drops as the FT execution time increases. When FT overheads are low, the naive analysis (obvious bounds) comes close to the response-time-based analysis while the relative differences increase as the FT overheads go up. The exception is shown in Figure 4 where the naive method performs much better (getting much closer to vanilla FP in fact). The reason is that the FT overhead ($c_{ft} = 1$) is *much* lower than the execution times of the tasks; the scheduling algorithm is not hindered all that much by these low overheads and hence seems to perform much better when compared to the analysis-based results. When the FT overheads are closer to the typical values ($c_{ft} = 5$ in Figure 3) we see that performance of the naive analysis-based results drops (in comparison with the calculated worst-case bounds; *i.e.,* the black line). The relative performance drops further when c_{ft} increases further (Figure 5).

Hence, our techniques perform better when the overheads for flushing shared resources increase. This is quite important, since different shared resources (caches, DRAMs or even I/O buses) will demonstrate varying degrees of overheads for flushing their state and our methods are never worse (and usually better) than the typical conservative estimates.

One important observation is that *the modified algorithms (i.e., the ones with the security constraints integrated) are still able to schedule a large number of task sets*. The algorithms without these constraints start differentiating themselves only for the higher utilization values when the FT overheads become a factor for the modified algorithms. Hence, we can still implement many real-time systems that meet both the timeliness guarantees as well as the security requirements.

C. Simulation Results for Other Schemes

While section IV presented the analysis for one instance of an FP algorithm (non-preemptive) and a scheduling constraint (Half-PF), other combinations are possible as well (enumerated below). We believe that performing similar analyses for all of these combinations, while laborious, is still feasible and builds upon the intuition(s) provided in this paper. We omit these extensive analyses here due to space limitations and intend to work on them as part of future work. In the meantime, we carried out a variety of simulations for these other combinations so that designers of secure real-time systems can gain a better understanding of the effect of the various parameters. In this section we enumerate the results from these simulations.

Given the set of generated tasks from Section VI-A, we use a simulator that schedules task sets using scheduling policies obtained by combinations of basic FP and Non-Preemptive FP with the constraints from Section III-B:

- Preemptive (or vanilla) FP (`FP`): tasks are scheduled by the basic FP scheduling policy. Preemptions are allowed with no FT invocations.
- NonPreemptive FP (`FP Fully Non-Preemptive`): It is the same as `FP` mentioned above, except that no preemptions are allowed but we do allow FT invocations when changing from high to low security levels.
- Preemptive FP with FT (`FP PF` but actually `FP Half-PF`): same as `FP`, however a resource flush task (FT) is executed whenever a higher security level task is being preempted by a task having a lower security but higher priority level.
- Preemptive FP with Resource Flush under certain conditions (`FP CPF`): Preemptions are not allowed when the preempting task has a higher security level than the one currently running; otherwise preemptions are allowed.

Figure 6 presents results for the performance analysis of the "random ordering" method (Definition 6) for the above algorithms. For the following discussion, the execution overhead for each FT invocation, unless otherwise stated, is 5 time units[10].

[9]While the typical schedulability tests for FP put the theoretical upper bound at 69% [17], it is possible for FP to schedule task sets with higher utilizations – *e.g.,* if they are harmonic in nature.

[10]We also saw similar trends for other values of c_{ft} but omit them due to space considerations.

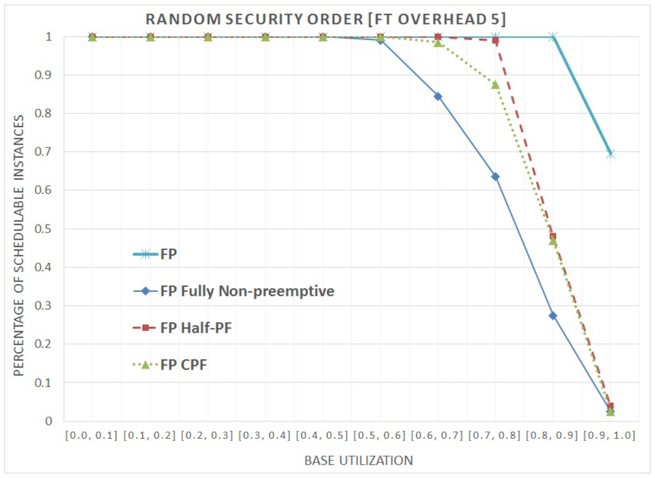


Fig. 6. Simulation-based Results [Random Ordering]

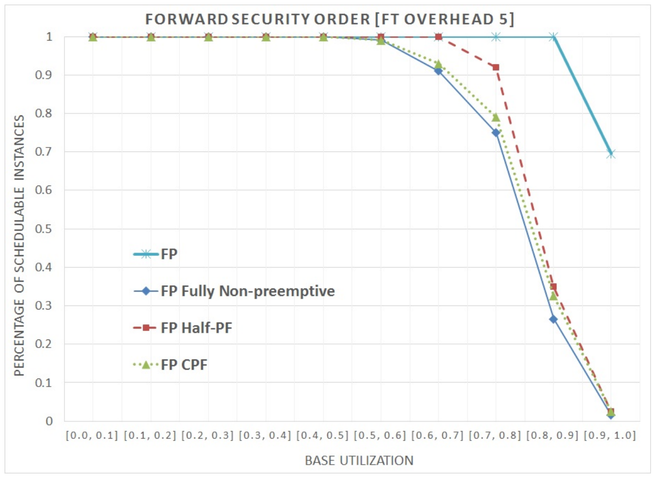


Fig. 7. Simulation-based Results [Forward Ordering]

As before, the vanilla FP performs the best. At the other extreme, we have the `FP Fully Non-Preemptive` algorithm that doesn't allow any preemptions. As expected, it has a tougher time in keeping up with `FP`, especially when the utilization values increase. These are the endpoints that we will use for comparing the remaining constraint-based algorithms.

Figure 6 shows that while `FP Half-PF` performs fairly well at scheduling most task sets, `FP CPF` is also able to match its performance, for the most part. `Half-PF` drops more task sets (as compared to `FP`) – its performance starts to degrade around the 75% utilization mark. It is interesting to note that this algorithm and `FP CPF` are still able to schedule some task sets as the utilization grows, at which point, interestingly, `FP CPF` catches up with `FP Half-PF`. Of course, both of these still perform much better than the `FP Fully Non-Preemptive` algorithm. The chief benefit of all three of these modified FP algorithms is that the overall *security is increased*. Now RTS designers can make choices based on quantifiable information – increased security versus a little drop in real-time utilization.

We conducted similar experiments for the Forward and Backward ordering of security levels (see Figures 7 and 8). As expected, the forward ordering performs worse than the random ordering. The `FP Half-PF` and `FP CPF` algorithms are similar in performance for the forward ordering – they both start dropping out earlier than random (around 65% and 55% respectively) and are closer to the `FP Fully Non-Preemptive` version. Hence, our conjecture that the forward ordering results in bad (potentially worst) behavior is confirmed.

The backward ordering (Figure 8), on the other hand, seems to be the *best* performer of the lot. In fact, its performance matches that of `FP` for the most part, dropping down (albeit a little) only for the really high utilization values. Again, this seems to underscore our claim that a backward ordering of security levels will be the best way to obtain the highest performance, while still guaranteeing the security properties. One interesting observation: for the backward ordering of security levels, `FP CPF` *is transformed into* `FP Half-PF`. The reasoning is simple: remember that CPF disallows preemptions when the higher priority task has a higher security level but allows preemptions in the opposite case. In the backward ordering, a higher priority task *always* has a lower (or equal) security level as the currently executing task – hence, preemption are always allowed. In Figure 8 the two lines are indistinguishable.

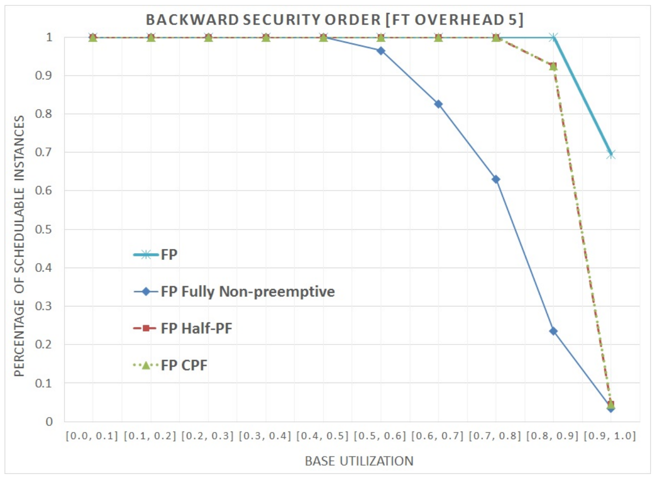


Fig. 8. Simulation-based Results [Backward Ordering]

An important point to note though: for the lower utilization levels (typically below 50%), all of the modified algorithms perform just as well as FP in scheduling the task sets. *Hence, for many real-time systems, these algorithms are able to not just improve the security properties of the systems, but do so without any observable drop in performance.*

We can definitely think of ways to reduce the number of FT invocations based on the following insights: **(1)** Consider the `PF/Half-PF` constraints; in any given task set τ and security ordering S, a particular task τ_i will cause a FT invocation iff it is preempted by a task, $\tau_j | \; pri_j \prec pri_i \;\&\; s_j \succ s_i$; hence all other preemptions will not result in FT executions – depending on the task set characteristics, this could potentially reduce a large number of preemptions; **(2)** `CPF` reduces many preemptions (and FT invocations) by definition; it also ensures that a lower priority job will experience a FT execution only *once* due to higher priority, higher security level tasks, thus reducing the total number of FT invocations; **(3)** All lower priority jobs, with higher security levels than the current job will only increase the number of FT executions for that job by *one* – essentially before it is scheduled to run. Once the job starts executing, it cannot be preempted by lower priority jobs (even with higher security levels); the latter cannot be scheduled as long as the current job has leftover execution.

The above rules show that the actual number of FT invocations due to these scheduling constraints could be reduced, perhaps even more aggressively than the bounds presented in Section IV. Results in Figure 9 also validate this observation.

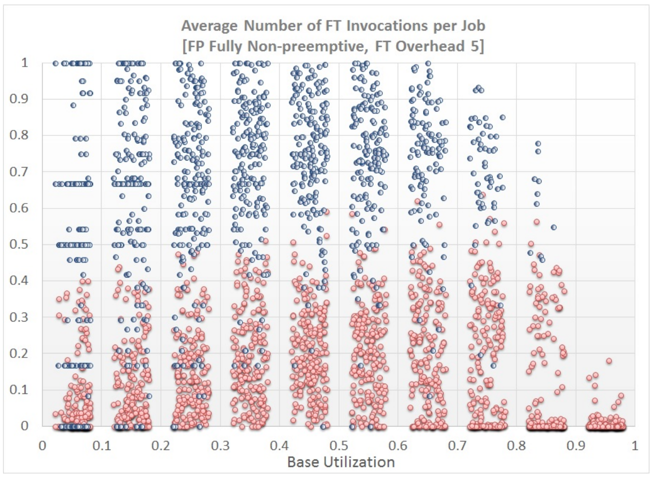


Fig. 9. Avg. #FT per Job for Analysis & Simulations [FP + Half-PF]

For each task set, on the Y-axis, the graph measures the number of FT invocations normalized to the total number of jobs, *i.e.,* the average number of FT executions per job. The X-axis represents the various utilization ranges/bins. This graph shows the results for the `FP Half-PF` algorithm for a random security ordering. The red dots (on the lower part of the graph) show the results we obtained from the simulations while the blue dots (on the upper parts of the graph) show the results for average number of FT invocations per job from the worst-case response time analysis. From this graph, we observe that: *(i)* the total number of FT invocations is typically *much less* than the number of jobs; *(ii)* this is also true as we reach the higher utilization task sets; *(iii)* the graph also shows that, for most tasks sets, the number of simulated FT executions is lower than the values calculated from our worst-case response time analysis. Hence, there is significant scope for reduction in these overheads and for corresponding increases in the schedulability of task sets. Other combinations of FP scheduling algorithms and security constraints also show similar behavior; they are omitted here due to space considerations.

D. Limitations

While we are able to show the value of, and methodology for, transforming security requirements (in this case prevent information leakage) into constraints for real-time scheduling, this is obviously not a silver bullet for solving all security problems. Many security requirements may not be amenable to being cast as scheduling constraints, *e.g.,* communication vulnerabilities. Also performance overheads due to such constraints could inhibit their adoption in many high utilization RTS, though with careful design processes this could be mitigated. The exact constraints depend on the system parameters.

VII. Related Work

There is a large body of work on identification, analysis and mitigation of covert side channels (*e.g.,* [10]–[12], [14], [21]). In particular, Hu [11] assumes a similar security model and mitigation strategy (cache flushing) and discusses how scheduling algorithms can be modified to minimize the number of flushes. However, tasks have no real-time requirements and the scheduler does not support any service guarantee. Hence, in the following we only focus on those works that are relevant to real-time systems. For example, it has been shown that scheduling of (real-time) tasks can be a source of information leakage; Son *et al.* [27] showed that a covert timing channel can be established in the rate-monotonic scheduler. Völp *et al.* [34] discuss unauthorized information flows obtained through altered scheduling behavior (*e.g.,* delayed preemption). They also showed how to modify a fixed-priority scheduler to reduce the effect of such malicious alterations. They also studied the effect of timing channels introduced by real-time resource locking protocols and addressed them by transforming the relevant protocols [33]. In contrast, we do not focus exclusively on timing channels but rather on the approach of transforming security properties into real-time scheduling constraints; we use information leakage as an example to illustrate our techniques. The above techniques are orthogonal to ours (we intend to combine them in the future).

There has been some work on reconciling the addition of security mechanisms into real-time systems: Xie *et al.,* [35] and Lin *et al.* [16] considered periodic task scheduling where each task requires a security service whose overhead varies according to the (quantifiable) level of the service. They propose new schedulers [35] and enhancements to existing schedulers (*viz.,* EDF [16]) to meet real-time requirements while maximizing the level of security achieved. In contrast, we study enhancements to FP to reduce information leakage through shared resources while meeting real-time requirements.

The issue of information leakage in real-time database systems with multi-level security constraints has been considered [1], [28], [29]. Son *et al.* [28] focus on transaction scheduling and concurrency control algorithms to meet both security and real-time requirements. This includes metrics to measure fulfillment of security requirements and a concurrency control algorithm that can trade-off of security and real-time requirements [1]. Son *et al.* [29] resolve the conflict between real-time and security requirements by defining a notion of partial security and trading-off between the two.

There also exists recent work on developing architectural frameworks for solving security problems such as intrusion detection [19], [26], [31], [37], [38], among others. They aim to create hardware/software mechanisms to protect against security vulnerabilities while our work aims at the scheduler level. It is not inconceivable that the two sets of approaches could be combined to make the system more resilient to attacks.

Finally, in the case of our PF and CPF algorithms, the issue of computing the number of FT invocations is related to computing the number of preemptions suffered by a task or group of tasks. Existing work [36] discusses how to compute the exact cost of preemptions for a task under fixed-priority scheduling by accounting for the exact number of times that the task is preempted by higher priority tasks. The fundamental difference compared to this work is: according to our Half-PF constraint we only invoke a FT on a transition from a higher security to a lower security task, *not on every preemption*. Furthermore, a FT must be invoked even if a higher security task is simply followed by a lower security task, *i.e.,* without a need for preemption (as discussed in Section IV, which details the analysis for non-preemptive FP). Hence, the strategy detailed in [36] is not directly applicable. Having said that, that work [36] does present some interesting ideas for extending our analysis and we intend to use it as part of future work.

VIII. Conclusion

In this paper we presented methods for integrating security requirements into real-time scheduling algorithms. We considered the problem of information leakage via shared resources in real-time systems to illustrate our techniques. We proposed the modification of scheduling algorithms to include security-related constraints to mitigate such problems. In particular, we modified fixed priority scheduling algorithms and showed that the proposed modification can mitigate information leakage via shared resources with acceptable performance overheads or/and impact on schedulability. Designers of real-time systems can now perform a proper assessment of the tradeoffs between security requirements and real-time guarantees.

There is scope to extend this work in multiple directions. We intend to continue the rigorous mathematical analysis to include other variants of FP. We will then extend our analysis to other scheduling algorithms such as EDF. Finally, we hope to consider multiple shared resources, instead of just one. We expect that applying constraint based approaches to the case of multiple shared resources will be considerably interesting and more challenging. We will also generalize our system model to include partial ordering among security levels.

References

[1] Q. Ahmed and S. Vrbsky. Maintaining security in firm real-time database systems. In *Computer Security Applications Conference, 1998. Proceedings. 14th Annual*, pages 83–90, 1998.

[2] A. N. Audsley, A. Burns, M. Richardson, and K. Tindell. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, pages 284–292, 1993.

[3] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security*, Aug 2011.

[4] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

[5] European Organisation for Civil Aviation Electronics. *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, Dec 1992.

[6] N. Falliere, L. Murchu, and E. C. (Symantec). W32.stuxnet dossier. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf, 2011.

[7] J. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20. IEEE, 1982.

[8] N. Grumman. RePLACE. http://www.northropgrumman.com/Capabilities/RePLACE/Pages/default.aspx.

[9] N. Grumman. Reverse Engineering for Large Applications. http://www.northropgrumman.com/Capabilities/RELA/Pages/default.aspx.

[10] W.-M. Hu. Reducing timing channels with fuzzy time. In *Research in Security and Privacy, 1991. Proceedings., 1991 IEEE Computer Society Symposium on*, pages 8–20, 1991.

[11] W.-M. Hu. Lattice scheduling and covert channels. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1992.

[12] T. Kim, M. Peinado, and G. Mainar-Ruiz. Stealthmem: system-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 11–11, Berkeley, CA, USA, 2012. USENIX Association.

[13] P. Kocher, R. Lee, G. McGraw, A. Raghunathan, and S. Ravi. Security as a new dimension in embedded system design, 2004.

[14] P. C. Kocher. Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[15] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental security analysis of a modern automobile. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 447 –462, may 2010.

[16] M. Lin, L. Xu, L. Yang, X. Qin, N. Zheng, Z. Wu, and M. Qiu. Static security optimization for real-time systems. *IEEE Transactions on Industrial Informatics*, 5(1), Feb. 2009.

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

[18] J. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[19] S. Mohan, S. Bak, E. Betti, H. Yun, L. Sha, and M. Caccamo. S3A: Secure system simplex architecture for enhanced security and robustness of cyber-physical systems. In *ACM Conference on High Confidence Networked Systems*, 2013.

[20] M.-Y. Nam, R. Pellizzoni, L. Sha, and R. Bradford. Asiist: Application specific i/o integration support tool for real-time bus architecture designs. In *Engineering of Complex Computer Systems, 2009 14th IEEE International Conference on*, pages 11 –22, june 2009.

[21] C. Percival. Cache missing for fun and profit. In *Proceedings of BSDCan*, 2005.

[22] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. In *IEEE Real-Time Systems Symposium*, pages 259–269, Dec. 1988.

[23] D. Reinhardt. Certification criteria for emulation technology in the australian defence force military avionics context. In *Proceedings of the Eleventh Australian Workshop on Safety Critical Systems and Software - Volume 69*, SCS '06, pages 79–92, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[24] K. Sampigethaya, R. Poovendran, and L. Bushnell. Secure Operation, Control, and Maintenance of Future E-Enabled Airplanes. *Proceedings of the IEEE*, 96(12):1992–2007, Dec. 2008.

[25] D. Shepard, J. Bhatti, and T. Humphreys. Drone hack: Spoofing attack demonstration on a civilian unmanned aerial vehicle. *GPS World*, August 2012.

[26] W. Shi, H.-H. S. Lee, L. 'Falk, and M. Ghosh. An integrated framework for dependable and revivable architectures using multicore processors. In *Proceedings of the 33rd annual international symposium on Computer Architecture*, ISCA '06, pages 102–113, 2006.

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

[28] S. Son. Supporting timeliness and security in real-time database systems. In *Real-Time Systems, 1997. Proceedings., Ninth Euromicro Workshop on*, pages 266–273, 1997.

[29] S. Son, C. Chaney, and N. Thomlinson. Partial security policies to support timeliness in secure real-time databases. In *Security and Privacy, 1998. Proceedings. 1998 IEEE Symposium on*, pages 136–147, 1998.

[30] S. Son, R. Mukkamala, and R. David. Integrating security and real-time requirements using covert channel capacity. *Knowledge and Data Engineering, IEEE Transactions on*, 12(6):865 –879, nov/dec 2000.

[31] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XI, pages 85–96, 2004.

[32] H. Teso. Aicraft hacking. In *Fourth Annual HITB Security Conference in Europe*, 2013.

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

[34] M. Völp, C.-J. Hamann, and H. Härtig. Avoiding timing channels in fixed-priority schedulers. In *ACM Symposium on Information, Computer and Communication Security*, pages 44–55, New York, NY, USA, 2008. ACM.

[35] T. Xie and X. Qin. Improving security for periodic tasks in embedded systems through scheduling. *ACM Trans. Embed. Comput. Syst.*, 6(3), July 2007.

[36] P. M. Yomsi and Y. Sorel. Extending rate monotonic analysis with exact cost of preemptions for hard real-time systems. In *Euromicro Conference on Real-Time Systems (ECRTS), 2007 19th IEEE*, pages 280–290, 2007.

[37] 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 Real-Time Embedded Technology and Applications Symposium*, 2013.

[38] C. Zimmer, B. Bhatt, F. Mueller, and S. Mohan. Time-based intrusion detection in cyber-physical systems. In *International Conference on Cyber-Physical Systems*, 2010.