# A Generalized Model for Preventing Information Leakage in Hard Real-Time Systems

Rodolfo Pellizzoni*, Neda Paryab*, Man-Ki Yoon†, Stanley Bak‡, Sibin Mohan†, and Rakesh B. Bobba§
*University of Waterloo, Ontario, Canada. {rodolfo.pellizzoni, n2paryab}@uwaterloo.ca
†University of Illinois at Urbana-Champaign, Urbana, IL. {mkyoon, sibin}@illinois.edu
‡United States Air Force Research Lab, Rome, NY. stanleybak@gmail.com
§Oregon State University, Corvallis, OR. rakesh.bobba@oregonstate.edu

***Abstract*—Traditionally real-time systems and security have been considered as separate domains. Recent attacks on various systems with real-time properties have shown the need for a re-design of such systems to include security as a first class principle. In this paper, we propose a general model for capturing security constraints between tasks in a real-time system. This model is then used in conjunction with real-time scheduling algorithms to prevent the leakage of information via storage channels on implicitly shared resources. We expand upon a mechanism to enforce these constraints *viz.,* cleaning up of shared resource state, and provide schedulability conditions based on fixed priority scheduling with both preemptive and non-preemptive tasks. We perform extensive evaluations, both theoretical and experimental, the latter on a hardware-in-the-loop simulator of an unmanned aerial vehicle (UAV) that executes on a demonstration platform.**

## I. Introduction

Real-Time Systems (RTS) are used to monitor and control a variety of systems across multiple domains. These include aircraft, space vehicles, submarines, automobiles, power plants and other critical infrastructures, industrial manufacturing systems, *etc.* The design of such systems is increasing in complexity by the day, thus opening up new surfaces for attack by malicious adversaries – a situation that was hitherto unknown. This was because, until recently, RTS *(a)* used specialized protocols, *(b)* were physically isolated from the external world and *(c)* executed on specialized hardware. Such systems are increasingly being connected to each other, oftentimes using unsecured networks such as the Internet. Moreover, attackers are increasing in sophistication and are able to bridge air gaps in industrial control systems [7], perform malicious code injection into the telematics units of modern automobiles [4], [13], and demonstrate potential vulnerabilities in avionics systems [28] and attacks on UAVs [23].

Given the additional constraints on the operation of real-time systems, vulnerabilities and the impact of exploiting them differ from those of traditional enterprise systems. A successful attack could destabilize the system resulting in harm to humans, the environment and/or the system. Attacks could range from the leakage of critical data [25] to hostile actions such as an adversary taking control of the system, say due to the lack of authentication [4], [13], [28]. Many of the aforementioned attacks succeeded because such systems were not designed with security in mind. This is the issue we intend to tackle in our work – a holistic *integration of security into the design of real-time resource management algorithms*. This is particularly important since simply tacking on 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.

We would like to thank Yassir Rizwan and Thomas Reidemeister for providing the demonstrator control system. This work is supported in part by a grant from ONR (N00014-13-1-0707 and N00014-12-1-0046) and by CMC Microsystems. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of sponsors.

In early work [18] we showed a simple proof of concept – how to prevent the leakage of information through shared caches by using a flushing mechanism to cleanup the state of the shared cache. The study was based on a simple, restrictive security model and was aimed at studying initial tradeoffs between security requirements and real-time guarantees.

In this paper we relax many of the restrictions in our earlier work and propose a *new, more general model* (Section II) to capture security constraints between tasks in a real-time system. It is fairly well understood that the use of shared resources can lead to information leakage without the need for explicit communication between tasks [12], [21], [35]. Hence, we focus on the problem of information leakage and propose a constraint that we name *noleak* to capture whether *unintended information sharing between a pair of tasks must be forbidden*. We also relax the constraints on the types of real-time resource management algorithms that are analyzed (for instance, preemptive vs. non-preemptive). Finally, we discuss implementation details on a *realistic platform* (FreeRTOS running on an ARM CPU) – we demonstrate both the model and the methods presented in this paper on a *hardware-in-the-loop simulator* that we developed for an *unmanned aerial vehicle* (UAV) platform.

In summary, our high level contributions are:

1) showing how to capture security constraints as relationships between tasks and a *new vendor oriented security model* for information leakage (Section II);
2) a mechanism that can be used to enforce such constraints, *viz.,* the *cleaning up of shared resource state* (Section III);
3) *algorithms* (both exponential exact enumerations and polynomial time approximations) that bound the effect of such mechanisms under fixed priority scheduling (Section IV);
4) discuss the effects of the preemptivity of the tasks on the overhead of the mechanism and also develop an algorithm that assigns the notion of preemptability to real-time tasks in an *optimal manner* (Section V);
5) demonstrate our developed mechanisms and schedulability analysis on a realistic application case study (for UAVs), along with an extensive evaluation based on synthetic tasks in Section VI.

## II. System and Security Model

We consider the fixed priority scheduling of a set $\Gamma = \{\tau_1, \ldots, \tau_N\}$ of $N$ sporadic real-time tasks on a uniprocessor.

Each task $\tau_i$ is characterized by a tuple $\{p_i, c_i, preempt_i\}$, where $p_i$ is the task's period or minimum inter-arrival time, $c_i$ is its worst-case execution time and $preempt_i$ determines the task preemptability. We assume implicit deadlines, *i.e.,* the relative deadline of task $\tau_i$ is equal to $p_i$. If $preempt_i = T$, then the task can be preempted by higher priority tasks; if instead $preempt_i = F$, a job of $\tau_i$ always run to completion once started. Tasks are assigned distinct priorities; without loss of generality, we assume that the priority of task $\tau_j$ is higher than the priority of task $\tau_i$ if and only if $j < i$. For ease of notation, let $hp_i = \{\tau_j | 1 \leq j < i\}$ be the set of all tasks with priorities higher than $\tau_i$ and let $hep_i = \{\tau_j | 1 \leq j \leq i\}$ be the set of higher priority tasks including $\tau_i$; similarly, we define the set of lower priority tasks $lp_i = \{\tau_j | j > i\}$. Finally, we assume that time is an integral quantity measured in multiples of the system tick. For simplicity we do not discuss the effects of shared resources; the analysis could be extended to include confidentiality-preserving real-time locking protocols [29].

Tasks in the system are subject to security constraints. In particular, we consider the issue of information leakage in systems with tasks at different security or protection levels. We model such security constraint by introducing a binary "no leak" relation between any two different tasks $\tau_i$ and $\tau_j$: if $noleak(\tau_i, \tau_j) = T$ (true), then we require that information leakage from $\tau_i$ to $\tau_j$ is prevented; otherwise if $noleak(\tau_i, \tau_j) = F$, no such constraints need to be enforced between $\tau_i$ and $\tau_j$. Note that we do not impose any other property on the $noleak$ relation; in particular, we do not require properties of symmetry (i.e., it might hold $noleak(\tau_i, \tau_j) = T$ but $noleak(\tau_j, \tau_i) = F$) or transitivity. We also do not assume any special relation between the real-time properties of a task and the $noleak$ requirements.

We argue that the described scenario (*i.e.,* tasks with different protection levels) can arise in various complex real-time systems where applications developed by different vendors are integrated together on the same computing platform. Examples include avionics systems designed according to the DO-178B standard [6], as well as emulation and integration systems designed for the porting of legacy applications, such as the "RePLACE" system from Northrup Grumman [8], [22].

## A. Example System: Avionics Demonstrator

In order to motivate and evaluate the presented research, we use the example of the Electronic Control Unit (ECU) for an avionics system. This demonstrative system, built and implemented at the University of Waterloo, runs many of the same types of tasks which could be expected to run on an Unmanned Aerial Vehicle (UAV) surveillance system (Figure 1). The ECU communicates locally with the inertial sensors, GPS localization system and actuators ("UAV" in the figure), as well as a camera subsystem. The ECU also uses off-board communication to exchange information with a base station. We assume that three parties are involved in building the ECU system, Vendor 1, Vendor 2, and the Integrator. Each party is responsible for a different ECU subsystem, which in turn comprises different real-time tasks. One or more tasks of each party have some degree of protected data.

*Vendor 1* is responsible for the *image subsystem*. The I/O Operation Task mimics the behavior of a camera driver; to perform repeatable experiments, our system simply places a fixed set of images into a Memory File System (MFS) and extracts them in order. Since conceptually this task manages input state and not image data, it does not need to be protected.

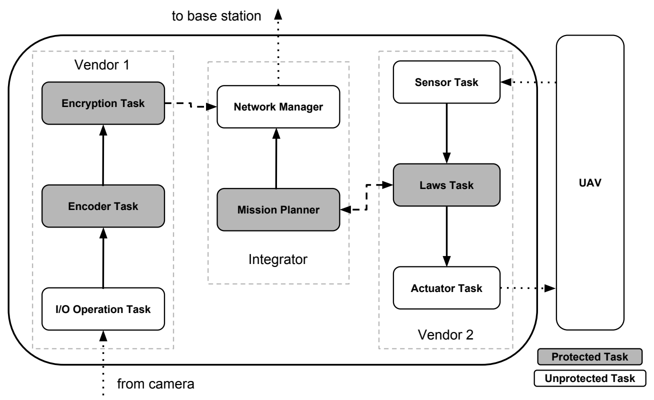


Fig. 1: Case Study Outline. A dashed arrow shows communication between tasks in different subsystem, a solid arrow is for communication within the same subsystem, and a dotted arrow is for external communication

The Encoder Task is realized as a JPEG compressor. Encryption Task uses the AES cipher using a protected, secret key. The encrypted image is then passed to the network manager in the Integrator subsystem.

*Vendor 2* is responsible for the *control subsystem*. The Sensor Task receives and parses incoming sensor data for the other tasks. The Laws Task computes the control output to move the UAV towards a way-point determined by the Mission Planner in the Integrator subsystem. Finally, the Actuator Task prepares the actual output commands and send them to physical actuators.

Finally, the *Integrator* is responsible for connecting the two previous subsystems together and performing mission control. The Mission Planner Task communicates with the Laws Task to determine the current position of the UAV and move it between a set of fixed way-points. The Network Manager sends encrypted data coming from the Mission Planner and the Encryption Task to the base station.

| Task Name | Worst Case Timing(ms) | Period(ms) |
|---|---|---|
| Software Control Tasks | 2 | 20 |
| Mission Planner | 0.002 | 100 |
| Encryption | 3 | 42 |
| Image Encoding | 18 | 42 |
| Image I/O | 1.46 | 42 |
| Network Manager | 0.03 | 10 |

TABLE I: case study timing parameters

Table I summarizes the key timing parameters of the implemented systems; we provide measured worst-case execution times based on a cold cache initial configuration. We report the cumulative execution time for all control tasks because they run at the same frequency. Note that communication in the image and integrator subsystem can use non-blocking buffers to avoid long blocking times.

## B. Security Model

The proposed binary $noleak$ relation between any two tasks can be generalized to implement a range of security constraints. In this work we focus on unintended information flow (or information leakage) between tasks of different vendors through the use of shared resources. In particular, we consider a vendor oriented security model where information leakage from a protected or sensitive task of one vendor to any task of a different vendor is considered undesirable while no such

| $noleak$ | | to | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Sens. | Laws | Act. | MP | Net. | AES | JPEG | I/O |
| from | Sens. | - | F | F | F | F | F | F | F |
| | Laws | F | - | F | T | T | T | T | T |
| | Act. | F | F | - | F | F | F | F | F |
| | MP | T | T | T | - | F | T | T | T |
| | Net. | F | F | F | F | - | F | F | F |
| | AES | T | T | T | T | T | - | F | F |
| | JPEG | T | T | T | T | T | F | - | F |
| | I/O | F | F | F | F | F | F | F | - |

TABLE II: $noleak$ relations of the case study

constraints are placed on tasks within a given vendor's subsystem. Such a model is useful in many scenarios. For example, in our avionics demonstrator scenario, images captured by the camera could have a higher security classification and it may be undesirable for Vendor 2 to gain unintended information about them even if the vendor is trusted with controlling the UAV. Similarly, the control laws from Vendor 2 may contain a proprietary algorithm and Vendor 2 may not want other vendors to gain unintended knowledge about the algorithm.

The $noleak$ relations between the tasks using the vendor oriented security model for the avionics demonstrator case study are listed in Table II. While the proposed vendor oriented security model has the flavor of multi-level security models (*e.g.,* [2]), it is quite different from such models. In fact the proposed model is quite relaxed relative to traditional multi-level security (MLS) security models such as the well-known Bell-LaPadula [2] model that aim to prevent information flow from a higher security level to a lower security level even within the same compartment (in this case vendor). In contrast, note that no constraints on leakage are placed between tasks from the same vendor in the proposed model. Similarly, no constraints on leakage are placed between unprotected tasks (not security sensitive) even if they are from different vendors. However, it is important to note that one could capture stricter security models using the proposed binary $noleak$ relation when the system at hand warrants such a stricter model.

## III. Scheduling and Security

Both the software and hardware execution platform influence tasks' ability to leak information. We assume that the employed platform already provides mechanisms such as virtual memory to prevent or control explicit communication between tasks. However, modern systems include a variety of resources, such as caches, DRAM and I/O interconnections, that are highly 'stateful': hence, by changing the state of the resource, the behavior of one task can affect the timing of another, later executed task. Timing attacks based on cache state are well documented in the literature [12], [21], [35]; for example, the authors of [35] demonstrated that a cryptographic key can be extracted by analyzing the cache contents of virtual machines. Other resources could similarly be used as covert channels between tasks. For example, DRAM controllers typically implement an open row mechanism that behaves in a manner that is similar to a cache [31]. It has been shown that I/O buses can carry traffic belonging to one task even after a new task has been scheduled [19].

TABLE III: Example task set and "no-leak" relation between tasks

| | preempt | $I_j$ |
|---|---|---|
| $\tau_1$ | T | 3 |
| $\tau_2$ | F | 2 |
| $\tau_3$ | T | 1 |

| $noleak$ | | to | | |
|---|---|---|---|---|
| | | $\tau_1$ | $\tau_2$ | $\tau_3$ |
| from | $\tau_1$ | - | T | F |
| | $\tau_2$ | T | - | T |
| | $\tau_3$ | T | F | - |

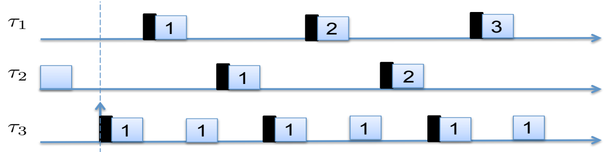


Fig. 2: Example task set: worst-case schedule. Vertical bars represent FTs. Numbers represent job indexes (i.e., $\tau_3$ executes a single job in the busy interval).

In general, there are multiple mechanisms that can be employed to mitigate or outright avoid information leakage due to a shared resource. For example, hardware caches can be partitioned using either hardware or software-based mechanisms [16]. However, such mechanisms are likely to be specific to a given resource, and they might furthermore negatively impact the execution times of tasks. Instead, in this paper we propose to use a general "flushing" mechanism: whenever we detect that a task might undesirably leak information to another task, we execute a synthetic 'Flush Task' (FT) function that resets the state of all needed resources. For example, cache content can be flushed and invalidated, and DRAM rows can be closed. In detail, based on the $noleak$ relation defined in Section II, we implement the following *No-Leak Flush (NLF)* mechanism:

**Definition 1** (NLF Mechanism)**.** *Let* $\Gamma'$ *be the set of tasks executed since the last FT. Then if there* $\exists \tau_j \in \Gamma', noleak(\tau_j, \tau_i) = T$*, a FT must be executed before scheduling task* $\tau_i$*.*

Note that our $noleak$ model does not make any assumption on which portion of a task reads/writes sensitive data; hence, in the situation stated by Definition 1, we need to flush even if the job of $\tau_j$ did not finish executing. Let $c_{ft}$ be the worst-case execution time of the FT. Note that we assume that if a FT is required, it is executed as part of $\tau_i$, *i.e.,* the execution time of $\tau_i$ is effectively increased to $c_i + c_{ft}$. Once the NLF mechanism has been defined, the schedulability of task set $\Gamma$ can be guaranteed by computing a safe upper bound $N_{ft}$ to the number of FT required in the busy interval of any task $\tau_i \in \Gamma$; we show how to do so in Section IV.

An example schedule under NLF is shown in Figure 2 for the task set in Table III, where $I_j$ represents the number of instances of $\tau_j$ in the depicted busy interval[1] starting with the release of $\tau_3$. Note that a FT is required before $\tau_3$ is scheduled because $\tau_2$ executed before the start of the busy interval and $noleak(\tau_2, \tau_3) = T$. Similarly, a FT is required before $\tau_1$ can preempt $\tau_3$ because $noleak(\tau_3, \tau_1) = T$; however, no FT is required when execution returns to $\tau_3$ since $noleak(\tau_1, \tau_3) = F$. Finally, a FT is required before executing $\tau_2$ even if $noleak(\tau_3, \tau_2) = F$ because $noleak(\tau_1, \tau_2) = T$ and no FT is run between $\tau_1$ and $\tau_3$.

It is interesting to note that task preemptivity can have a significant effect on the FT number. As an example, Figure

[1]Note that since Definition 1 depends only on the sequence of task executions, for simplicity throughout all figures we do not report or draw execution times to scale.

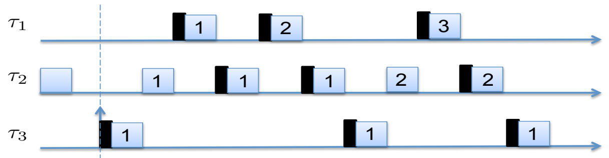


Fig. 3: Example task set: preemptive worst-case schedule.

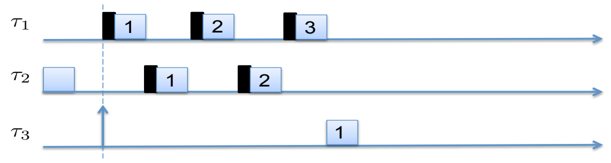


Fig. 4: Example task set: non-preemptive worst-case schedule.

3 shows the schedule for the same task set as in Table III, except that all tasks are preemptive. In this case, the number of FT is 9 rather than 8. Figure 4 shows the same example when all tasks are non-preemptive, resulting in 5 FT. As we show in Sections IV-A, IV-B, these are in fact exact bounds on the worst-case number of FT suffered by the example task set. In general, task preemption creates additional context switches that can increase the number of FT; on the other hand, executing a task non-preemptively creates blocking time for higher-priority tasks. Based on such observation, in Section IV we first provide a schedulability analysis for task set $\Gamma$ assuming that the preemptability $preempt_i$ of each task $\tau_i$ is known; then, using the derived schedulability analysis, in Section V we show how to optimally assign $preempt_i$.

## IV. Schedulability Analysis

Let $\tau_i$ be a task under analysis. Our schedulability analysis relies on determining an upper bound to the number $N_{ft}$ of FT for $\tau_i$. In the remaining of the paper, we shall consider bounds on $N_{ft}$ based only on the $noleak$ relation and the number of higher priority jobs that interfere with $\tau_i$, i.e., we make no assumption on the exact arrival time of interfering jobs. While this could lead to a potentially pessimistic bound on $N_{ft}$, it nevertheless allows us to decouple the determination of the worst-case number of $FT$, which depends on the job execution order, from the determination of the number of interfering jobs of higher priority tasks, which is based on the critical instant arrival pattern. Therefore, we shall write $N_{ft}(noleak, \{I_j | \tau_j \in hp_i\})$ to denote that $N_{ft}$ is a function of $noleak$ and the number $I_j$ of jobs of higher priority tasks $\tau_j$ that interfere with $\tau_i$.

We use the analysis strategy in [3] for fixed-priority scheduling to determine the schedulability of $\tau_i$, except that we add a term $N_{ft}(noleak, \{I_j | \tau_j \in hp_i\}) \cdot c_{ft}$ to account for the overall FT time. More in detail, $\tau_i$ is schedulable iff $\exists t, 0 < t \leq p_i$, such that:

$$B_i + N_{ft}(noleak, \{I_j | \tau_j \in hp_i\}) \cdot c_{ft} + \sum_{\forall \tau_j \in hp_i} (I_j \cdot c_j) + c_i \leq t, \tag{1}$$

where $B_i$ represents the maximum blocking time induced by lower priority tasks and their FT. If $\tau_i$ is non-preemptive, then the number of interfering jobs $I_j$ of $\tau_j$ is computed as:

$$I_j = \left\lfloor \frac{t - c_i}{p_j} + 1 \right\rfloor, \tag{2}$$

while if $\tau_i$ is preemptive it is computed as:

$$I_j = \left\lceil \frac{t}{p_j} \right\rceil. \tag{3}$$

Furthermore, the maximum blocking time $B_i$ is:

$$B_i = \max_{\forall \tau_j \in lp_i \wedge preempt_j = F} \bar{c_j} - 1, \tag{4}$$

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

Note that in practice it is sufficient to test Equation 1 on all points before a discontinuity in $I_j$, which are $S_i = \{r \cdot p_j | \tau_j \in hp_i \wedge 1 \leq r \leq \lfloor p_i / p_j \rfloor\}$ if $\tau_i$ is preemptive, and $S_i = \{r \cdot p_j + c_i - 1 | \tau_j \in hp_i \wedge 1 \leq r \leq \lfloor p_i / p_j \rfloor\}$ otherwise; [3] shows how to further reduce the number of required testing points. Furthermore, from Equation 1 it follows immediately that we can compute the *slack* $\Delta_i$ for $\tau_i$ as:

$$\Delta_i = \max_{t \in S_i} \Big( t - \big( B_i + N_{ft}(noleak, \{I_j | \tau_j \in hp_i\}) \cdot c_{ft} + \sum_{\forall \tau_j \in hp_i} (I_j \cdot c_j) + c_i \big) \Big), \tag{5}$$

where $\tau_i$ is schedulable iff the slack is non-negative, in which case $\Delta_i$ represents the additional amount of time that $\tau_i$ can take to complete while remaining schedulable; we will use this value in Section V.

A trivial bound on $N_{ft}(noleak, \{I_j | \tau_j \in hp_i\})$ can be obtained as the number of context-switches, including the one at the beginning of the busy interval. If a task $\tau_j$ can preempt another task $\tau_k$ in the busy interval, $\tau_j$ accounts for two context-switches (when $\tau_j$ starts by preempting $\tau_k$, and when execution returns to $\tau_k$); otherwise, $\tau_j$ accounts for only one context-switch (when it starts). Hence, let $\Gamma_i^2 = \{\tau_j \in hp_i | \exists \tau_k \in hep_i : k > j \wedge preempt_k = T)\}$ be the set of all higher priority tasks that can preempt another task in the busy interval for $\tau_i$ (i.e., either $\tau_i$ or another task with priority lower than $k$ but higher than $\tau_i$ must be preemptive), and let $\Gamma_i^1 = hp_i \setminus \Gamma_i^2$ be the set of all other tasks in $hp_i$. Then the number of context-switches $CS_i$ is upper bounded as follows:

$$CS_i = \sum_{\forall \tau_j \in \Gamma_i^1} I_j + \sum_{\forall \tau_j \in \Gamma_i^2} 2 \cdot I_j + 1. \tag{6}$$

In the following Section (IV-A) we first show how to compute an exact bound (assuming no knowledge of tasks' arrival times) $N_{ft}(noleak, \{I_j | \tau_j \in hp_i\})$ based on a SMT formulation. Since the complexity of the algorithm is exponential, in Section IV-B we then show how to derive a safe bound based on a min-cost flow graph problem that can be solved in polynomial time in the number of tasks $N$. We will then show through simulations in Section VI-B that the graph bound is a good approximation of the exact bound.

### A. Exact FT Bound

We created an SMT formulation of the exact FT bound problem using the Z3 SMT solver [5]. The input to this problem is the $noleak$ matrix, the number of jobs for each higher priority task, and whether each higher priority task is preemptive or not. The problem is formulated by creating a set of variables for each step, where a step is a job start or a job end. The set of variables at each step includes (1) an integer for the number of flushes so far, (2) a vector of booleans whether each task is running, (3) a vector of booleans for whether each task is in memory, (4) a vector of integers for the number of jobs remaining to be started for each task, and (5) the transition type taken to get to the next step (encoded as an integer). At each step, a transition occurs, subject to precondition and postcondition constraints. Possible transitions include, for each task $\tau_i$, the task can start, for each task $\tau_i$ and lower priority task $\tau_j$ (including an idle task), $\tau_i$ can end and $\tau_j$ resumes and an FT occurs, or $\tau_i$ can end and $\tau_j$ resumes and no FT occurs.

Conditions are then placed on both the precondition at each step and the postcondition at the subsequent step. For example, a task $\tau_i$ is allowed to start only if the jobs left to start for that task is greater than zero (precondition), and then in the subsequent step the number of jobs left to start will be one less than in the previous step (postcondition). Another example is that task $\tau_i$ can end and task $\tau_j$ resume with a FT only if $\tau_i$ is running, and no higher priority tasks are running, and $\tau_j$ is the next highest priority task, and $\tau_j$ is running, and there is some task in memory for which $\tau_j$ requires a flush and so on. Some of the postconditions in this case would be that only task $\tau_j$ is in memory (because of the FT that has occurred), and that the number of flushes so far is one greater than the in the previous step.

The SMT solver looks for a model where, at the last step, the number of flushes so far is above a threshold. This threshold is iteratively increased until the maximum is exceeded, after which the constraints are unsatisfiable. The last satisfiable model obtained gives an ordering of events that produces that number of flushes $N_{ft}$.

An upper bound on the number of schedules Z3 checks can be given by noticing that at each scheduling event (job start or end, *i.e.,* a context switch), at most a single event (such as a task starting) is possible for each task in the busy interval. This means that runtime of the exact bound approach for a specific task under analysis $\tau_i$ is upper-bounded by $\mathcal{O}\big((CS_i)^{|hep_i|}\big)$, where $CS_i$ is the number of context-switches and $|hep_i|$ is the number of higher or equal priority tasks. While this bound is exponential, solutions for task sets consisting of six or seven higher priority tasks could typically be found in a few minutes. We used this ideal bound to compare with the graph bound in order to evaluate the pessimism of the faster method.

## B. Approximated FT Bound

We now show how to compute a fast bound to $N_{ft}(noleak, \{I_j | \tau_j \in hp_i\})$ using a min-cost flow graph formulation, as defined below.

**Definition 2** (Min-cost flow problem)**.** *Let $(V, E)$ be a flow network, i.e., a directed graph where $V$ is the set of vertices and $E$ the set of directed edges of the form $e = (v \to v', u, a)$, where $v, v' \in V$, $u > 0$ is the capacity of the edge and $a$ is its cost. Let $source \in V$ be the source vertex, producing an amount of flow $F > 0$, and let $sink \in V$ be the sink vertex, consuming an amount of flow equal to $F$. Finally, let $f(e)$ be the flow on edge $e$. Then the min-cost flow problem is to minimize the total cost $\bar{A}$ of the flow:*

$$\bar{A} = \sum_{\forall e=(v \to v', u, a) \in E} f(e) \cdot a, \tag{7}$$

*subject to the constraints:*

- *capacity constraint: $\forall e = (v \to v', u, a) \in E : f(e) \le u$;*
- *flow conservation: $\forall v \in V : \sum_{\forall e=(v \to v', u, a) \in E} f(e) - \sum_{\forall e=(v' \to v, u, a) \in E} f(e) = k$, where $k = F$ for source, $k = -F$ for sink, and $k = 0$ for all other vertices.*

We shall say that a flow assignment $f$ for $(V, E)$ is a *valid flow* if it satisfies all capacity and flow conservation constraints.

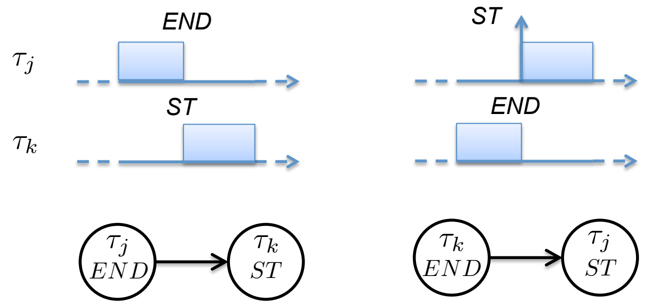


Fig. 5: Flow Graph Intuition: Context-Switch at Job End. One unit of flow is exchanged between a job that finishes (END) and one that starts (ST) executing. Note any priority relationship is valid since a higher priority task $\tau_j$ could arrive at the same time $\tau_k$ finishes (right side of figure).

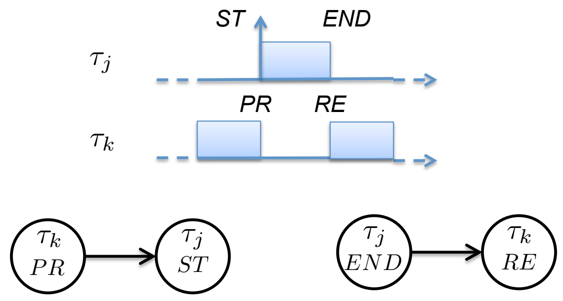


Fig. 6: Flow Graph Intuition: Context-Switch at Preemption and Resumption. One unit of flow is exchanged between a preempted job (PR) of task $\tau_k$ and a starting job (ST) of higher priority task $\tau_j$. In this example, when the job of $\tau_j$ ends (END), execution is returned to $\tau_k$ (RE).

Before formally detailing how we construct the flow graph in Definition 3, we provide the intuition behind our method. The key idea is to encode the sequence of jobs executing during the busy interval of $\tau_i$ as a chain of vertices exchanging flows between them. The exchange of one unit of flow on an edge between a vertex and the next one in the chain represents a context-switch between jobs of the tasks represented by those vertices. We assign a cost of $-1$ to edges representing context-switches that result in the execution of a FT and we compute the minimum cost over any valid flow; we can show that if the resulting (negative) minimum cost is $\bar{A}$, then $-\bar{A}$ represents an upper bound to $N_{ft}(noleak, \{I_j | \tau_j \in hp_i\})$.

Note that a job could be context-switched out in two different situations: *(a)* the job has finished executing or *(b)* the job is preempted by a higher priority task. Similarly, a job could be context-switched into in two different situations: *(c)* the job starts executing or *(d)* the job resumes execution after having been preempted. To encode each situation, we associate four different vertices to each task $\tau_j$: $\tau_j.END$ (job *end*ing); $\tau_j.PR$ (job *pr*eempted); $\tau_j.ST$ (job *st*arting); and $\tau_j.RE$ (job *re*suming). We can then recognize three types of context-switch and associated flow exchanges between vertices:

- A job of a task $\tau_j$ ends and a job of a task $\tau_k$ starts. Figure 5 shows this as an exchange of flow between $\tau_j.END$ and $\tau_k.ST$. Note: any priority relationship and any preemptivity is possible for tasks $\tau_j$ and $\tau_k$.
- A job of a lower priority task $\tau_k$ is preempted by a job of a higher priority task $\tau_j$ that starts execution. The situation is depicted on the left side of Figure 6 as an exchange of flow between $\tau_k.PR$ and $\tau_j.ST$. Note that it must hold $j < k$, and furthermore $preempt_k = T$.

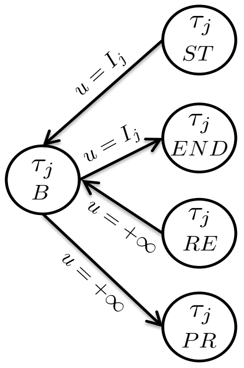


Fig. 7: Vertex Group: Preemptive task $\tau_j \in hp_i$

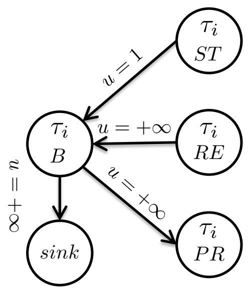


Fig. 8: Vertex Group: Preemptive task under analysis $\tau_i$. The sink consumes $F = 1$ units of flow.

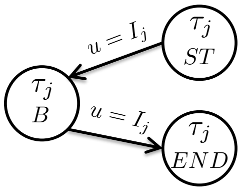


Fig. 9: Vertex Group: Non-preemptive task $\tau_j \in hp_i$

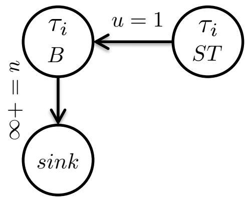


Fig. 10: Vertex Group: Non-preemptive task under analysis $\tau_i$. The sink consumes $F = 1$ units of flow.

- A job of a higher priority task $\tau_j$ ends, and a preempted lower priority task $\tau_k$ resumes executing. This situation is depicted on the right side of Figure 6 as an exchange of flow between $\tau_j.END$ and $\tau_k.RE$. Similarly to the previous case, it must hold $j < k$ and $preempt_k = T$.

Note that each job must start and end once in the busy interval and, furthermore, the number of times the job is preempted must be equal to the number of times the job is resumed. Hence, for a task $\tau_j$, the incoming flow to vertices $\tau_j.ST$ and $\tau_j.RE$ must be equal to the outgoing flow from vertices $\tau_j.END$ and $\tau_j.PR$. We can enforce such constraints by adding a vertex $\tau_j.B$ (*b*alance) that maintains the flow conservation. The resulting *vertex group* for a task $\tau_j \in hp_i$ is shown in Figures 7 and 9, where all edges between vertices in the group have a cost $a = 0$. Note that vertices $\tau_j.ST \to \tau_j.B$ and $\tau_j.B \to \tau_j.END$ have a capacity of $I_j$ to enforce the fact that the task cannot execute more than $I_j$ jobs in the busy interval. The vertex group for task $\tau_i$ is represented in Figures 8 and 10. In this case we omit $\tau_i.END$ since we do not need to consider any FT after the task under analysis ends; instead, the $sink$ drains $F = 1$ unit of flow from $\tau_i.B$ to represent the end of the busy interval. Similarly, we add a $source$ node to the graph that injects one unit of flow to represent the context-switch at the beginning of the busy interval.

Finally, we add edges between vertices of type $END, ST, PR$ and $RE$ of any two tasks $\tau_j, \tau_k \in hep_i$ based on the tasks' priorities and preemptivity, as in Figures 5, 6. We assign such edges a cost $a = -1$ if $noleak(\tau_j, \tau_k) = T$, and $a = 0$ otherwise. Note that as discussed in the examples in Section III, before the busy interval starts, a job of any task could have executed last. Hence, when considering edges from the $source$ node to a task $\tau_j$, we need to assign a cost $a = -1$ if there exists any task $\tau_k$ such that $noleak(\tau_k, \tau_j) = T$. Finally, we set the capacity constraint for all such vertices to $+\infty$ since the constraint on the number of jobs $I_j$ executed in the busy interval is already enforced by capacities on the edges in the vertex group.

We can now define the graph. For ease of notation, we define $I_i = 1$ to represent the fact that only one job of the task under analysis appears in the busy interval.

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

1) *a* source *and a* sink *which produce/consume an amount of flow* $F = 1$*;*
2) *vertices* $\tau_i.B, \tau_i.ST$*;*
3) *for each task* $\tau_j \in hp_i$*, vertices* $\tau_j.B, \tau_j.ST, \tau_j.END$*;*
4) *for each preemptive task* $\tau_j \in hep_i$*, vertices* $\tau_j.RE, \tau_j.PR$*;*

*and the following set of directed edges E:*

1) *for each task* $\tau_j \in hep_i$*, the following edges (if the corresponding vertices exist):* $(\tau_j.ST \to \tau_j.B, I_j, 0), (\tau_j.B \to \tau_j.END, I_j, 0), (\tau_j.RE \to \tau_j.B, +\infty, 0), (\tau_j.B \to \tau_j.PR, +\infty, 0)$*;*
2) *edge* $(\tau_i.B \to sink, +\infty, 0)$*;*
3) *for each task* $\tau_j \in hep_i$*, an edge* $(source \to \tau_j.ST, +\infty, a)$*, where* $a = -1$ *if there exists* $\tau_k \in \Gamma, noleak(\tau_k, \tau_j) = T$*, or* $a = 0$ *otherwise;*
4) *for each pair of tasks* $\tau_j \in hp_i, \tau_k \in hep_i$*,* $j \neq k$*, an edge* $(\tau_j.END \to \tau_k.ST, +\infty, a)$*, where* $a = -1$ *if* $noleak(\tau_j, \tau_k) = T$*, or* $a = 0$ *otherwise;*
5) *for each preemptive task* $\tau_k \in hep_i$ *and each task* $\tau_j \in hp_i$ *such that* $j < k$*, an edge* $(\tau_k.PR \to \tau_j.ST, +\infty, a)$*, where* $a = -1$ *if* $noleak(\tau_k, \tau_j) = T$*, or* $a = 0$ *otherwise;*
6) *for each task* $\tau_j \in hp_i$ *and each preemptive task* $\tau_k \in hep_i$ *such that* $j < k$*, an edge* $(\tau_j.END \to \tau_k.RE, +\infty, a)$*, where* $a = -1$ *if* $noleak(\tau_j, \tau_k) = T$*, or* $a = 0$ *otherwise.*

Note that in Definition 3, edges of Types 1-2 are edges in the vertex groups shown in Figures 7-10, while edges of Types 3-6 are edges between vertex groups representing context-switches.

Figure 11 shows a complete example graph for the task set in Table III, where $\tau_i = \tau_3$ is the task under analysis (note that flow assignments $\tilde{f}$ and $\hat{f}$ are used in the upcoming Theorem 1 to illustrate the proof). We use dashed edges to represent context-switches where $noleak = T$ and dotted edges for context-switches where $noleak = F$. Based on the discussed context-switch rules, the edges between vertex groups can be constructed as follows:

- an edge from *source* to every $ST$ (Type 3 in Definition 3);
- an edge from every $END$ to every $ST$ (Type 4), to represent context-switches between an ending and a starting job;
- an edge from every $PR$ to every $ST$ of a higher priority task (Type 5; $\tau_3$ to $\tau_1$ and $\tau_2$; $\tau_2$ has no $PR$ since it is non-preemptive) to represent context-switches where a job is preempted;
- an edge from every $END$ to every $RE$ of a lower priority task (Type 6; in the example, both $\tau_1$ and $\tau_2$ to $\tau_3$; again, $\tau_2$ has no $RE$) to represent context-switches where a job resumes from preemption.

|  | FT number | Trivial Bound | Graph Algorithm |
|---|---|---|---|
| Original Example | 8 | 11 | 8 |
| All Tasks Preemptive | 9 | 11 | 9 |
| All Tasks Non-Preemptive | 5 | 6 | 5 |

TABLE IV: Example task set: FT bounds

Note that since for each $\tau_j$, there exists another task $\tau_k$ such that $noleak(\tau_k, \tau_j) = T$, all edges from the source are dashed; following Table III, the only dotted edges are from $\tau_1$ to $\tau_3$ and from $\tau_3$ to $\tau_2$.

Table IV summarizes the $N_{ft}$ bounds computed by the exact SMT formulation, the min-cost flow algorithm, and the trivial bound on the example task set, for the three preemptivity assignments of Figure 2, 3 and 4. Note that in this case, the bounds computed by the min-cost flow algorithms are exact, while the trivial bound always overestimates the value of $N_{ft}$. Finally, the computed bound corresponds to the number of FT for the schedule reported in the figures.

Theorem 1 states that the flow algorithm is indeed always correct; the main intuition is that we can algorithmically construct a flow to match any feasible job schedule.

**Theorem 1.** *Let $\bar{A}$ be the min-cost for the flow graph in Definition 3. Then $-\bar{A}$ is a valid upper bound to $N_{ft}(noleak, \{I_j | \tau_j \in hp_i\})$.*

*Proof:* Let $\phi$ be any valid sequence of job executions in the busy interval, *i.e.,* any sequence that respects the number of interfering jobs $I_j$ and preemptivity for each task $\tau_i \in hep_i$. [2] The proof will show that we can construct a valid flow assignment $\hat{f}$ on the graph that results in a cost $\hat{A} = -N_{ft}(\phi)$, where $N_{ft}(\phi)$ is the number of FTs required in sequence $\phi$ (assuming that the first job in the sequence, say of a task $\tau_f$, suffers a FT if it is possible, i.e., there exists any task $\tau_k$ such that $noleak(\tau_k, \tau_f) = T$). But since $\bar{A}$ is the min-cost for the flow graph, it must hold $-\bar{A} \geq -\hat{A} = N_{ft}(\phi)$; hence, $-\bar{A}$ is indeed an upper bound to $N_{ft}(noleak, \{I_j | \tau_j \in hp_i\})$, since it bounds the number of $FT$ required by any valid sequence of job executions in the busy interval of $\tau_i$.

To ease exposition, we first summarize how the rest of the proof works. We first construct a valid, initial flow assignment $\tilde{f}$ that mirrors the valid sequence $\phi$ by exchanging one units of flow along a chain of vertices representing the jobs in the sequence. We then obtain the desired flow assignment $\hat{f}$ by modifying $\tilde{f}$; intuitively, this is performed by removing the flow through the vertices representing certain jobs in the sequence. The resulting flow $\hat{f}$ has a cost $\hat{A} = -N_{ft}(\phi)$ by construction; we finally show that $\hat{f}$ is a valid flow.

We construct the initial flow $\tilde{f}$ by starting with an assignment $\tilde{f}(e) = 0, \forall e \in E$ and then adding flow to edges in this way: 1) we send one flow unit on the edge from *source* to $\tau_f.ST$, the task of the first job in the sequence $\phi$. 2) Next, we consider each context-switch between jobs in $\phi$. Assume that execution switches from a job of a task $\tau_k$ to a job of a task $\tau_l$. Then we add one unit of flow from either $\tau_k.END$ or $\tau_k.PR$, depending on whether the job of $\tau_k$ ends or is preempted, to either $\tau_l.ST$ or $\tau_l.RE$, depending on whether the job of $\tau_l$ starts or resumes execution after a preemption. 3) We send one unit of flow from $\tau_i.B$ to *sink*, where $\tau_i$ is the task under analysis. 4) Finally, for any task $\tau_j \in hep_i$, we send flow between vertices $\tau_j.ST, \tau_j.END, \tau_j.RE, \tau_j.PR$ and vertex $\tau_j.B$, such that the flow conservation is met at vertices $\tau_j.ST, \tau_j.END, \tau_j.RE, \tau_j.PR$. Example: consider Figure 11 and the related job sequence in Figure 2. Note $\tilde{f}(\tau_3.PR \rightarrow \tau_1.ST) = \tilde{f}(\tau_1.END \rightarrow \tau_3.RE) = 3$ since $\tau_3$ is preempted three times by $\tau_1$ and then immediately resumes. Since $\tau_3$ is preempted two more times by $\tau_2$, we also need to set $\tilde{f}(\tau_3.B \rightarrow \tau_3.PR) = \tilde{f}(\tau_3.RE \rightarrow \tau_3.B) = 5$ to meet flow conservation at vertices $\tau_3.PR$ and $\tau_3.RE$.

It is straightforward to see that $\tilde{f}$ is valid flow. Since at most $I_j$ jobs of a task $\tau_j$ can be executed in $\phi$, it follows that at most $I_j$ units of flow can be sent/received by vertices $\tau_j.END$ and $\tau_j.ST$, respectively; hence, the capacity constraint on edges $\tau_j.ST \rightarrow \tau_j.B$ and $\tau_j.B \rightarrow \tau_j.END$ are respected; all other edge capacities are obviously respected since they are infinite. The flow conservation constraint at vertices *source*, *sink*, and $\tau_j.ST, \tau_j.END, \tau_j.RE, \tau_j.PS$ for all tasks is respected by construction. Finally, since the number of times a job of task $\tau_j$ starts executing in $\phi$ must be equal to the number of times it finishes executing, and furthermore the number of times a job of $\tau_j$ is preempted must be equal to the number of times that the job resumes from preemption, the flow conservation is also respected for $\tau_j.B$. Hence, flow $\tilde{f}$ is valid.

Unfortunately, the cost $\tilde{A}$ of the constructed flow $\tilde{f}$ might not match $-N_{ft}(\phi)$: there might exist a context-switch between jobs of tasks $\tau_k$ and $\tau_l$ in the sequence, such that $\tau_l$ requires a FT, but the corresponding edge cost for the context-switch is 0. Consider the example of Figure 2, where a flush is required before executing the first job of $\tau_2$ once it preempts $\tau_3$. In this case $noleak(\tau_3, \tau_2) = F$, so sending flow on the edge $\tau_3.PR \rightarrow \tau_2.ST$ has a cost of 0, but we still need to flush because $noleak(\tau_1, \tau_2) = T$ and $\tau_1$ has been executed since the last FT. To solve the problem, we can intuitively obtain $\hat{f}$ from $\tilde{f}$ by removing the execution of $\tau_3$ between $\tau_1$ and $\tau_2$, so that we send flow directly on the edge from $\tau_1.END$ to $\tau_2.ST$, which has a cost of -1.

More precisely, assume that a FT is required for a job of $\tau_l$ in $\phi$, that the task of the job that causes the FT is $\tau_p$ (*i.e.,* the task of the latest job to execute in $\phi$ before the job of $\tau_l$, such that $noleak(\tau_p, \tau_l) = T$) and that the two jobs are not executed one after the other in $\phi$. Then to obtain $\hat{f}$ from $\tilde{f}$, for any such job $\tau_l$ we add one unit of flow to the edge from the corresponding vertex of $\tau_p$ (either $\tau_p.END$ or $\tau_p.PR$, based on $\phi$) to the corresponding vertex of $\tau_l$ ($\tau_p.ST$ or

[2] As stressed in Section IV, note that the definition does not consider the exact arrival times of tasks in the busy interval.

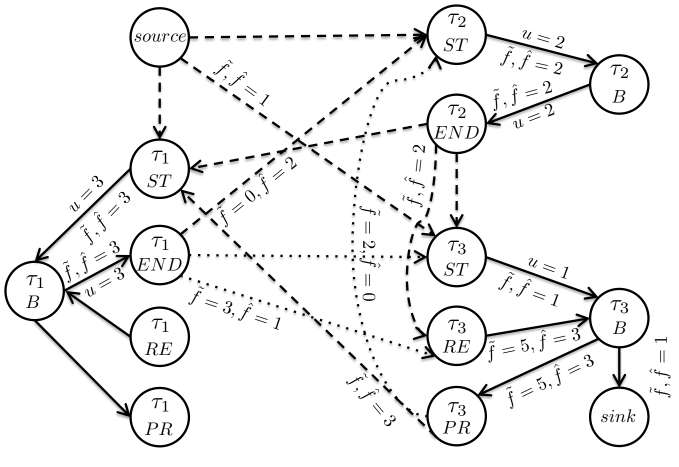


Fig. 11: Example task set: flow graph. Where not explicitly labeled, an edge has $u = +\infty$ and $\tilde{f} = \bar{f} = 0$. Solid edges and dotted edges have cost $a = 0$. For dashed edges, $a = -1$.

| $noleak$ | | to | | | | |
|---|---|---|---|---|---|---|
| | | $\tau_1$ | $\tau_2$ | $\tau_3$ | $\tau_4$ | $\tau_5$ |
| from | $\tau_1$ | - | F | F | T | F |
| | $\tau_2$ | F | - | T | F | F |
| | $\tau_3$ | T | F | - | F | F |
| | $\tau_4$ | F | T | F | - | F |
| | $\tau_5$ | F | F | F | F | - |

TABLE V: Example non-tight task set: *noleak* relation. Tasks indexed in inverse priority order; $\tau_5$: task under analysis. $I_j = 1$ for all tasks in $hep_5$. $\tau_3$ is the only preemptive task.

$\tau_p.RE$) and we remove the flow that would circulate between vertices corresponding to jobs that are executed in $\phi$ between the jobs of $\tau_p$ and $\tau_l$. The resulting cost $\hat{A}$ of $\hat{f}$ must be equal to $-N_{ft}(\phi)$, since by construction we send one unit of flow on an edge with $a = -1$ for each FT in $\phi$. Example: in Figure 11, $\hat{f}(\tau_1.END \rightarrow \tau_3.RE) = 1, \hat{f}(\tau_3.PR \rightarrow \tau_2.ST) = 0, \hat{f}(\tau_1.END \rightarrow \tau_2.ST) = 2$ (rather than values of $3, 2, 0$ for $f$), since for each of the two jobs of $\tau_2$ we need to send flow directly from $\tau_1.END$ to $\tau_2.ST$ rather than circulating flow from $\tau_1.END$ to $\tau_3.RE$ and from $\tau_3.PR$ to $\tau_2.ST$.

We finally show that $\hat{f}$ is still a valid flow. First note that removing one unit of flow circulating through a task $\tau_j$ cannot violate graph constraints for $\tau_j$ itself: reducing the amount of flow cannot violate a capacity constraint, and since we are removing both one unit of incoming flow from either $\tau_j.ST$ or $\tau_j.RE$, and one unit of outgoing flow from either $\tau_j.END$ or $\tau_j.PR$, the flow conservation at $\tau_j.B$ is still respected. It remains to show that we can add one unit of flow to the edge from the vertex of $\tau_p$ to the vertex of $\tau_l$; this is not trivial since the corresponding edge might not exist in the graph. If the job of $\tau_p$ sends flow from $\tau_p.END$ and the job of $\tau_l$ receives flow on $\tau_l.ST$, then this is trivially true since there is an edge between the $END$ and $ST$ vertices of any two tasks. Therefore, consider the two following remaining cases:

**Case 1:** the job of $\tau_p$ sends flow from $\tau_p.PR$. Since the considered job is the last of $\tau_p$ to execute before the one of $\tau_l$, it follows that $\tau_l$ is preempting $\tau_p$ in the schedule of $\phi$. Hence, $\tau_l$ must be higher priority than $\tau_p$, and furthermore the considered job of $\tau_l$ must be starting execution (rather than resuming from preemption), thus it must be receiving flow to $\tau_l.ST$. Therefore, we can add one unit of flow to the edge from $\tau_p.PR$ to $\tau_l.ST$, which exists in the graph.

**Case 2:** the job of $\tau_l$ receives flow on $\tau_l.RE$. This case is specular to the previous one, in the sense that $\tau_p$ must be preempting $\tau_l$. Therefore, we can add one unit of flow to the edge from $\tau_p.END$ to $\tau_l.RE$, which exists in the graph. ∎

**Graph Bound Tightness:** Note that Theorem 1 only shows that the resulting bound is safe. As a matter of fact, there are task sets where the computed bound is not tight; an example is shown in Table V. Solving the flow graph results in a value $N_{ft} = 5$, for the implied schedule shown in Figure 12. Note that this schedule satisfies the constraints of the graph, since each direct preemption and resumption ($PR$ to $ST$ and $END$ to $RE$ events) satisfies priority ordering. Nevertheless, this schedule is invalid, since lower-priority task $\tau_4$ is indirectly preempting task $\tau_3$. One possible valid worst-case schedule is shown in Figure 12, where the number of FTs is equal to 4; in fact, in this case it is easy to see that there is no valid schedule that results in $N_{ft} > 4$, since for each task $\tau_j \in hep_5$, there is a unique task $\tau_k$ such that $noleak(\tau_k, \tau_j) = T$.

**Graph Bound Computational Complexity:** Orlin's algorithm [20] for the min-cost flow problem has a complexity of $O(|E|^2)$, where $|E|$ is the number of edges in the graph. The number of edges based on Definition 3 is $O(N^2)$ in the number

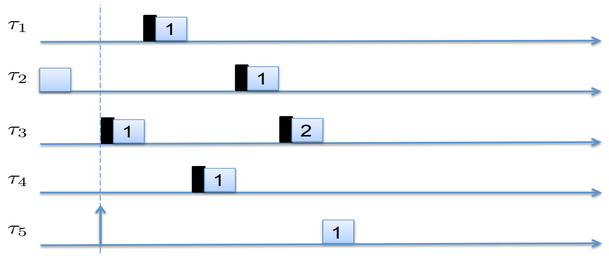


Fig. 12: Example non-tight task set: invalid schedule implied by the FT Graph.

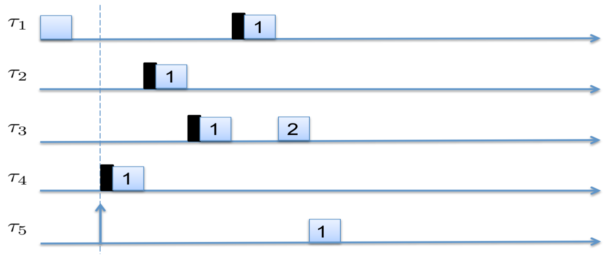


Fig. 13: Example non-tight task set: valid worst-case schedule.

of tasks in $\Gamma$, since we need a fixed number of edges between any two tasks in $hp_i$. Hence, the overall complexity of deriving the graph bound is $O(N^4)$.

## V. Optimal Preemptivity Assignment

Based on the schedulability analysis in Section IV, Algorithm 1 details how to assign preemptivity $preempt_i$ to every task $\tau_i \in \Gamma$. The algorithm is optimal, in the sense that if there exists any $preempt_i$ assignment that makes the task set schedulable according to our analysis, then Algorithm 1 will find one such assignment.

The algorithm iterates on all tasks starting from the highest priority task $\tau_1$ to the lowest priority task $\tau_N$. At each step, the algorithm tries to determine if the current task $\tau_i$ can be executed non-preemptively; it does so by checking that the blocking time that executing $\tau_i$ non-preemptively would cause on each higher priority task $\tau_j$ would not make $\tau_j$ unschedulable (Line 2), by checking that $\bar{c}_i - 1$ is not greater than the slack $\Delta_j$ of $\tau_j$, computed assuming zero blocking time (Line 7, since we do not know if any lower priority task is non-preemptive this point). The main intuition is that if $\tau_i$ can be executed non-preemptively, then doing so is convenient; setting $preempt_i = F$ can potentially reduce the number of FT suffered by $\tau_i$ and lower priority task (Lemma 1), and furthermore reduce the number of jobs interfering with $\tau_i$ (Lemma 2). Based on the lemmas, Theorem 2 then states that Algorithm 1 is optimal.

**Lemma 1.** *Consider the bound on $N_{ft}(noleak, \{I_j | \tau_j \in hp_i\})$ computed by either the trivial, graph or exact algorithm, and let $\tau_j \in hep_i$ be a non-preemptive task. Changing $\tau_j$ to execute preemptively results in a bound on $N_{ft}$ that is no less than the original one.*

*Proof:* In the case of the trivial bound, the proof follows immediately from Equation 6, since changing $\tau_j$ to execute preemptively cannot reduce the set $\Gamma_i^2$.

For the graph bound case, it suffices to note that if $\tau_j$ is executed preemptively, additional vertices and edges are added to the min-cost graph, but none are removed. Hence, any valid flow when $\tau_j$ is non-preemptive is also a valid flow when $\tau_j$

is preemptive, which implies that the resulting bound for $N_{ft}$ cannot decrease.

Finally, in the case of the exact bound, notice that any task ordering that is valid for the non-preemptive case is also valid in the preemptive case: since we make no assumption on the exact arrival time of jobs, if a job of task $\tau_j$ with $j < k$ follows a job of $\tau_k$ rather then preempting it, we can simply assume that the job of $\tau_j$ arrives immediately after the one of $\tau_k$ finishes executing. Again, this implies that the preemptive case cannot result in a lower $N_{ft}$ number, concluding the proof. ■

**Lemma 2.** *The slack $\Delta_i$ computed according to Equation 5 for the case when $\tau_i$ is non-preemptive cannot be less than the slack when $\tau_i$ is preemptive.*

*Proof:* According to Lemma 1, when $\tau_i$ is non-preemptive the number $N_{ft}$ of FTs suffered by the task is less than or equal to the preemptive case. Furthermore, note that the values $I_j$ computed according to Equation 2 in the non-preemptive case are similarly less than or equal to the values computed according to Equation 3. Therefore, based on Equation 5 the slack with $preempt_i = F$ is greater than or equal to the slack with $preempt_i = T$. ■

**Theorem 2.** *The preemptivity assignment of Algorithm 1 is optimal for schedulability analysis based on the slack time computation in Equation 5, where $N_{ft}(noleak, \{I_j | \tau_j \in hp_i\})$ is derived according to either the trivial, exact or graph bound.*

*Proof:* The proof proceeds by induction on the task index $i$ in Algorithm 1. In particular, we prove the following property: if Algorithm 1 sets $preempt_i = T$, then there exists no preemptivity assignment for $\{\tau_1, \ldots, \tau_{i-1}\}$ such that $preempt_i = F$ and tasks $\{\tau_1, \ldots, \tau_{i-1}\}$ are schedulable. In other words, the algorithm always assigns a task $\tau_i$ to execute non-preemptivily if it is feasible to do so. Based on Lemma 2, this implies that the algorithm is optimal.

**Base case:** the property is trivial for $i = 1$, since $\tau_1$ is always assigned to execute non-preemptively.

**Inductive step:** by contradiction, assume that the algorithm assigns $preempt_i = T$, but there exists a preemptivity assignment $\{\overline{preempt}_1, \ldots, \overline{preempt}_{i-1}\}$ such that $preempt_i = F$ and $\{\tau_1, \ldots, \tau_{i-1}\}$ are schedulable. Let $\{preempt_1, \ldots, preempt_{i-1}\}$ be the preemptivity assignment that the algorithm picked at previous steps $1, \ldots, i-1$. Since the algorithm assigns $preempt_i = T$, then tasks $\{\tau_1, \ldots, \tau_{i-1}\}$ cannot be schedulable with assignment $\{preempt_1, \ldots, preempt_{i-1}, preempt_i = F\}$: Line 2 must have evaluated to false, meaning that making $\tau_i$ non-preemptive would cause at least one task in $\{\tau_1, \ldots, \tau_{i-1}\}$ to miss its deadline due to excessive blocking time. Therefore, it follows that the two assignments $\{\overline{preempt}_1, \ldots, \overline{preempt}_{i-1}\}$ and $\{preempt_1, \ldots, preempt_{i-1}\}$ must be different. We now have two cases:

**Case 1:** there exists at least one task $\tau_k \in hp_i$ such that $\overline{preempt}_k = F$ and $preempt_k = T$. This contradicts the inductive hypothesis: since $\tau_k$ is feasible with the assignment $\{\overline{preempt}_1, \ldots, \overline{preempt}_k = F\}$, at step $k$ the algorithm must have assigned $preempt_k = F$.

**Case 2:** for all tasks $\tau_k \in hp_i$ such that $\overline{preempt}_k \neq preempt_k$, it holds $\overline{preempt}_k = T$ and $preempt_k = F$. This contradicts the assumption that $\{\tau_1, \ldots, \tau_{i-1}\}$ are schedulable under $\{\overline{preempt}_1, \ldots, \overline{preempt}_{i-1}, preempt_i = F\}$ but not under $\{preempt_1, \ldots, preempt_{i-1}, preempt_i = F\}$: based on Lemmas 1, 2, the slack of a task cannot decrease when either the task itself or a higher priority task is executed non-preemptively.

Since neither case is possible, the inductive step follows.

■

**Algorithm 1** Preemptivity Assignment

```
1:  for $i = 1 \ldots N$ do
2:      if $\forall j = 1 \ldots i-1 : \bar{c}_i - 1 \leq \Delta_j$ then
3:          $preempt_i \leftarrow F$
4:      else
5:          $preempt_i \leftarrow T$
6:      end if
7:      Compute $\Delta_i$ based on Equation 5 using $B_i = 0$
8:      if $\Delta_i < 0$ then
9:          Return FAIL
10:     end if
11: end for
12: Return SUCCESS
```

## VI. Evaluation

To evaluate the effectiveness of the proposed mechanism, we implemented NLF in our avionics demonstrator. We also applied the derived schedulability analysis to both the task set in our demonstrator example, as well as sets of synthetic tasks. We begin by describing in more details our implemented hardware and software platform. Sections VI-A and VI-B then discuss schedulability results in more details.

**Hardware Platform:** To allow indoor experimentation, the UAV part of the platform comprises a combination of an actual aerial vehicle and a Hardware-In-the-Loop (HIL) simulator. The vehicle has 3 degrees of freedom, with its actual position being fixed. The HIL component uses the vehicle's dynamics to simulate changes in position and returns a GPS-like positional signal to the ECU. The ECU platform is based on a Xilinx FPGA using an ARM Cortex A9 processor core running at 667 Mhz. Our FT implementation focuses on the processor cache, which is the most easily exploitable stateful resource on the platform; we employ available hardware functionality to flush the entire cache content (both L1 and L2).

**Software Implementation:** We implemented the described NLF mechanism by modifying the FreeRTOS real-time kernel. FreeRTOS natively supports preemptive fixed-priority scheduling. We extended the task descriptor to include preemptability information, and modified the scheduling function to avoid rescheduling while a non-preemptive task is running. We then modified the context-switch function to determine whether a FT is required before executing a task. Our implementation is able to perform the check in constant time as long as the number of tasks in the system is less than the bit-width of the machine (32 bits for our hardware platform) by using bit-arrays encoded as integers. In details, we maintain a binary array $mustflush[i]$ which determines whether a FT is required before executing task $\tau_i$. The array is reset to 0 whenever a FT is executed. Whenever a task $\tau_j$ is executed, we then update the array such that $mustflush[i] \leftarrow mustflush[i] | noleak(\tau_j, \tau_i)$, i.e., $mustflush[i]$ is set to 1 if executing $\tau_j$ requires a FT for $\tau_i$. Since the array is implemented as an integer, we can perform the update in constant time using bit-wise logical operations.

## A. Demonstrator Results

| Sens. | Laws | Act. | MP | Net. | AES | JPEG | I/O |
|---|---|---|---|---|---|---|---|
| F | F | F | F | F | F | T | F |

TABLE VI: Demonstrator: Preemptivity Assignment

We used the CPU cycle counter to measure the worst-case running time for the FT function. It was found to be $340 \mu s$, which corresponds to the case where the entire cache content has been modified, and must be written back to main memory. We then used our derived preemptability assignment and schedulability analysis to assign $preempt_i$ values and determine feasibility. We tested using the trivial bound, graph bound and exact bound, as well as normal RM scheduling with no flushes. Table VI shows results in terms of preemptability assignment, which is the same for all algorithms, while Table VII shows response time results in terms of the maximum response time / period ratio for any task.

| Algorithm | Max Response Time Ratio | Min Period(ms) |
|---|---|---|
| RM (no flush) | 64% | 27 |
| Exact (Z3) | 75% | 32 |
| Graph | 75% | 32 |
| Trivial | 83% | 36 |

TABLE VII: Demonstrator: Schedulability Results

All mechanisms result in a schedulable system, which corresponds to our observation running the demonstrator. Response time ratio for the trivial bound is appreciatively worse than for the graph bound; RM has the best schedulability but does not provide any security guarantees. We also used the analysis to determine the minimum period at which the image subsystem could be run while still remaining schedulable. Once again, results for the graph bound are better than for the trivial bound. Finally, results for the exact bound matched the graph bound, although the analysis took 7 minutes, whereas computing the graph bound took less than a second.

## B. Synthetic Results

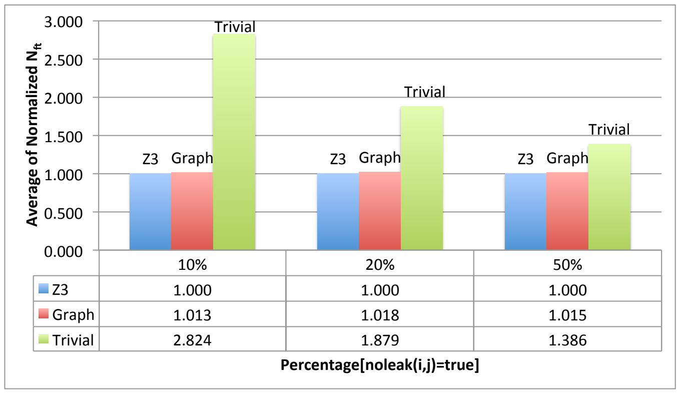
TABLE VIII: Experimental Parameters.

| Parameter | Value |
|---|---|
| Number of tasks, $N$ | $[5, 20]$ |
| Task period, $p_i$ | $[5ms, 100ms]$ |
| Task execution time, $c_i$ | $[0.3ms, 3ms]$ |
| FT overhead, $c_{ft}$ | $\{0.1ms, 0.5ms\}$ |

Table VIII summarizes the parameters used for the generation of the synthetic task sets used in our evaluation. We generated 3000 task sets that fall into each utilization group, $[0.02 + 0.1 \cdot i, 0.08 + 0.1 \cdot i]$ for $i = 0, \ldots, 9$, *i.e.,* 300 sets per group. The base utilization of a task set is defined as the total sum of the task utilizations. Each group is generated with the following three different settings; the first 100 sets with the probability of $noleak(\tau_i, \tau_j)$ for any pair of tasks being 10%, the next 100 with the probability of 20%, and the last 100 sets with a probability of 50%.

Each input instance consists of $[5, 20]$ tasks, each $\tau_i$ of which has a period $p_i \in [5ms, 100ms]$ and an execution time $c_i \in [0.3ms, 3ms]$. The deadline of each task is equal to its period, *i.e.,* $d_i = p_i$. Task priorities are assigned according to the Rate Monotonic (RM) algorithm [15]. Except for the last set of experiments, the preemptiveness of each task is assigned in a random manner. These task parameters are in line with the values measured on the demonstrator platform where a typical FT execution time was $0.34ms$ (Section VI-A).

*1) Evaluation of FT bound:* We first evaluate our (approximate) FT bound by comparing it with the exact bound found by the SMT solver as well as the trivial bound. For each task set, we first calculate the worst-case response time of the lowest-priority task using the graph-based analysis from Section IV. Then, we feed the information on the number of higher priority jobs and the no-leak matrix to an SMT solver so that it can calculate the exact bound on the number of FT invocations. The results are based on the calculation of the number of flushes that occur during the worst-case busy period of the lowest-priority task.



| | 10% | 20% | 50% |
|---|---|---|---|
| 🟦 Z3 | 1.000 | 1.000 | 1.000 |
| 🟥 Graph | 1.013 | 1.018 | 1.015 |
| 🟩 Trivial | 2.824 | 1.879 | 1.386 |

Fig. 14: Flush Task Bounds as calculated by the SMT solver, the Graph-based Analysis and the Trivial Analysis

Figure 14 shows the geometric mean of the number of flushes (that occur during the lowest-priority task's busy period) found by the graph-based approximation (*Graph*) and by the trivial bound (*Trivial*) normalized to the exact bound, *Z3*. As we can see from the results, our graph-based approximation method computes a *tight bound on the number of flushes irrespective of the percentage of no-leak relation*. On average, this number is 1% or 2% more compared to the exact bound. On the other hand, the trivial bound is often three times the exact bound. Note that the trivial bound tends to be more pessimistic when the probability of $noleak(\tau_i, \tau_j) = T$ is low, since it is only based on the number of context-switches and ignores the no-leak relation itself.

*2) Evaluation of Schedulability:* We also evaluated the effects of the graph-based and trivial bounds on the schedulability of task sets by varying the flush times, $c_{ft}$, as well as the no-leak percentage. The results of these experiments are shown in Figures 15, 16 and 17.

The X-axis plots the utilization bins for the experiments while the Y-axis represents the total percentage of schedulable instances for task sets for each bin (100 tasks per bin). The graphs represent the schedulability of *(a)* no flushes (RM), *(b)* the number of flushes computed by the graph-based approximation (Graph) and *(c)* the trivial bound for the number of flushes (Trivial). The graphs also show the effects of varying the FT execution times (the different values of $C_{ft}$) and the percentage of $noleak(\tau_i, \tau_j)$ for each experiment.

Figures 15 and 16 show the two extreme cases *i.e.,* when the difference in the schedulability between 'Graph' and 'Trivial' is the largest and the smallest. The graph-based method outperforms the trivial bounds when the no-leak percentage is low since the latter is highly pessimistic (Figure 14). The schedulability of 'Trivial' further decreases as the overhead for flushing, $c_{ft}$, increases. On the other hand, the difference becomes smaller as the no-leak percentage becomes larger

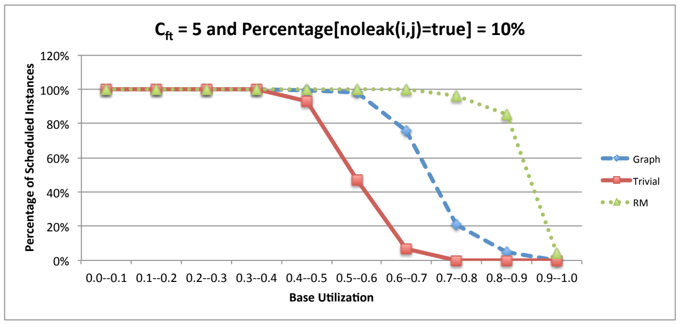


Fig. 15: Schedulability of Task Sets: Graph-based Bounds vs. Trivial Bounds [FT Overheads = 5, noleak = 10%]

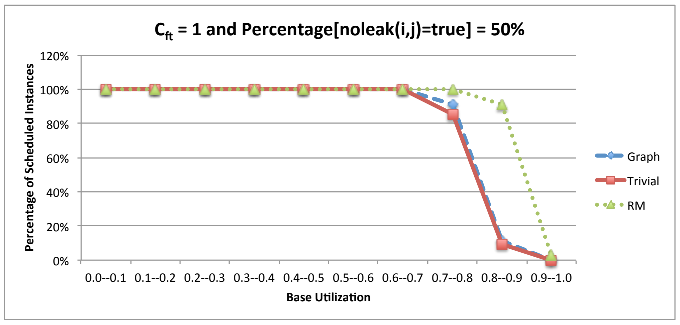


Fig. 16: Schedulability of Task Sets: Graph-based Bounds vs. Trivial Bounds [FT Overheads = 1, noleak = 50%]

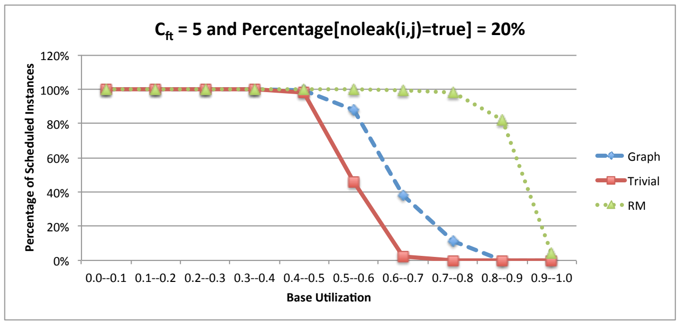


Fig. 17: Schedulability of Task Sets: Graph-based Bounds vs. Trivial Bounds [FT Overheads = 5, noleak = 20%]

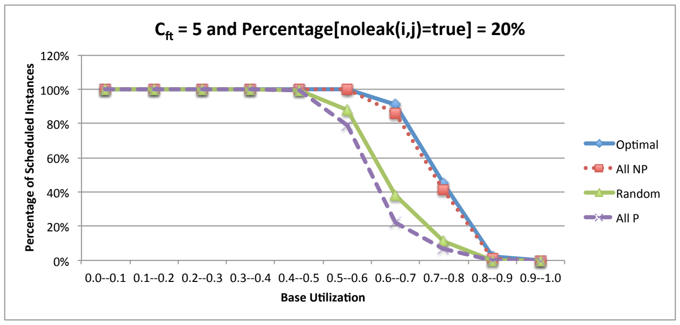


Fig. 18: Preemptivity Assignments [Optimal vs. All Non-Preemptive vs. All Preemptive vs. Random]

(and thus the pessimism of 'Trivial' becomes smaller) and the flushing time becomes shorter. Figure 16 shows the case when the performance of the graph-based method is similar to that of 'Trivial'. These figures also show that *(i)* schedulability drops as $c_{ft}$ increases because of the extended blocking times due to longer flushes and, *(ii)* schedulability drops as the no-leak percentage increases because of the greater chance of a flush occurring during a context switch.

Figure 17 shows behavior that lies between the two extreme cases presented above. It might even be a 'typical' case where $20\%$ of the tasks have the noleak relationship between them to be true and the flush time overheads are around $0.5ms$. As expected, our graph-analysis based bounds outperform the trivial bounds by being able to schedule more instances of task sets.

*3) Preemptability Assignment:* Finally, we performed synthetic experiments to evaluate the effect of task preemptability. The results are shown in Figure 18, where we plot the total percentage of schedulable task sets as a function of the utilization bin, similar to the previous experiment. The three graphs represent the optimal assignment using Algorithm 1 (Optimal), the case where all tasks are either preemptive (P) or non-preemptive (NP), and using random assignment (Random). Note that while NP performs better than P on average for this specific parameter assignment, in general the two approaches are incomparable, i.e., there are task sets which are schedulable by P but not NP and vice-versa. Being optimal, the assignment generated by Algorithm 1 performs better than either.

## VII. Related Work

A body of work on identification, analysis and mitigation of covert side channels (*e.g.,* [9]–[12], [21]) already exists. For instance, Hu [10] uses a similar mitigation strategy (cache flushing) and discusses how scheduling algorithms can be modified to minimize the number of flushes. However, tasks do not have real-time requirements and the schedulers do not support service guarantees. In real-time systems, it has been shown that scheduling of (real-time) tasks can be a source of information leakage [25]. Völp *et al.* [30] *(a)* discuss unauthorized information flows by use of scheduling behavior (*e.g.,* delayed preemption); *(b)* they showed how to modify fixed-priority schedulers to reduce the effect of malicious alterations; *(c)* studied the effect of timing channels introduced by real-time resource locking protocols and finally, *(d)* addressed them by transforming the relevant protocols [29]. In contrast, we focus on the more general approach of transforming security properties into real-time scheduling constraints; information leakage is just an example to illustrate our techniques.

In our previous paper [18] we established some initial ideas on how to integrate security-related constraints with real-time schedulers. The main differences between that work and what is presented in this paper are: *(a)* we have generalized the security model that is used to capture the relationships between tasks; *(b)* we broaden the types of real-time scheduling algorithms by further reducing constraints (preemptive vs. non-preemptive) from that paper and *(c)* we demonstrate the feasibility of our methods by implementing the concepts on a realistic application and hardware platform.

There exists some work on reconciling security mechanisms and real-time properties [14], [32]. They propose new scheduling algorithms (and modifications to existing ones) to meet real-time requirements while maximizing the level of security achieved. In contrast, we propose (and analyze) enhancements to FP schedulers to reduce information leakage through shared storage channels while still meeting real-time requirements. The issue of information leakage has been studied in real-time database systems [1], [26].

Recent work looks to develop architectural frameworks for solving problems such as intrusion detection [17], [24], [27], [34], [36], among others. The idea is to create hardware/software mechanisms to protect against security vulnerabilities while our work aims at the design (scheduler) level. It is not inconceivable that the two sets of approaches could be combined to make the system more resilient to attacks.

Finally, 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 [33] discusses how to compute the exact costs for preemptions for a task. A fundamental difference between this and our work is: we develop algorithms that don't need to cleanup state (invoke FT algorithms) at every preemption point.

## VIII. Conclusions and Future Work

We generalized the idea of using security-based constraints on real-time scheduling by introducing the *noleak* relation. This can be applied in a generic manner to capture constraints between tasks in a real-time system. We also demonstrated issues with implementing this model on a hardware-in-the-loop simulator along with an extensive evaluation, both theoretical and experimental. Designers of real-time systems can now gain a better understanding of the costs involved in integrating security requirements in hard real-time systems. This could potentially lead to better, safer RTS in the future.

For future work, we intend to study other security issues such as the integrity and availability in real-time systems. We also plan to extend our analysis to other classes of scheduling algorithms.

## 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] D. Bell and L. LaPadula. Secure Computer Systems: Mathematical Foundations and Model. Technical Report 2547, MITRE, March 1973.

[3] E. Bini and G. Buttazzo. Schedulability analysis of periodic fixed priority systems. *IEEE Transactions on Computers*, 53(11):1462 – 1473, 2004.

[4] 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.

[5] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

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

[7] 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.

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

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 45–54, 2013.

[17] 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.

[18] S. Mohan, M.-K. Yoon, R. Pellizzoni, and R. Bobba. Real-Time Systems Security Through Scheduler Constraints. In *Euromicro Conference on Real-Time Systems*, 2014.

[19] 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.

[20] J. Orlin. A polynomial time primal network simplex algorithm for minimum cost flows. *Mathematical Programming*, 78:109–129, 1997.

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

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

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

[24] 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.

[25] 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.

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

[27] 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.

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

[29] 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.

[30] 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.

[31] Z. P. Wu, Y. Krish, and R. Pellizzoni. Worst case analysis of dram latency in multi-requestor systems. In *Proceedings of the 34th IEEE Real-Time Systems Symposium*, 2013.

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

[33] 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.

[34] 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.

[35] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 305–316, New York, NY, USA, 2012. ACM.

[36] 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.