Avoiding Timing Channels in Fixed-Priority Schedulers

Marcus Völp, Claude-Joachim Hamann, Hermann Härtig
Technische Universität Dresden
Department of Computer Science
01062 Dresden
{voelp, hamann, haertig}@os.inf.tu-dresden.de

ABSTRACT

A practically feasible modification to fixed-priority schedulers allows to avoid timing channels despite threads having access to precise clocks.

This modification is rather simple: we compute at admission time a static predicate that states whether a thread may possibly leak information; if such a thread blocks we switch to the idle thread instead.

We describe the modified scheduler, provide a mechanical PVS-based proof of noninterference and show how common admission algorithms can be reused to give real-time guarantees for this modified scheduler. While providing similar isolation guarantees, our approach outperforms time-partitioning schedulers in terms of achieved real-time guarantees.

Categories and Subject Descriptors

D.4.6 [**Software**]: Operating SystemsSecurity and Protection[Information flow controls]; D.4.1 [**Software**]: Operating SystemsProcess Management[Scheduling]

General Terms

Security, Verification

Keywords

real-time, fixed-priority scheduling, security, information flow, noninterference

1. INTRODUCTION

We envisage open systems (though we do not restrict ourselves to) as described in Deng et al. [4], and Härtig et al. [7], where not necessarily trustworthy virtualised legacy operating systems and their applications execute next to security-sensitive and real-time critical applications on top of a small microhypervisor. In these systems potentially untrusted and malicious legacy code is used even for security critical operations. Therefore appropriate countermeasures must be applied to enforce that these systems preserve the confidentiality of the secret data they process. As an example application mix of such systems imagine a nonreal-time legacy operating system used for compiling and text processing, nonreal-time banking transactions, the bank credentials of which require protection and real-time network and disk drivers streaming cryptographic-protected video content to a real-time video player for display. Naturally, these different components get assigned different priorities (legacy OS, bank transaction — low priority, no real-time; drivers, video — high priority, real-time) and they are classified into different security classes (or levels) to reflect their confidentiality requirements (legacy OS, drivers — low security level, public; bank transaction, video — high security level, secret).

This paper is concerned with illegal information flows through the fixed-priority scheduling subsystem. Precisely, we investigate how malicious code (e.g., a virus or a Trojan Horse executing inside a virtual machine) can exploit the scheduling subsystem to illegally transfer information and how this information leakage can be prevented by modifying a budget-enforcing fixed-priority scheduler to treat potentially-leaking blocked threads as if they were ready. We provide a machine checked proof that this countermeasure not only prevents all direct information flows but also information flows that happen indirectly via components we trust not to leak information intentionally (like for example, the cryptographic wrapper connecting the real-time disk and network drivers with the real-time video player).

Naturally, modifying the scheduler affects which real-time guarantees can be given. To determine whether a thread set will meet their real-time requirements under a given scheduler (i.e., to decide whether they are schedulable), an admission algorithm is run. For fixed-priority schedulers, such admission algorithms typically consider blocking times that may occur for example as a side effect of resource contention and self-suspension. So, by adjusting blocking times, we can reuse this large class of acceptance algorithms that are available for fixed-priority scheduling to determine whether a thread set is schedulable with our modified scheduler. The thread set may thereby contain real-time threads and nonreal-time application threads and the virtual CPUs which are the scheduling entities of virtual machines visible to the hypervisor.

The remainder of this paper is structured as follows. In Section 2 we briefly introduce our scheduling model which supports arbitrary statically created thread sets run on a

©ACM, (2008). This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in ASIACCS'08, March 18-20

budget-enforcing fixed priority scheduler. In particular, strictly-periodic threads are contained in this model. We investigate how information flow can occur through alterations in a thread's scheduling related behaviour in Section 3 and present how the scheduler must be modified to prevent this illegal information flow in Section 4. A sketch of the noninterference proof for this scheduler is given in Section 5. We deduce in Section 6 that not only unauthorised information flow is prevented, but also that our modification isolates threads with no authorised information flows in a timely fashion. After that we restrict ourselves to strictly-periodic threads when we discuss how our approach affects real-time guarantees in Section 7. This restriction is partially lifted in the discussion on further practical factors in Section 8. Section 9 relates our work to other work in the area; Section 10 concludes.

2. BUDGET-ENFORCING FIXED-PRIORITY SCHEDULING

In classical real-time scheduling systems, threads are typically executed unconstrained according to their scheduling parameters after a worst-case analysis has been performed. Programming errors or attacks from malicious programs may, however, cause threads to exceed their worst-case execution times. Budget-enforcing schedulers are robust against these attacks. If in between two releases, a thread τ_i has executed for an amount of time equal to its worst-case execution budget ($wcet_i$) any further execution is deferred to the next release. Likewise, if blocking is considered, further execution is deferred to the next release after a total worst-case budget $wct_i := wcet_i + wcbt_i$ is depleted where $wcbt_i$ is the worst-case blocking time.

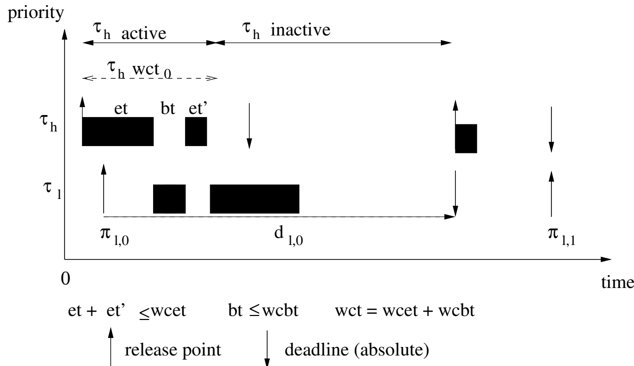

Figure 1: Scheduling according to fixed priorities. During times when τ_h blocks, the lower prioritised thread τ_l may run.

We describe an arbitrary thread $\tau_i \in T$ $(i = 1, ..., n)$ of the thread set T through a possibly infinite sequence of releases. The kth release $(k = 0, 1, ...)$ is parametrised by a release point $\pi_{i,k}$, the budgets for this release $wcet_{i,k}$ and $wct_{i,k}$ which get refilled at this release point and a relative deadline $d_{i,k}$. A real-time thread must have completed its work for this release latest at $\pi_{i,k} + d_{i,k}$. Otherwise it is said to miss this deadline. Execution is deferred if either the budgets have expired or the deadline has passed.

Examples for releases include the beginning of a period for periodic threads and the arrival of publicly visible events such as the arrival of a network package. Because we constrain these events to be publicly visible, no information is leaked due to the fact that a thread is released.

The above model is sufficient to express the real-time properties of a wide range of common real-time thread sets. For example, strictly-periodic threads are described by an infinite sequence of equidistant release points (i.e., $\pi_{i,k+1} - \pi_{i,k} = const$). In addition, strictly-periodic threads are assigned the same worst case budgets in each period. The events activating aperiodic threads are directly reflected by the releases, provided these are public. Deferred servers can be described as a sequence of releases where each release describes the arrival of a thread executed on these servers and the budget of this release corresponds to the worst case execution time of this thread. The total budget of the deferred server in a given period is the sum of the budgets assigned to the releases in this period. Thread sets consisting of virtual machines that have been assigned a share of processor times can be mapped to the above model by assigning them budgets proportional to their share and by releasing all virtual machines at equidistant release points whose distance corresponds to the sum of assigned budgets. Thread sets under a time-partitioning scheduler can be mapped by setting the release points to the respective partition begins and by setting the budgets to their size.

Threads are scheduled in a fixed-priority based manner. Hence, a priority $prio_i$ is a further parameter of a thread τ_i. We assume, that the scheduler supports sufficiently many priorities so that each two threads are assigned different priorities. High-priority threads may preempt lower prioritised threads at any instant. It is a common approach to disregard context switching overhead or to include it in the worst case execution time.

Furthermore, threads may suspend themselves (e.g., when waiting for I/O completion in a blocking systemcall) and temporarily disable preemptions (e.g., during critical sections). Otherwise, threads are assumed to be independent, that is, there are no temporal precedence constraints. In Section 7f, we show how these restrictions can partially be lifted and investigate in detail blocking due to self suspension (Section 7.1), blocking due to nonpreemptibility (Section 7.2), resources (Section 8.1) and precedence constraints (Sections 8.2) for strictly-periodic thread sets. In addition, we will investigate how real-time guarantees are preserved for aperiodic and sporadic threads in Section 8.3.

A thread may be in one of the following four commonly-used states:

running the thread is assigned a processor and is executing

ready the thread is not executing but ready for execution (it has all required resources with the exception of a processor)

blocked the thread has suspended itself and waits for some external event or for some signal from another thread

inactive the thread budgets have been depleted respectively the deadline has passed. A thread is inactive until its next release point. At this time, its worst case budgets get refilled to the budgets of this release.

In addition to these states, we define the term **active** to denote a thread which is not inactive, i.e., which has a positive remaining worst case budget. Such a thread may

be running, ready, blocked or it may have completed its execution in its current release without having exhausted its budgets. In this latter case, the thread remains active until an amount of time equal to the remaining budget has passed. A thread which has not consumed its worst case budgets for a given release we say has **stopped early**.

Throughout the paper, τ_h denotes a high prioritised thread and τ_l denotes a low (or lower than τ_h) prioritised thread. We write $prio(\tau_h) > prio(\tau_l)$ despite of the numerical values of the priorities. Furthermore, let $T_{high}(\tau)$ and $T_{low}(\tau)$ be the set of threads with higher respectively lower priority than τ. We write $T_{P,high}$ for the set of threads in T_{high} which in addition fulfil a given predicate on threads P (respectively $T_{P,low}$ for threads in T_{low}). Figure 1 illustrates the introduced notions.

3. INFORMATION FLOW

Obviously, higher prioritised threads can directly influence when and for how long lower prioritised threads run. When a higher prioritised thread τ_h blocks, the scheduler will select a lower prioritised thread τ_l; when it executes this selection will be deferred until τ_h blocks or its budget is depleted (see Foss et al. [25] for a more detailed analysis describing which information can be deduced in rate-monotonic scheduling). In the presence of legitimate communication between threads, indirect influences become possible by directly influencing a sender which in turn relays timing information in its messages. In Section 3.2, we investigate this indirect influence in detail.

The actions leading to direct or indirect influences we call **run** — the thread executes some code — and **block** — the thread has invoked some blocking systemcall.

Before defining the noninterference property let us clarify our security-related assumptions.

1. *The scheduling parameters, the scheduling algorithm and therefore the resulting schedule are public information. In particular, the events which trigger releases must be publicly visible.*
2. *All threads have access to precise clocks.* [1]

Note it is still possible to hide the existence of threads by scheduling them hierarchically (see e.g., [18]) on top of a thread that is visible in the public schedule. Note further that hierarchical scheduling is a means of supporting dynamic thread creation, a functionality which is not per se supported by our approach. Other means to create threads dynamically are discussed in the course of this paper.

3.1 Noninterference

Noninterference [19] characterises the absence of illegal information flows through a system, in our case the scheduling subsystem. Suppose no information may flow from a thread τ to a thread τ' via the scheduling system S. We qualify this by asserting that what τ' can observe about S remains unchanged despite differing behaviour (i.e., differing action sequences) of τ.

More formally, we define an information-flow policy as a triple $(C, \leq, dom)$ where $(C, \leq)$ is a bounded lattice over the finite set of domains (or security classes) C, $\leq$ is a partial order on security levels. In particular, $\leq$ is transitive. $dom : T \rightarrow C$ is a function, which assigns each thread $\tau \in T$ a security class $c \in C$. Information may flow from τ' to τ provided $dom(\tau') \leq dom(\tau)$. We write $dom(\tau) \nleq dom(\tau')$ to denote that no such information flow may happen (thus, $\nleq$ is the complement of $\leq$). $\top \in C$ denotes the top (or greatest) element of the lattice $(C, \leq)$, that is, $\forall c \in C.\ c \leq \top$ holds (analogously, $\bot$ denotes the bottom element).

Given access to a precise clock, a thread τ may observe the precise points in time when it gets selected by the scheduler. In addition, it may learn through messages about the points in time when other threads τ' run, provided τ may legitimately receive messages from these threads.

Following Rushby [19], we can now formally define our noninterference property:

Definition 1. Noninterference.
Let $output(S)(\tau, t)$ be a function which returns for each point in time t the thread τ' that was selected by the scheduler to run at t, provided that τ may receive from τ' (i.e., provided $dom(\tau') \leq dom(\tau)$) and which otherwise returns the special symbol $- \notin T$. (We will give a precise formal definition of this function in Section 5.) Let $purge(S)(\tau)$ be a function which removes from S all the actions — though not the threads themselves — of those threads τ' from which no information must flow to τ (i.e., for which $dom(\tau') \nleq dom(\tau)$ holds). Then the scheduling system S is **noninterference secure** if the following predicate holds at any time $t \in \mathbb{N}$:

$$\begin{aligned} noninterference(S) := \forall \tau \in T. \forall t \in \mathbb{N}. \\ output(S)(\tau, t) = output(purge(S))(\tau, t) \end{aligned}$$

Noninterference states that whatever sequences of actions higher classified threads τ' execute, their behaviour as seen by τ is identical to a system in which these threads do not act at all.

Note that for this predicate to hold, actions of these higher classified threads may not change the points in time at which threads with a lower security class than τ execute. Thus, they may only act during the *holes* reported by the output function (i.e., at those points in time when the output function returns $-$). In fact they have to act such, that the occurrence of holes does not vary over time. Our proposed modification is to treat active, blocked or early stopping threads that may possibly leak information as if they were ready. This ensures that during those times when *output* returns $-$, either the original thread runs if an appropriate run action is contained in its action sequence or the idle thread runs if the original thread blocks or stops early. In the latter case, the idle thread, which runs effectively at the original thread's priority, ensures that future visible actions and holes happen at the same points in time.

Purging the actions of a thread but not the thread itself results in this thread stopping early immediately when it is released. The scheduler will thus select the idle thread whenever in the original schedule *output* returns $-$.

[1] The authors are aware that fuzzy time [5] successfully reduces the bandwidth of scheduling-related covert channels. We maintain this assumption for two reasons: Firstly, the selected noninterference property cannot be proven while some information is leaked over a covert channel. Secondly, some real-time applications require precise time stamps and precisely triggered events. Precise clocks are a precondition for both unless dedicated hardware is available for this purpose (e.g., capture-compare units).

3.2 Information Flow by Altering Thread Scheduling Behaviour

Varying the blocking and running behaviour of a high-priority thread directly influences the times when low priority threads run. In the following, we investigate how a higher prioritised thread can signal information indirectly by directly influencing low-priority threads and how delaying preemptions may be exploited to leak information.

Surprisingly, information may be signalled indirectly through low-priority threads in spite of their trustworthiness not to forward timestamps in their messages.

3.2.1 *Indirect Influence*

A higher prioritised thread τ_h can make use of a lower prioritised thread τ_l to signal another thread τ_x (τ_x can have a higher or a lower priority than τ_h), provided τ_l is authorised to communicate to τ_x. This is because the messages sent by τ_l may carry timestamps and these can be directly influenced by τ_h blocking or running.

Even if we would trust τ_l not to send timestamps, timing information may be leaked from τ_h to τ_x because τ_h is able to influence the ordering in which messages arrive at τ_x.

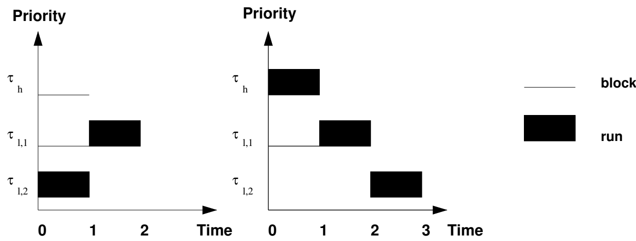

Figure 2: The high-priority thread τ_h causes a different execution order of the threads $\tau_{l,1}$ and $\tau_{l,2}$ depending on whether it blocks or runs. Even tough we trust $\tau_{l,1}$ and $\tau_{l,2}$ not to send explicit timing information, information about τ_h's behaviour can be deduced from the order in which messages arrive.

Figure 2 shows an example in which τ_h influences the execution order of the threads $\tau_{l,1}$ and $\tau_{l,2}$. Assume in a given period $\tau_{l,1}$ blocks for one unit of time, then executes for one unit. If τ_h blocks at time 0, $\tau_{l,2}$'s messages arrive at the receiver τ_x (not shown in the Figure) before $\tau_{l,1}$'s messages; if τ_h runs at time 0, the messages arrive in reverse order.

This scenario shows that despite $\tau_{l,1}$ and $\tau_{l,2}$ being trusted not to send timestamps in their messages, information may be leaked from τ_h as long as the communication channel used by the low threads reveal the order in which messages arrive.

3.2.2 *Influence through Delaying Preemptions*

Uniprocessor operating systems typically synchronise short critical sections by temporarily disabling device interrupts and other causes of preemptions. However, when preemptions are delayed, the thread causing the preemption remains blocked until interrupts are re-enabled.

To isolate the effect of malicious and erroneous threads we have to enforce a maximum duration in which preemptions are disabled, for example, similar to *delayed preemption* as implemented in the L4 microkernel [13, 3]. Let max_delay_i be the maximum time by which a thread τ_i may delay a single preemption.

A running low-priority thread τ_l knowing when a high-priority thread τ_h unblocks can alter delaying and not delaying this preemption to signal information to τ_h. The high-priority thread τ_h can, in turn, detect these delays by reading the system clock immediately after it gets the CPU. Preemptions of even lower prioritised threads remain unaffected as they are deferred until τ_l stops running anyway.

4. A NONINTERFERENCE-SECURE SCHEDULER

To prevent unauthorised leakage of information, we modify the operating-system scheduler to treat active threads that are blocked or that finished execution without exhausting their worst-case budgets as if they were ready. Each time the currently running thread blocks, becomes inactive or is preempted, our scheduling algorithm selects the highest-priority active thread

1. that is either ready

or that could potentially cause unauthorised information flows and

2. that is blocked
3. or that has finished without exhausting its worst-case execution budget or total worst-case budget.

In the first case, the scheduler switches to the selected thread. In the second and third case, the scheduler cannot switch to the selected thread because it is not ready. Instead, we introduce a **countermeasure** to prevent information leakage: the scheduler switches to the system's idle thread, which we trust not to send any messages and which we can therefore assign the ultimately highest security class $\top$. The time the system idles because of this countermeasure is accounted to the thread selected by the scheduler.

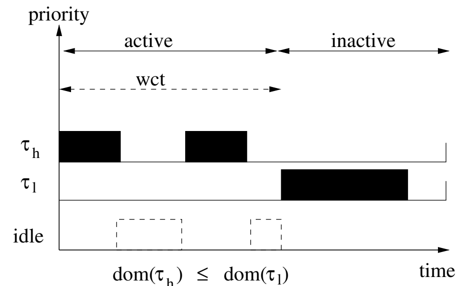

Figure 3: To prevent information leakage, our modified scheduler prevents τ_l from running during the times τ_h blocks or finishes early.

Figure 3 illustrates this algorithm. The system idles while τ_h is active and not running. As a consequence, the lower prioritised thread τ_l can no longer distinguish whether it did not run because the high-priority thread τ_h did run or because τ_h did block or stop early and the scheduler switched to the idle thread. The view of τ_l on τ_h therefore remains unchanged in spite of alterations in τ_h's behaviour.

In the following, we state more precisely the predicate when this countermeasure must be applied. To minimise the scheduling overhead, we prefer predicates that can be

checked statically while the system is off-line or during the admission. With static predicates, our modified scheduler achieves a near zero scheduling overhead compared to an unmodified fixed-priority scheduler which enforces worst-case times. This is because the check for the second case degenerates to checking an additional flag that can be stored in the thread-control block. The scheduler will then either switch to the selected thread or to the idle thread depending on this flag

4.1 Transitive Policies

As we have argued in Section 3.2, an active high-priority thread τ_h can directly influence a lower prioritised thread τ_l that is active at some point in time when τ_h is active or by influencing some intermediate-priority thread τ_x that is active when τ_l is active and which communicates with τ_h.

In the latter case, information is only leaked if τ_l wants to run and can do so because τ_h (respectively τ_x) blocks. However, we cannot consider thread actions because we search for a predicate that can be statically determined.

Therefore we choose the conservative but statically computable predicate $p_{trans}(\tau_h)$ that is true if and only if there is a lower prioritised thread τ_l to which τ_h is not authorised to send[2]. More formally:

Definition 2. Predicate for Transitive Policies.

$$p_{trans}(\tau_h) := \exists\ \tau_l \in T_{low}(\tau_h). \\ dom(\tau_h) \not\leq dom(\tau_l)$$

It is easy to see how the countermeasure with this predicate prevents information leakage to low-priority threads τ_l: if such a thread exists, no lower prioritised thread than τ_h (except the idle thread) is scheduled until τ_h's budgets are depleted or the deadline passes. To see how p_{trans} prevents a thread τ_h from indirectly influencing another thread τ_x, we have to consider two cases: $p_{trans}(\tau_h)$ and $\neg p_{trans}(\tau_h)$.

Case $p_{trans}(\tau_h)$: No thread τ_l with $prio(\tau_l) < prio(\tau_h)$ may be influenced directly by τ_h. Therefore, the timestamps these threads τ_l may report to τ_x are independent of τ_h's behaviour.

Case $\neg p_{trans}(\tau_h)$: τ_h may directly influence all lower prioritised threads (e.g., τ_l) and indirectly those threads to which these lower prioritised threads are authorised to send (e.g., τ_x with $dom(\tau_l) \leq dom(\tau_x)$). But then $dom(\tau_h) \leq dom(\tau_x)$ holds because of $\neg p_{trans}(\tau_h) \Rightarrow dom(\tau_h) \leq dom(\tau_l)$ and the transitivity of $\leq$.

4.2 Intransitive Policies

In intransitive policies, $c_1 \leq c_2 \wedge c_2 \leq c_3 \Rightarrow c_1 \leq c_3$ does not necessarily hold for all security classes $c_i \in C$. The intuition of intransitive information-flow policies is to authorise communication between two threads only if this communication happens through a dedicated third thread. It is up to this third thread to appropriately filter the information flow. An example for such a third thread is a crypto-gateway, that is, a server we trust to encrypt messages from the sender before relaying them to the receiver, thereby protecting the confidentiality of these messages.

[2]We do not claim our predicates to be minimal in the sense that they are the least restrictive predicates that can be statically computed. Further predicates remain to be investigated.

The static predicate $p_{trans}(\tau_h)$ is not sufficient to prevent indirect influences of other threads when the security policy is intransitive. Consider the case $\neg p_{trans}(\tau_h)$ in Section 4.1. In this case τ_h is authorised to send to all lower prioritised threads. Assume τ_l is the crypto gateway to a thread τ_x with which τ_h must not communicate directly. Because of $\neg p_{trans}(\tau_h)$, τ_h may directly influence τ_l's timing. As we have seen in Section 3.2.1, this direct influence of τ_h is sufficient to signal information to τ_x even if we trust τ_l not to leak timestamps with its messages. In fact, τ_l has no means to prevent this information leakage.

One possibility to address intransitive information flow policies is to choose a more restrictive predicate. For example,

Definition 3. Predicate for Intransitive Policies.

$$p_{intrans}(\tau_h) := \exists \tau \in T.\ dom(\tau_h) \not\leq dom(\tau)$$

However, this predicate would result a system in which the scheduler applies the countermeasure to all but those threads having the lowest security class ($dom(\tau) = \bot$). The resulting system would, in practice, be no better than time partitioning.

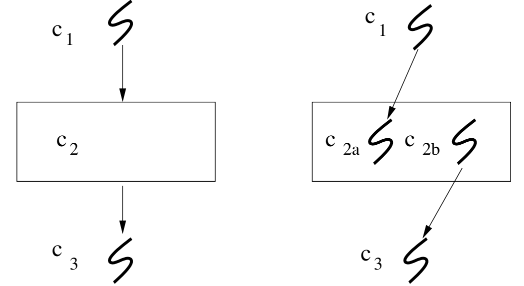

Figure 4: The crypto gateway decomposed into two threads with security levels c_{2a}, c_{2b}. The intransitive security policy $c_1 \leq c_2 \wedge c_2 \Rightarrow c_3$ is split into the transitive policy with $c_1 \leq c_{2a}$ and $c_{2b} \leq c_3$.

For this reason, we propose an alternative solution. Notice that the threads which mediate the communication in an intransitive policy must be trusted not to relay information arbitrarily. Thus, it is likely that the code executed by these gateway threads is sufficiently manageable to undergo a stringent security evaluation. This manageability gives us confidence that the following restructuring into a multi-threaded server is feasible.

Instead of accepting and relaying messages in the same thread, which would ultimately be subject to a direct influence by τ_h if this thread has a lower priority, we split this functionality into two threads and assign them distinct security levels. Figure 4 illustrates this approach. The two server threads are assigned distinct security levels c_{2a} and c_{2b} instead of c_2 from the original information flow policy. Consequently, the information flow policy becomes transitive at this point and the predicate $p_{trans}(\tau_h)$ would hold for the message relaying thread since $dom(\tau_h) \not\leq c_{2b}$, provided this relaying thread has a lower priority than τ_h. Obviously, to prevent illegal information flows, the internal communication between the two server threads has to be imple-

mented carefully, e.g., exploiting previous results from the NRL Pump [8].

4.3 Delayed Preemption Leaks

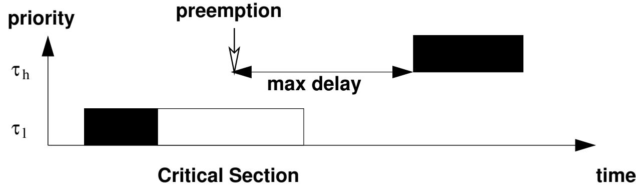

Figure 5: Countermeasure to prevent information leakage from delaying preemptions.

To prevent information leakage from delaying preemptions, a second modification to the scheduler is required. When delaying a higher prioritised thread's preemption could possibly leak information, the modified scheduler delays all preemptions caused by this higher prioritised thread τ_h by $max_delay_low(\tau_h)$:

Definition 4. Maximum Preemption Delay.

$$max_delay_low(\tau_h) := \max_{\tau_l \in T_{low}(\tau_h)} (max_delay(\tau_l))$$

The following static predicate determines when this countermeasure has to be applied.

Definition 5. Predicate for Delayed Preemption.

$$p_{delay}(\tau_h) := \exists\ \tau_l \in T_{low}(\tau_h).\ dom(\tau_l) \not\leq dom(\tau_h)$$

Because the high-priority thread τ_h controls how often it blocks and because each time it blocks may lead to delaying τ_h, we have to account the time between the high-priority preemption occurring and τ_h running to this high-priority thread. Figure 5 illustrates this countermeasure.

5. PROOF OF NONINTERFERENCE

In the following section, we sketch the formal proof of the noninterference property (Definition 1). The countermeasure to prevent information flow due to nonpreemptibility has not yet been included in the proof.

We formalised our system in PVS [17]. The sources are available at [26]. However, to illustrate our proof we use a more mathematical notation. Let $s.actions$ denote the element $actions$ of a record s. We write $s \setminus actions := x$ to update this field in the record with x, leaving the not mentioned fields unchanged.

We model the scheduler as a sequence of state-transformers that perform the individual transitions of a fixed-priority scheduler at each clock tick. When no such transition occurs at an individual clock tick the thread state remains the same and only the ticks in the remaining budgets are adjusted.

The model deviates from an implementation in a real system in two points:

- Instead of checking at each clock tick whether a budget has depleted, a real-system implementation would program a timer interrupt to trigger when the scheduler is to be invoked next.
- A real-system implementation would maintain a linked lists of ready threads — the ready queue — to avoid searching the array of existing threads.

Otherwise, a real-system implementation has to perform the same state-transitions.

Notice that while the enforcement of worst case execution budgets *wcet* is relevant for preserving the real-time guarantees, with regards to information leakage only the enforcement of the total worst case budgets *wct* is relevant. A thread which exceeds its *wcet* and which is unconstrained by our countermeasure may prevent lower prioritised threads from executing for a longer time but this prevention can legitimately be seen. On the other hand, threads constrained by the countermeasure will prevent this execution anyhow until *wct* is depleted because the idle thread runs whenever the original thread does not. For this reason we omit *wcet* from the proof of noninterference and consider only *wct* budgets.

The state-transformers are:

deadline step set a thread to *inactive* when the deadline has passed or the budget is depleted

release step activate a thread τ_i at every release point $\pi_{i,k}$ and refill the budget to $wct_{i,k}$

end action determine whether the current action of the thread has stopped

next action adjust the thread state according to the thread's next action.

More formally, let s be the record *state* which contains besides the above scheduling parameters for each thread τ a dynamic state comprised of its current release $s.release(\tau) = k$ and for this release the remaining time $s.rem_time(\tau)$ that the time the current action lasts, the remaining worst case budget $s.rem_wct(\tau)$, a list of actions which remain to be executed in this release period $srem_actions(\tau)$ and a thread state $s.ts(\tau) \in \{blocked, ready, cm_blocked, inactive\}$ (a thread τ is in *cm_blocked* if it is *blocked* and $p_{trans}(\tau)$ holds).

We obtain the actions from the per thread function $s.actions$ which records for each release $k \in \mathbb{N}$ the trace (list) of actions this thread will perform. In the proof we consider arbitrary action sequences and compare for each thread τ the output of this system with the output of an identical system in which this action list is purged for all threads from which τ must not receive. Consequently *purge* is:

Definition 6. Purge.

$$purge(s, \tau) := s \setminus actions := \lambda k \in \mathbb{N}, \tau' \in T. \begin{cases} \langle\rangle & \textbf{if}\, dom(\tau') \not\leq dom(\tau) \\ s.actions(k, \tau') & \textbf{otherwise} \end{cases}$$

where $\langle\rangle$ denotes the empty list.

We define *output* by adding to the state a field $s.event$ which records for each clock tick t the highest priority thread which executes during this tick. Thus *output* is:

Definition 7. Output.

$$output(s, \tau, t) := \begin{cases} s.event(t) & \textbf{if}\, dom(s.event(t)) \leq dom(\tau) \\ - & \textbf{otherwise} \end{cases}$$

The above state transformers have the form:

$$State \times \mathbb{N} \times T \rightarrow State$$

and are completely formalised in the PVS sources [26]. Due to space limitations we present here only an extract of the *end_action* transformer.

Definition 8. End Action.

$$
\begin{array}{l}
\mathit{end_action}(s,t,\tau) := \\
\quad \textbf{If } s.ts(\tau) \neq \mathit{inactive} \ \wedge \ s.\mathit{rem_time} = 0 \ \wedge \\
\quad\quad s.\mathit{rem_actions}(\tau) \neq \langle\rangle \\
\quad \textbf{Then} \\
\quad\; \textbf{Cases } s.\mathit{rem_actions}(\tau) \textbf{ Of} \\
\quad\quad \mathit{run}(\mathit{time_span}) : \\
\quad\quad\; s \setminus ts(\tau) := \mathit{runnable}, \\
\quad\quad\quad \mathit{rem_actions}(\tau) := \mathit{tail}(s.\mathit{rem_actions}(\tau)) \\
\quad\quad\quad \mathit{rem_time}(\tau) := \mathit{time_span}, \\
\quad\quad \mathit{block}(\mathit{time_span}) : \\
\quad\quad\; s \setminus ts(\tau) := \begin{cases} \mathit{cm_blocked} & \textbf{if} p_{trans}(\tau) \\ \mathit{blocked} & \textbf{otherwise} \end{cases} \\
\quad\quad\quad \mathit{rem_actions}(\tau) := \mathit{tail}(s.\mathit{rem_actions}(\tau)) \\
\quad\quad\quad \mathit{rem_time}(\tau) := \mathit{time_span} \\
\quad \textbf{Else} \\
\quad\; \ldots \\
\quad \textbf{Endif}
\end{array}
$$

where t is the point in time for which *end_action* should be evaluated.

If the current action of this thread has finished ($s.\mathit{rem_time}(\tau) = 0$), the presented part of *end_action* selects the following action in $s.\mathit{rem_actions}(\tau)$(provided more actions remain for this period) and depending on this action adjusts the thread state accordingly. In particular we set the thread to *cm_blocked* if the thread blocks and $p_{trans}(\tau)$ holds.

The state transformers are invoked for each thread in the above order (i.e., $\sigma := \mathit{deadline} \circ \mathit{release} \circ \mathit{end_action} \circ \mathit{next_action}$) for each thread in the recursively defined state transformer *dispatch_step*:

Definition 9. Dispatch Step.

$$
\begin{array}{l}
\mathit{dispatch_step}(s,t,\tau_i) := \\
\begin{cases} \sigma(s,t,\tau_i) & \textbf{if } i = 0 \\ \sigma(\mathit{dispatch_step}(s,t,\tau_{i-1}),t,\tau_i) & \textbf{otherwise} \end{cases}
\end{array}
$$

This step is in turn invoked recursively for each point in time starting from an initial state s_0:

Definition 10. Dispatch.

$$
\begin{array}{l}
\mathit{dispatch}(s_0,t) := \\
\begin{cases} \mathit{dispatch_step}(s_0,t,\tau_{\max}) & \textbf{if } t = 0 \\ \mathit{dispatch_step}(\mathit{dispatch}(s_0,t-1),t,\tau_{\max}) & \textbf{otherw}. \end{cases}
\end{array}
$$

Thus, our main theorem stating confidentiality is

Theorem 1. Confidentiality.

$$
\begin{array}{l}
\forall \tau_i \in T, t \in \mathbb{N}, s_0. \\
\quad \mathit{output}(\mathit{dispatch}(s_0,t),\tau_i,t) = \\
\quad\; \mathit{output}(\mathit{dispatch}(\mathit{purge}(s_0,\tau_i),t),\tau_i,t)
\end{array}
$$

The proof is straightforward with help of the following predicate and by induction over all points in time t and over all thread indices i. We proved this predicate to be an invariant of our scheduler with a similar induction over t and i:

Definition 11. Same High State.

$$
\begin{array}{l}
\mathit{same_high_state}(s,s_p)(\tau) := \quad \forall \tau' \in T_{high}(\tau). \\
\quad \textbf{If} p_{trans}(\tau')\textbf{Then} \\
\quad\quad s.\mathit{rem_wct}(\tau') = s_p.\mathit{rem_wct}(\tau') \wedge \\
\quad\quad (s.ts(\tau') = \mathit{ready} \vee s.ts(\tau') = \mathit{cm_blocked}) \Leftrightarrow \\
\quad\quad\; (s_p.ts(\tau') = \mathit{ready} \vee s_p.ts(\tau') = \mathit{cm_blocked}) \wedge \\
\quad\quad s.ts(\tau') \neq \mathit{blocked} \wedge s_p.ts(\tau') \neq \mathit{blocked} \\
\quad \textbf{Else} \\
\quad\quad \mathit{dyn}(s,\tau') = \mathit{dyn}(s_p,\tau') \\
\quad \textbf{Endif}
\end{array}
$$

where s_p is the purged state of s and $dyn(s,\tau)$ denotes the entire dynamic state of τ (i.e., $s.ts(\tau)$ plus all the remaining times and actions $s.rem_*$).

Definition 11 formalises the following proposition on the dynamic thread state of those threads τ' which have a higher priority than τ: Provided $\neg p_{trans}(\tau')$ holds, the dynamic state is the same in the original state s and in the purged state s_p. Otherwise, if the countermeasure predicate $p_{trans}(\tau')$ holds, at least the remaining *wct* is the same in s and in s_p and either in both states or in none of the two, the scheduler considers these threads as if they were ready (i.e., $s.ts(\tau') = \mathit{ready}$ or $s.ts(\tau') = \mathit{cm_blocked}$; $s_p.ts$ respectively).

It follows that threads for which this predicate holds produce the same *output*. Thus, our main theorem holds because this predicate is an invariant.

6. TIMELY ISOLATION OF NONINTERFERING THREADS

A consequence of the arguments we gave in the context of unauthorised information flows is that in our system, the precise points in time when a thread τ may run cannot be affected by the actions of a thread τ from which τ must not receive information (i.e., for which $dom(\tau) \not\leq dom(\tau)$ holds). Our system therefore isolates time wise the thread τ from the thread τ'.

Commercial time-partitioning systems such as LynxOS [16] typically implement a hierarchical fixed-priority scheduler inside their partitions to support Posix threads. To achieve the above kind of timely isolation between τ and τ' they do, however, rely on the underlying partition scheduler and schedule both threads in different partitions. With our solution, this hierarchical approach is no longer necessary because our countermeasure can be used to timely isolate threads directly with the fixed-priority scheduler.

7. REAL-TIME GUARANTEES

All previously reported results hold for arbitrary thread sets that are scheduled on top of a budget-enforcing fixed-priority scheduler. In the following discussion on preserved real-time guarantees, we restrict ourselves only to strictly periodic thread sets. We will partially lift this restriction in Section 8.

Remember, strictly-periodic threads are those with equidistant release points $\pi_{i,k}$ and identical budgets for each release. In the following, we omit the release index k. Furthermore, let $\Pi_i = \pi_{i,k+1} - \pi_{i,k}$ for all k.

In a real-time system, it is crucial that all hard real-time threads meet their deadlines. For this reason, an admission test is performed which decides before the thread set

is executed whether each thread will meet all its deadlines. Probably the most popular result is Liu and Layland's criterion [14] which says that a set of n periodic threads can be scheduled by the rate monotonic (RM) policy if

$$\sum_{1 \leq i \leq n} \frac{wcet_i}{\Pi_i} \leq n \cdot (2^{\frac{1}{n}} - 1).$$

Here is assumed that $\Pi_i = d_i$ for all i, that all the threads never block, and the threads are ordered by increased periods. Lehoczky et al. proposed the time-demand analysis method [12] that provides a sufficient and necessary schedulability test. On the other hand, Sha et al. [23] included blocking times showing that the thread set is schedulable by the RM priority assignment if

$$\forall k \in \{1, ...n\}. \sum_{1 \leq i \leq k} \frac{wcet_i}{\Pi_i} + \frac{b_k}{\Pi_k} \leq k \cdot (2^{\frac{1}{k}} - 1)$$

where b_k is an upper bound on the duration of blocking that the kth thread may experience due to resources held by lower priority threads. In this section, we follow Liu [11] to determine the different blocking times depending on the different blocking reasons.

In addition to the usual influence of high-priority threads on the timing behaviour of lower-prioritised threads, our modified scheduler may prohibit ready lower-prioritised threads from running because it switches to the idle thread to avoid information leakage. In the worst case, a thread τ_l is prohibited from running during each blocking time b_h of all higher prioritised threads τ_h for which the countermeasure is applied. We call this time the "prohibition time" $b_l(pr)$ of a thread τ_l. It holds:

Equation 1. Prohibition Times.

$$b_l(pr) = \sum_{\tau_h \in T_{p_{trans},high}(\tau_l)} \lceil \frac{\Pi_h}{\Pi_l} \rceil b_h$$

with $T_{p_{trans},high}(\tau_l) = \{\tau_h \in T_{high}(\tau_l) | p_{trans}(\tau_h)\}$ as introduced in Section 4.1.

Obviously, prohibiting threads from running increases the idle time. We quantify the increase in idle time by the prohibition time of the lowest-prioritised thread $b_{idle}(pr)$.

To assure that the thread set remains schedulable on the modified scheduler, we reuse a common admission test for an unmodified scheduler and take into account the prohibition times as an additional blocking term. In some situations, a thread set must be rejected when considering prohibition times that would otherwise have been accepted by an acceptance test for an unmodified scheduler.

Consequently, this results in a lower utilisation compared to the utilisation of this unmodified scheduler U_{orig}. Precisely, the maximum utilisation which can be achieved due to our modification is $U = U_{orig} - \frac{b_{idle}(pr)}{\Pi_{idle}}$.

Now we investigate how well-behaving real-time threads are constrained by our countermeasures. For this, we compare an admission with and without prohibition times and relate these results to a system that always enforces worst-case behaviour (e.g., a time-partitioning scheduler [3]).

[3] A time-partitioning scheduler computes for each thread τ_i fixed, recurring and nonoverlapping time intervals of size $wcet_i + wcbt_i$. Because no two threads are active at the same time, the blocking times of one thread cannot be used to run another thread.

7.1 Blocking due to Self Suspension

Threads may block due to self suspension (e.g., in a blocking systemcall when they wait for the completion of asynchronous I/O). We assume that an upper bound x_i on the time a thread τ_i blocks due to self suspension is known a-priori.

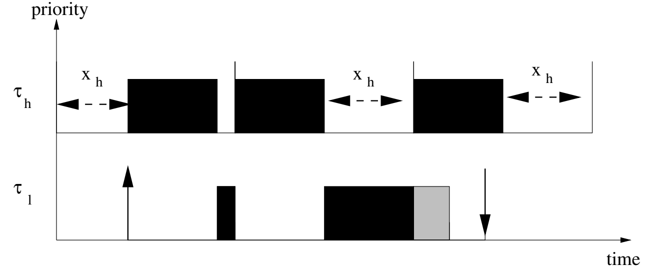

Figure 6: Blocking due to self suspension. τ_l misses its deadline because the gray-shaded part does not fit.

Liu [15] (pg. 165 ff) gives an upper bound $b_l(ss)$ on the blocking time due to self suspension:

Equation 2. Blocking due to Self Suspension.

$$b_l(ss) = x_l + \sum_{\tau_h \in T_{high}(\tau_l)} \min(wcet_h, x_h)$$

We illustrate this formula using the example in Figure 6. A low-priority thread τ_l may consume the time the higher prioritised thread τ_h suspends itself. For example, τ_l runs in τ_h's second period independent of when τ_h suspends itself during this period. The blocking time origins from τ_h delaying its execution prior to the period of τ_l. Consequently τ_h consumes additional time in the interval of τ_l's period. This additional time is at most the minimum of the worst-case execution time of τ_h and its maximum self-suspension time x_h. The latter case is shown in the figure.

Because malicious and erroneous threads can signal information both by the amount of time they self suspend and through the point in time when they self suspend we have to prevent low-priority threads from running during the entire time in which such a thread could run or suspend itself. Attributing the prohibition times of these threads to the blocking time due to self suspension, we get the blocking time $b_l^{cm}(ss)$ for our countermeasure.

Equation 3. Blocking Time Self Suspension Countermeasure

$$b_l^{cm}(ss) = x_l + \sum_{\tau_h \in T_{\neg p_{trans},high}(\tau_l)} \min(wcet_h, x_h) + \sum_{\tau_h \in T_{p_{trans},high}(\tau_l)} \lceil \frac{\Pi_h}{\Pi_l} \rceil x_h$$

The second sum in this formula is the prohibition time $b_l(pr)$ if we consider self suspension as the only reason for blocking.

Because the maximum self-suspension time must be considered for every period of higher prioritised threads rather than only once and in the minimum with *wcet*, an admission test for an unmodified scheduler could accept more thread sets.

Compared to a scheduler that always enforces the worst-case behaviour, our approach accepts more thread sets because the prohibition times need only to be considered for

those threads for which the countermeasure must be applied. Threads authorised to send to other (respectively lower-prioritised) threads remain unaffected. A realistic example in which this situation arises are real-time drivers (e.g., disk, network, sensor, etc.) that read confidential data only in its encrypted form. Such a real-time driver can be classified with the lowest security class $\perp$. As these drivers typically have short periods, rate monotonic scheduling would assign them a high priority. While the modified scheduler leaves the driver unrestricted, a time-partitioning scheduler must execute it in a separate partition of the length $wcet_i + x_i$ to prevent leakage from secret applications to the low-classified driver. A time-partitioning scheduler can therefore not exploit the self-suspension time to schedule additional threads.

7.2 Blocking due to Nonpreemptibility

To determine the maximum blocking time due to nonpreemptibility we need to know how many times K_h a thread τ_h suspends itself after it starts and the duration of the largest nonpreemptible critical section $max_delay_low(\tau_h)$ (Definition 4).

Thus, the blocking time due to nonpreemptibility is [15]:

Equation 4. Blocking Time due to Nonpreemptibility.

$$b_h(np) = (K_h + 1)\ max_delay_low(\tau_h)$$

A thread τ_h for which we also apply our first countermeasure of switching to the idle thread (i.e., for which $p_{trans}(\tau_h)$ holds) can be blocked due to nonpreemptibility only before it starts. Thus we get:

Equation 5. Blocking Time due to Nonpreemptibility Countermeasure.

$$b_h(np) = \begin{cases} max_delay_low(\tau_h) & if\ p_{trans}(\tau_h) \\ (K_h + 1)\ max_delay_low(\tau_h) & otherwise \end{cases}$$

In theory, our approach could accept more thread sets than the unmodified scheduler when $K_h \cdot max_delay_low(\tau_h)$ is large compared to the self suspension time x_h. In practice, nonpreemptible critical sections are short and this effect could not be seen. Still an admission for our modified scheduler performs no worse compared to an admission for a classical scheduler as far as blocking due to nonpreemptibility is concerned.

8. PRACTICAL CONSIDERATIONS

In the following, we shortly address resources, precedence constraints as well as aperiodic and sporadic threads.

8.1 Resources

A complete investigation of blocking due to resource limitations is out of the scope of this paper. We only investigate the consequences of our countermeasure on threads that block on a resource. We do not consider illegal information flow due to the resource allocation.

In Section 7.2 we discussed a solution for nonpreemptible critical sections. Here we focus on situations where the holder of a resource or critical section can be preempted.

When resource holders can be preempted in their execution, higher prioritised threads may fail to immediately lock the resource. In this situation, a higher prioritised thread τ_h can suspend itself to give the low-priority resource holder τ_l the chance to free this resource. As shown in Figure 7, this

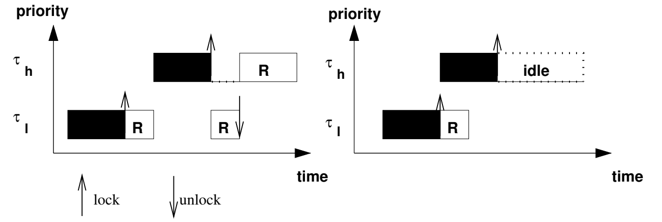

Figure 7: Self suspension of τ_h allows τ_l to release the resource unless the countermeasure prevents τ_l from running.

is no longer possible when the countermeasure switches to the idle thread instead. Not having investigated the security of resource allocation protocols yet, we propose to use nonpreemptible critical sections instead.

In a multiprocessor system, nonpreemptible critical sections and self suspension can be combined to implement preemption-aware critical sections [9]. Whenever the resource is held by a thread on another processor a requesting thread suspends itself so that other threads could be scheduled on its CPU when the countermeasure is not applied. With acquiring the resource, the thread signals the scheduler not to preempt it. Thus, the resource is either free or held on another processor and the switch to the idle thread does not hinder any thread to free the resource. Figure 8 illustrates this locking scheme.

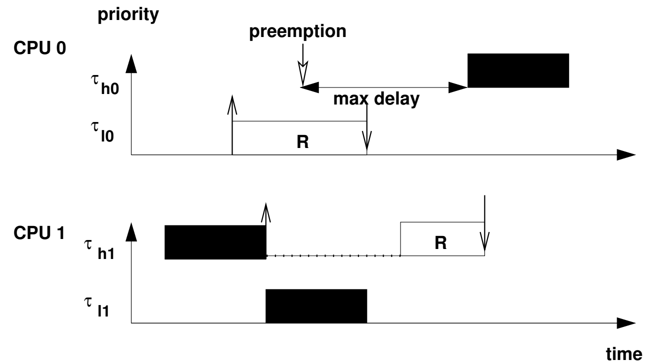

Figure 8: A preemption-aware multiprocessor lock.

8.2 Precedence Constraints

In a practical system, threads are usually not independent from each other, because they rely on the results produced by other threads. These dependencies are typically denoted by a precedence graph which states which threads must complete before a later thread in the directed graph can sensibly be scheduled.

Considering dynamic dependencies, as they arise for example when requesting resources, is out of the scope of this paper. However, precedence due to data dependencies can be computed statically. We exploit effective release times which we compute in a slightly different way and activate a thread τ only after all those threads have produced their results on which τ depends.

The effective release time of a thread τ is therefore set to the latest response time of the threads on which τ depends. Figure 9 illustrates this procedure.

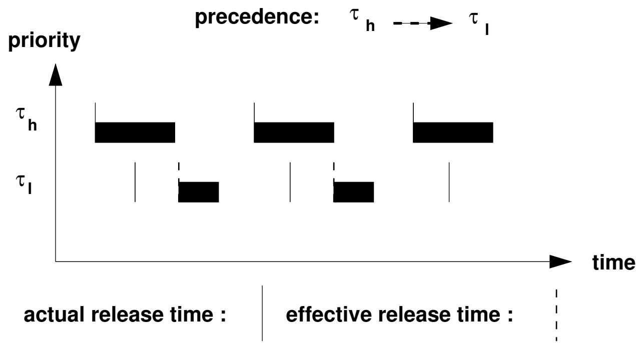

Figure 9: Adjustment of effective release times to reflect precedence constraints.

Because the precedence constraints are public information and because the response times are computed based on worst-case execution and blocking times, the resulting adjusted schedule reveals no additional information.

8.3 Aperiodic and Sporadic Threads

Sporadic servers and deferrable servers [15] are means to increase the response time of sporadic and aperiodic threads. These are threads which have varying period lengths respectively soft or no deadlines. The admission plans these servers as normal real-time threads. The scheduler then exploits the time and priority planned for these servers to execute sporadic and aperiodic threads after they have arrived.

If the arrival of a sporadic or aperiodic thread can be considered as public information, we can treat the sporadic server or the deferrable server as a thread that does not execute until a respective sporadic or aperiodic thread arrives. This event is the server's first release point. Consequently, the budgets of these servers need to be considered only after this release point.

The server inherits the information flow properties of the arrived thread and we apply the countermeasure if the respective predicate holds for the arrived thread. Because, in general, we do not know in advance whether the countermeasure predicate holds for the arriving thread, the admission has to treat the server as potentially leaking. Alternatively this decision can be made dynamically when reserving distinct deferrable servers for possibly leaking and not leaking threads.

The same line of arguments holds for threads that are suspended right before they start execution (e.g., when they await an external event). If occurrence of these events is public information, we need not to switch to the idle thread for this portion of the self-suspension time. Instead we adjust the release point of these servers to the point in time at which the event arrives.

Sporadic or deferrable servers can be used to implement dynamic thread creation by executing the created threads as payload of these servers.

9. RELATED WORK

The scheduling covert channel has been addressed previously. We first elaborate on related approaches which modify the system scheduler to close or to mitigate this channel. Then we briefly sketch alternative approaches to prevent threads from leaking information through their external timing behaviour.

Son et al. [25] analyse the impact of rate-monotonic scheduling on covert timing channels. The authors formalise and classify covert timing channels inherent with RM scheduling. They conclude the RM-channel to be positive deducible, which means in some settings the covert timing channel cannot be closed. Furthermore, they present a measure for the capacity of this channel. Our approach closes this timing channel by forcing the possibly leaking senders into their worst-case behaviour.

Time-partitioned systems [10], as defined for example in the Arinc 653 standard [1], assign each thread a partition and schedule threads hierarchically when their respective partition is active. The partitions themselves are scheduled according to an off-line-computed clock-driven schedule. Time-partitioned systems are noninterference secure because they prevent threads in different partitions affecting each other's timing. In Section 7.1 we identify cases in which an admission for our approach can accept more thread sets. In addition we achieve a higher performance for nonreal-time threads because only those threads have to be restricted that could possibly leak information. In the special situation when unauthorised information flows are only from lower to higher prioritised threads, our approach leaves the system unrestricted.

Hu [6] proposes the lattice scheduler to minimise the times when countermeasures (e.g., cache flushing) have to be applied. The lattice scheduler allocates for each security class a time partition (called time slot) and schedules threads of this security class hierarchically whenever the respective time slot is active. In contrast to a traditional time-partitioned system, the lattice scheduler allows higher classified threads to consume part of the remaining in a time slot. In the case when all threads of the security class that was active in the current time slot have stopped early, the lattice scheduler selects a higher security class to consume the remaining time in this slot. No information is leaked because only higher classified threads are activated when lower classified threads stop early. The lattice scheduler inherits most of its real-time properties from time-partitioning schedulers. However, because higher classified threads may consume the remaining time of early stopped threads it may in general accept more thread sets. In contrast to our solution, blocking times in the middle of a thread's execution cannot be consumed by threads of a different security class.

Boucher et al. [2] propose a best-effort scheduler that selects threads according to their time-value function. This function describes how valuable it is to schedule the thread at a given point in time. In addition, the scheduler continuously measures covert-channel bandwidth and when this bandwidth reaches a certain threshold, it switches to a scheme similar to lattice scheduling. A control policy can then trade covert-channel bandwidth against real-time performance by dynamically adjusting this threshold when an important real-time thread risks missing its deadline. The main difference compared with our approach is that the real-time admission must be changed to take into account when the covert-channel bandwidth has reached the threshold. Their approach has a considerably higher overhead than ours because they have to evaluate the information-flow policy online. The use of static predicates relieve us from this necessity.

Other approaches to prevent information leakage caused

by variations in a thread's external timing behaviour include fuzzy time [5] and security type systems.

Hu et al. [5] adds noise to timing sources and event delivery. Lacking a precise timing source it becomes more difficult to signal information by varying the timing behaviour of a thread. As illustrated beforehand, fuzzy time cannot close but only mitigate timing channels. Furthermore, real-time threads require some degree of precise timing.

A large body of work investigates type systems that assert confidentiality for a program that is scheduled under a specific class of schedulers (e.g., uniform [24] or probabilistic schedulers [22]) or that assert program confidentiality independent of the underlying scheduler [22].

Among these Russo et al. [21] propose a type system that allows threads to signal to the scheduler when only threads of the same security class should be scheduled. This type system asserts confidentiality of a multithreaded program with dynamically created threads running on top of a noninterferent scheduler. A more recent paper of these authors [20] describes a transformation in which a single threaded program is transformed into a multithreaded program such that each thread assigns only to variables of a single security class. They show that this transformation establishes noninterference for the lattice $low \leq high$ when executing the threads under a round-robin scheduler. Real-time was no concern in this work.

Acknowledgements

Thanks are due to Hendrik Tews, Michael Roitsch and to the anonymous reviewers for their comments and advices that helped us improve this paper. We further thank Intel Corp. for their support. This work is in part funded by the European Commission through PASR grant 004700.

10. CONCLUSIONS

We investigated illegal information flows through the scheduling subsystem by alterations in the scheduling related behaviour of threads and propose a rather simple countermeasure that closes these channels. A budget-enforcing fixed-priority scheduler is modified to treat active threads that block or that stop early and that could potentially leak information as if they were ready. While this increases the system's idle time, information can no longer be leaked via the scheduling subsystem and some threads become timely isolated from the behaviour of others.

We quantify the increase in idle time by the prohibition time of the lowest-prioritised thread $b_{idle}(pr)$. Because the prohibition time is an additional blocking factor, we showed how to reuse existing admission tests to determine the schedulablity of a given thread set. The utilisation we achieve compared to the respective original, unmodified test is $U = U_{orig} - \frac{b_{idle}(pr)}{\Pi_{idle}}$.

The decision when to apply our countermeasure we base on a static predicate. Thus, the additional scheduling overhead introduced with our approach is negligible.

In the near future, we plan to formally prove noninterference for a system with blocking due to nonpreemptibility. Additional directions of future research include dynamic-priority real-time schedulers, resources and reduction of system idle times due to less restrictive predicates.

11. REFERENCES

[1] ARINC. *ARINC 653-1 Standard.*

[2] P. Boucher, R. Clark, I. Greenberg, D. Jensen, and D. Wells. Towards a multilevel-secure, best-effort real-time scheduler. In *4th IFIP Working Conference on Dependable Computing for Critical Applications*, San Diego, CA, USA, Jan 1994.

[3] U. Dannowski, J. LeVasseur, E. Skoglund, and V. Uhlig. L4 eXperimental Kernel Reference Manual, Version X.2. Technical report, University of Karlsruhe, 2004. Latest version available from: `http://l4hq.org/docs/manuals/`.

[4] Z. Deng and J. Liu. Scheduling real-time applications in an open environment. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 308–319, Dec. 1997.

[5] W. Hu. Reducing timing channels with fuzzy time. In *IEEE Computer Society Symposium on Research in Security and Privacy*, Oakland, CA, USA, May 1991.

[6] W. Hu. Lattice Scheduling and Covert Channels. In *IEEE Symposium on Security and Privacy*, Washington, DC, USA, 1992.

[7] H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter. The Nizza Secure-System Architecture. In *First International Conference on Collaborative Computing: Networking, Applications and Worksharing*, San Jose, California, USA, Dec. 2005.

[8] M. Kang and I. Moskowitz. A Pump for Rapid, Reliable, Secure Communication. In *ACM Conference on Computer and Communication Security*, pages 119 – 129, Nov 1993.

[9] L. Kontothanassis, R. Wisniewski, and M. Scott. Scheduler Conscious Synchronization. *ACM Transactions on Computer Systems*, Feb. 1997.

[10] H. Kopetz. The time-triggered architecture. In *ISORC*, 1998.

[11] J. Lehoczky, L. Sha, and Y. Ding. The rate-monotonic scheduling algorithm: Exact characterization and average case behaviour. In *Real-Time Systems Symposium*, pages 166–171, Dec 1989.

[12] J. P. Lehoczky, L. Sha, and Y. Ding. The rate-monotonic scheduling algorithm: Exact characterization and average case behavior. In *Real-Time Systems Symposium*, pages 166–171, Dec 1989.

[13] J. Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain Resort, CO, Dec. 1995.

[14] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in an Hard-Real-Time Environment. *Journal of the ACM*, 20.1:46–61, Jan 1973.

[15] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[16] Lynuxworks. Lynxos: Partitioning operating systems vs. process-based operating systems. http://www.lynuxworks.com/products/whitepapers/partition.php.

[17] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated*

Deduction (CADE), volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[18] J. Regehr and J. A. Stankovic. HLS: A framework for composing soft real-time schedulers. In *RTSS '01: Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, Washington, DC, USA, 2001. IEEE Computer Society.

[19] J. Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-2, Stanford Research Institute, 1992.

[20] A. Russo, J. Hughes, D. Naumann, and A. Sabelfeld. Closing itnernal timing channels by transformation. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS'06)*, 2006.

[21] A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *19th IEEE Computer Security Foundations Workshop*, Venice, Italy, July 2006.

[22] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *CSFW '00: Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW'00)*, Washington, DC, USA, 2000. IEEE Computer Society.

[23] L. Sha, R. Rajkumar, and J. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronisation. *IEEE Transaction on Computers*, 39, 1990.

[24] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, San Diego, California, United States, 1998.

[25] J. Son and J. Alves-Foss. Covert timing channel analysis of rate monotonic real-time scheduling algorithm in mls systems. In *7th Annual IEEE Information Assurance Workshop*, West Point, NY, USA, June 2006.

[26] M. Völp, C. J. Hamann, and H. Härtig. Avoiding Timing Channels in Fixed-Priority Schedulers - PVS Sources. available from http://os.inf.tu-dresden.de/~voelp/sources/sec_rt_trans.tgz.