On Confidentiality-Preserving Real-Time Locking Protocols

Marcus Völp, Benjamin Engel, Claude-Joachim Hamann, Hermann Härtig
Department of Computer Science, Operating Systems Group
Technische Universität Dresden, Germany
Email: {voelp,engel,hamann,haertig}@os.inf.tu-dresden.de

***Abstract*—Coordinating access to shared resources is a challenging task, in particular if real-time and security aspects have to be integrated into the same system. However, rather than exacerbating the problem, we found that considering real-time guarantees actually simplifies the security problem of preventing information leakage over shared-resource covert channels. We introduce a transformation for standard real-time resource locking protocols and show that protocols transformed in this way preserve the confidentiality guarantees of the schedulers on which they are based. Through this transformation, we were able to prove that four out of the seven investigated protocols are information-flow secure.**

***Keywords*—covert channels, information-flow security, real-time systems, resources**

I. Introduction

Today, many safety-critical systems must not only meet soft and hard real-time requirements, they must also secure the data they process. Consider for example satellite-control systems or base stations with encrypted command channels. Unauthorized or accidental disclosure (i.e., leakage) of secret keys opens up possibilities for tampering with these devices, causing communication blackouts or damage to valuable equipment. Privacy is at risk if patient monitoring systems and other surveillance systems leak information about legitimately observed subjects to unauthorized entities. Virtualization, as it is used for example in smart phones to provide isolation between private and business/government use of the phone, does not help prevent information leakage unless all mutually exclusive shared resources conceal the resources' access patterns. What these examples show is that

1) real-time and security are no longer disjoint concerns but objectives that have to be fulfilled simultaneously and in the same system,
2) systems differ largely in the amount of leaked information they tolerate before security is at risk, and
3) security often hinges on the secrecy of only a few bits.

Channels giving rise to security-policy violations in the form of information leakage are called covert channels and a common mitigation strategy has been to reduce their bandwidth. However, point 3 shows that it is not very helpful for all systems [1]. Even at rates as low as one bit per second, keys recommended for pre-quantum symmetric and asymmetric cryptography [2] are leaked within two minutes respectively well within one hour. These times are for the disclosure of the complete key, ignoring that knowledge of parts of these keys already simplifies attacks on the ciphers.

This paper's concern is the unauthorized disclosure of secret information in the course of shared resource accesses. More precisely, we seek to prevent information leakage via resource locking protocols. In real-time systems, these protocols control access to mutual exclusive resources to prevent unbounded priority inversion and to guarantee bounded blocking times.

The possible ways to relay information via a shared resources can be classified as follows:

1) *storage channels:* Writers store secrets directly in the state of the resource. Storing part of the secret data in a field of a shared object, exhausting the storage space of a shared disk or altering the configuration register of some device in a secret dependent way are examples of this class of information leakages.
2) *protocol-independent timing channels:* Writers alter the resource such that it exhibits a different timing behavior for subsequent accesses. Positioning the head near the axis/edge of a disk drive by reading an inner/outer sector is an example of such a timing channel. It influences the seek time of subsequent disk accesses.
3) *protocol-related timing channels:* Writers encode secret information by altering when and for how long they prevent others from obtaining mutual-exclusive access to the resource. For example, releasing the shared resource immediately or after a measurable time when a higher prioritized task blocks on the resource influences the higher prioritized task's resource acquisition times and allows relaying secrets.

Clearly, because locking protocols are agnostic of the way tasks access resources, changing the protocol cannot prevent covert channels of the first kind. Instead, we have to rely on trusted tasks to mediate access to these resources and to sanitize the information they receive. In Section III, we shall see though that protocols can prevent the exploitation of channels of the second and third kind, if they can be reformulated as downward-donating protocols [3]. More precisely, they preserve the confidentiality guarantees of the scheduler on which they are based. Transforming seven classical and recent real-time locking protocols into downward-donating protocols, we were able to proof that four of these seven protocols are secure in the sense described above. Although information security adds another aspect to locking, we found that real-

time systems actually simplify the task of securing resource accesses.

II. Background

In real-time systems, locking protocols mediate access to shared resources to guarantee pre-runtime calculable bounds on resource access times. Typically, these protocols are formulated as a set of rules determining when resources are allocated to tasks and how tasks, which hold resources, are scheduled. Except for the protocol-related concerns (e.g., when to run the resource holding task instead of the scheduled one to limit priority inversion), these rules often match those of resource-agnostic schedulers such as fixed priority or EDF. We call these resource-agnostic schedulers the underlying schedulers on that the protocols are based. In Section III, we show that downward-donating resource locking protocols introduce no further means for leaking secrets provided their underlying schedulers are secure with regards to covert channels.

Information is *confidential* if it is disclosed only to authorized entities. In our setting, entities are tasks, which execute on behalf of their users. *Information-flow policies* define which entities are authorized to receive which information. A common formalization for the static part of these policies is the lattice model [4]. In this model, policies are characterized as triples $(L, \leq, dom)$, where L is a set of secrecy levels (containing for example low for public information and $high$ for secret information), $\leq$ is a relation between secrecy levels (e.g., $high \not\leq low$ denotes that no information must flow from $high$-classified tasks to low-classified tasks) and dom is a mapping of entities and information to secrecy levels. An entity e is *cleared* (authorized to know) information I if and only if $dom(I) \leq dom(e)$. The secrecy level $l = dom(e)$ is the *classification* of e. In this case, we also say e is l-classified. The pair $(L, \leq)$ forms a lattice. We will later use the lattice property that any finite subset F of secrecy levels has a unique least upper bound $\sqcup(F) \in L$.

Matos et al. [5] propose an elegant way to deal with dynamic aspects of information flow policies such as policy changes in systems with decentralized information-flow controls [6], [7], role changes [8], or the declassification of sanitized information: by temporarily changing the dominates relation $\leq$ in the lattice $(L, \leq)$.

Our approach is based on the observation that increasing actual execution times preserves the real-time guarantees of the system provided the corresponding worst-case execution time is not exceeded. Increasing the resource access times that a task perceives to the worst-case resource access times is therefore safe from a real-time perspective. Viewed from a security perspective, this increase allows us to equalizes the observable timing of resource accesses and hence to avoid exploitation of timing channels based on varying resource holding times. J. Agat [9] introduced timing leak transformations to avoid leakage of confidential data in the execution of unbalanced conditionally executed code paths. Engblom et al. [10] were first to realize that delaying code paths to their worst-case execution times avoids the leakage of timing-encoded information.

In addition to equalizing the time spend in resource requests, we must also take care of correctly accounting this time. We generally assume that the real-time operating system protects real-time tasks from malicious or erroneous tasks. In particular, we assume some form of address space or memory protection when running differently classified tasks in the same system. We assume further that the scheduler enforces budgets. That is, it forcibly preempts tasks that risk overrunning their deadline or that are up to exceeding their worst-case execution times. Wolter et al. [3] introduce *downward donation*[1] in the context of microkernel-based systems as one of two mechanisms to account on-behalf execution by servers and other resources to the resource-requesting clients. Similar to priority inheritance [11], recipients of downward-donated time inherit the priority of all donating tasks. However, unlike the priority-inheritance mechanism, which leaves it open to which task inherited time is accounted, downward donation attributes the donatee's execution time to the donator.

To simplify the following discussion, we will stick with the microkernel formulation and talk about donation between the tasks of the set of tasks to schedule. We will extend this task set with special protocol tasks implementing various aspects of the protocol. However, keep in mind that we introduce our transformation primarily for the purpose of proving resource access protocols secure wrt. covert channels of the second and third kind. Any implementation mimicking the behavior of our transformed protocols inherits these security properties.

If a task τ_s issues a protocol-related request to task τ_r, τ_s donates its time and priority to τ_r in a downward-donating fashion. That is, whenever the scheduler picks τ_s it will run τ_r instead of τ_s attributing any execution of τ_r to τ_s' budget. Downward donation is transitive in the sense that if a task τ_s donates to task τ_r, which itself has issued a request to another task τ_t, then τ_t will receive the time and priority of τ_s.

Of course there are several covert channels due to the variability in execution due to the scheduler (see e.g., Foss et al. [12] and Völp et al. [13]). In this work we shall ignore these channels in the sense that we prove a conditional statement: if the underlying scheduler is non-interference secure, then downward donation preserves this property and downward-donating protocols prevent the exploitation of shared resource timing channels.

Non-interference [14] is the prevailing formalization for the complete absence of security policy violating information flows. Intuitively, a system's scheduler is non-interference secure if for every secrecy level $l \in L$ variations in the behavior of higher-than-l classified tasks have no impact on the information that an l-classified task may learn. We say a locking protocol is *confidentiality preserving* if the underlying scheduler is non-interference secure and the protocol can be shown to preserve this property.

Following Audsley's classification of real-time locking protocols [15], we shall focus in this paper on predictable, preemptive and blocking protocols for controlling mutually exclusive access to shared resources. We discard non-predictable protocols from our analysis as they are not suited for real-time systems. Non-preemptive protocols, such as the flexible

[1] Wolter et al. chose the term downward donation in contrast to upward donation to indicate whether the donatee receives both the time and priority of the donator or just the time. The typical applications of these mechanisms is to help out lower prioritized tasks by donating the high priority of a blocked task down to these tasks. In contrast, upward donation is typically used to boost up priority for the time of the resource access.

multiprocessor locking protocol (FMLP) [16], require other mechanisms to prevent leakage due to non-preemptive execution [13]. As usual, we assume all resource accesses are coordinated by a single instance of the locking protocol. An extension to disjoint resource groups is straight forward.

III. Downward Donation Preserves Non-interference

We now prove that downward donation preserves the non-interference properties of schedulers if restricted to equally classified tasks or to higher classified time-consuming tasks. In Section IV A–F, we will use these results in our transformation of real-time locking protocols. If a task τ_i accesses a resource, we first donate to the higher classified time-consuming protocol task τ_i^M, which hides from τ_i when exactly the resource is assigned to τ_i. τ_i^M in turn may donate to equally classified protocol tasks τ_j^R to mimic the priority inheritance rules of the locking protocol and to mediate the order in which resources are assigned to requesting tasks.

We first introduce our formal model of schedulers, state what it means for a scheduler to be non-interference secure and then prove the two main theorems of this paper.

A. A Formal Model of Schedulers

To prove non-interference of downward donation in a scheduler-independent way, a formal model of the scheduler and of the scheduled task set is required. This model must expose all possible information flows and abstract from unnecessary details to remain manageable.

Informally, a scheduler implements a mapping for each point in time from runnable tasks to processors. Whether a task is released and runnable, out of budget, awaiting the release of its next job or blocked on asynchronous IO or on other tasks holding a resource depends on the behavior of this task. In particular, this behavior may change over time as a reaction to previous scheduling decisions. For example, adaptive real-time tasks (such as video decoders) may switch to low quality modes if they did not receive sufficient time to produce a high quality result. We do not restrict the type of task (strictly periodic, sporadic or aperiodic) in our model. Notice however that real-time locking protocols may re-introduce such a restriction as we require bounded resource access times.

More formally, we characterize the scheduling-related behavior of each task τ_i in the task set T as a function ϕ_i, which maps possibly observed historic scheduling decisions (i.e., the parts of the schedules for time steps $0 \dots t-1$ that τ_i may see) to the prospective action of τ_i at time t. Actions can be $block$ to indicate the task's inability to execute at time t or run to indicate that the task τ_i is runnable at time t and may thus be assigned to a processor. Later, we shall introduce $donate(\tau)$ as further actions with $\tau \in T \uplus \{-\}$. If τ_i performs the action $donate(\tau)$, it downward-donates its time and priority to the donatee τ. Here and in the following $\uplus$ stands for the disjoint union of two sets. For the benefit of a simpler notation, we will later replace run with $donate(-)$ and write $\phi(\tau_i) := \phi_i$ for the characteristic function ϕ of the task set T.

As indicated, the type of the schedule S_{t-1} up to the time $t-1$ is a mapping of points in times and processors ($p \in [0, m-1]$) to tasks: $[0, t-1] \to [0, m-1] \to T \uplus \{-\}$. We use the special symbol $-$ to indicate that no task was selected. One parameter of the characteristic function ϕ_i of a task τ_i is the sanitized version of the schedule, that is, the allocation of those tasks to processors from which τ_i is authorized to receive information. The function $purge(t-1, S_{t-1}, dom(\tau_i))$ produces such a schedule from the schedule S_{t-1} by replacing all occurrences of tasks τ' with $-$ from which τ_i must not receive. For these tasks, $dom(\tau') \not\leq dom(\tau_i)$ holds.

Definition 1 (Purge): Let S_{t-1} be a schedule up to time $t-1$ and $l \in L$ a secrecy level. The function

$$purge(t-1, S_{t-1}, l) := \lambda\ t \in [0, t-1].\lambda\ m \in [0, m-1]. \begin{cases} S_{t-1}(t, m) & \textbf{if } dom(S_{t-1}(t, m)) \leq l \\ - & \textbf{otherwise} \end{cases} \tag{1}$$

extracts from S_{t-1} the schedule that tasks classified at l or a higher secrecy level are authorized to see.

Figure 1 illustrates the purging of the schedule for the task τ_1. It shows an example schedule of four tasks $\tau_1 \dots \tau_4$ of increasing priority. τ_3 and τ_4 are authorized to send to both τ_1 and τ_2 but τ_2 is not authorized to send information to τ_1 (i.e., $dom(\tau_i) \leq dom(\tau_j), i \in \{3, 4\}, j \in \{1, 2\}, dom(\tau_2) \not\leq dom(\tau_1)$). Because $dom(\tau_2) \not\leq dom(\tau_1)$, any occurrence of τ_2 is removed from the schedule. In general, this does not mean that τ_2 cannot influence τ_1's behavior. For example, if the scheduler would run τ_1 at t_1 as a result of τ_2 blocking. Non-interference secure schedulers avoid these influences.

The type of the characteristic function ϕ_i is $[t \in \mathbb{N} \to ([0, t-1] \to [0, m-1] \to T \uplus \{-\}) \to \{run, block\}]$. The first parameter of type $\mathbb{N}$ is the time t for which ϕ_i characterizes τ_i's behavior. For example, $\phi_i(t, \dots) = run$ indicates that τ_i will be runnable at time t. In our theorems, ϕ_i will appear as a universally quantified parameter, generalizing our results to arbitrary task behaviors.

The final ingredient of our formal model is a scheduler S. Given a time t, the characteristic task functions ϕ_i of all tasks $\tau_i \in T$ and the historic schedule S_{t-1} (i.e., the schedule up to time $t-1$), S produces the new schedule S_t up to time t by evaluating $\phi_i(t, purge(t-1, S_{t-1}, dom(\tau_i)))$ and by assigning up to m runnable tasks to the m processors of the system.

We evaluate ϕ_i and S in an interlaced fashion. ϕ_i takes as a parameter the schedule up to time $t-1$ to produce the prospective action of τ_i for time t. S in turn takes this result to produce the schedule for time t and so on. Figure 1 illustrates this interlaced evaluation for a secure fixed-priority scheduler taken from [13].

Although the characteristic function ϕ_1 of τ_1 indicates that τ_1 is runnable at time t_1, the scheduler refrains from assigning this task to a processor to avoid information leakage from τ_2 to τ_1. If the scheduler would have selected τ_1, τ_2 could have relayed information to τ_1 by altering between running and blocking at t_1 to send a 1 or a 0. One possibility for τ_1 to receive this information is to read the system clock or some other source of timing information and to check whether it was selected at time t_1. In this way, τ_1 could receive information from τ_2 or from all other tasks from which τ_2 may legitimately receive. To not risk overlooking such illegal information flows, our model makes the following two pessimistic assumptions

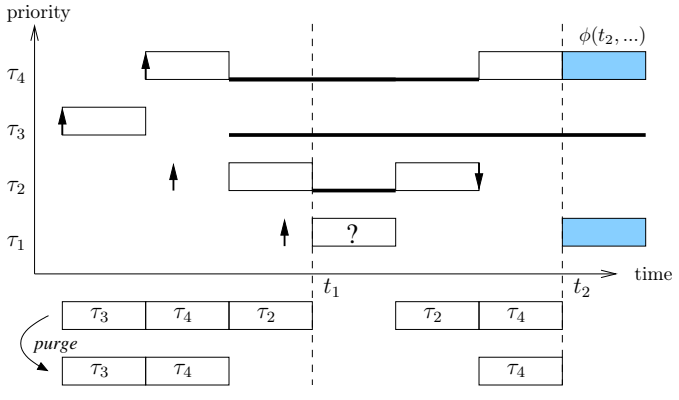

Fig. 1. Interlaced evaluation of the characteristic task function ϕ and the scheduler S. Execution and blocking of tasks $\tau_1 \dots \tau_4$ are denoted by bars or bottom lines, respectively; releases and deadlines by up- and downward arrows. The behavior of τ_1 at time t_2 (i.e., the result of $\phi_1(t_2, \dots)$) may depend on the previous schedule, but tasks from which τ_1 must not receive are removed (purged) from this schedule.

by passing ϕ_i the complete historic scheduled (purged to τ_i's secrecy level):

1) tasks have access to sufficiently precise clocks
2) all legitimate communications happen and are instantaneous.

Overestimation the adversaries' possibilities in this way allows us to detect all information flows (even if only small amounts of secret information are transmitted). For an analysis of systems that tolerate a limited bandwidth, the above two assumptions may of course be relaxed.

Notice that our model abstracts from the concrete data written into and read from shared resources. Instead, we have only formalized the timing of resource accesses by considering when tasks donate to other tasks to help free the resources they hold. For this reason, we can only prove the absence of information leaks that exploit the duration or timing of resource accesses but not the leaks by storing information in the resources. As already mentioned in the introduction, these leaks have to be eliminated by trusted instances anyway. Our approach helps eliminating potential indirect leaks that reveal when these sanitizing instances access the resource on behalf of their clients.

B. Non-interference

We now define what it means for a scheduler to be secure as far as information leakage is concerned. Intuitively, a system is non-interference secure wrt. to entities (observers) classified at a secrecy level $l \in L$ if whenever the system is presented inputs that are indistinguishable from the perspective of such an observer, it will produce the same outputs on all channels that this observer is authorized to see. In other words, variations in the higher-than-l classified inputs must not affect lower-than-l classified outputs. Inputs in our setting are the behaviors of those tasks (formalized by their characteristic functions) from which tasks τ with $dom(\tau) \leq l$ may legitimately receive (i.e., tasks τ' with $dom(\tau') \leq dom(\tau)$ for all τ such that $dom(\tau) \leq l$). Outputs are the points in time (and processor allocations) of these observable tasks. The function $purge(t, S_t, l)$ extracts these outputs from the schedule S_t. As described before, our formalization captures only the timing of operations not the data on which these operations execute. We now define when two inputs to our scheduler are indistinguishable from the perspective of an l-classified observer.

Definition 2 (l-Similar Task Behavior): Let ϕ and ψ be two characteristic functions of the task set T. We say ϕ and ψ are l-similar *if for all tasks $\tau \in T$, it holds that $dom(\tau) \leq l \Rightarrow \phi(\tau) = \psi(\tau)$.*

In other words, tasks that can legitimately be seen by an l-classified observer exhibit the same behavior. Tasks that must remain concealed from such an l-classified observer may differ in their execution or blocking behavior.

Definition 3 (Non-Interference): A scheduler S is non-interference secure with regards to an l-classified observer *if for all l-similar pairs of characteristic functions ϕ and ψ and for all points in time t it holds that $purge(t, S_t^{\phi}, l) = purge(t, S_t^{\psi}, l)$. It is* non-interference secure *if the above property holds for all observers.*

From the condition $purge(t, S_t^{\phi}, l) = purge(t, S_t^{\psi}, l)$ we can immediately conclude that no matter what higher-than-l classified tasks do, a non-interference scheduler will execute observable tasks on the same processors and at the same points in time. In other words, the behavior of higher classified tasks has no effect on the schedule of lower-than-l classified tasks.

C. Downward Donation

Downward donation is a mechanism by which tasks can forward their time and priority to other tasks. Whenever a task τ_s issues a request to a second task τ_r and awaits the reply to this request, τ_s may execute the action $donate(\tau_r)$ to indicate to the scheduler that τ_r should run on the time and priority of τ_s. As such, one can think of downward donation as an add on to a donation-agnostic scheduler. Whenever this scheduler picks τ_s at time t, it attributes any execution to the budgets of τ_s, considers its deadline and priority for this selection, etc. However, instead of running τ_s it switches to τ_r if τ_r is runnable at time t ($\phi(\tau_r)(t, \dots) = donate(-)$). If not it assumes that τ_r and hence also τ_s is blocked and proceeds selecting another task (e.g., the next lower prioritized runnable task). In this way, τ_s donates its execution time and priority to τ_r because τ_r replaces τ_s in the schedule for the time of the donation.

Although we shall investigate in this paper only locking protocols, which use downward donation locally to help out tasks on the same processor, our theorems apply also to multiprocessor systems where donatees migrate to the processor of the donator.

Of course, donation can be transitive if τ_r's action at time t is $donate(\tau_r')$ for some task τ_r'. We assume that downward donation is only used in such ways that donation chains are acyclic and write $\tau_r = donatee^*(t, S_t, \tau_s)$ to denote the task τ_r at the end of the donation chain of τ_s (i.e., $\phi(\tau_s)(t, \dots) = donate(\tau') \wedge \phi(\tau')(t, \dots) = donate(\tau'') \wedge \dots \wedge \phi(\tau^n)(t, \dots) = donate(\tau_r) \wedge (\phi(\tau_r)(t, \dots) = donate(-) \vee \phi(\tau_r)(t, \dots) = block)$). We say a task is

time-consuming if it never blocks on donated time.

Theorem 1: Downward donation to equally classified tasks preserves confidentiality
Let S be a non-interference secure scheduler (see Def. 3) for task sets that do not donate in a downward-donating fashion (i.e., task actions are limited to $block$ and $run = donate(-)$). Then, S is also non-interference secure for tasks τ_s that donate to equally-classified tasks τ_r. For these tasks $\phi(\tau_s)(t, \ldots) = donate(\tau_r) \Rightarrow dom(\tau_s) = dom(\tau_r)$ holds.

Proof: By transformation of the characteristic behavior ϕ of the task set T into a new characteristic behavior ϕ' where donations are replaced by $block$ or run actions and by realizing: (1) the scheduler is already secure for the transformed behavior with no donation actions included and (2) tasks, which are not authorized to learn about those tasks involved in the donation, will observe the same purged schedule as if the donating tasks (and all their legitimate receivers) had shown the transformed behavior in the first place. This second point is supported by the fact that $dom(\tau_r) \leq dom(\tau_s)$ for equally-classified donating tasks. The proof proceeds by analysis of the different cases that this transformation has to consider (see Appendix A for the detailed case analysis). ■

Theorem 2: Downward donation to time-consuming higher classified tasks preserves confidentiality
Let S be a scheduler that is non-interference secure for task behaviors that contain only downward donations to equally classified tasks. Then, S is also non-interference secure if a task $\tau_s \in T$ donates in a downward-donating fashion to higher classified tasks τ_r, provided that τ_r runs only on donated time, receives donated time only from τ_s and consumes all time τ_s donates to it (either directly or indirectly by donating to other time consuming tasks).

Proof: The proof of Theorem 2 cannot rely on $dom(\tau_r) \leq dom(\tau_s)$ to hold. Otherwise, it is identical to the above proof of Theorem 1 with the exception that an l-classified observer may be authorized to learn about τ_s but not about the donatee τ_r. Because τ_r runs only on donated time it receives only from τ_s, it cannot happen that τ_s blocks on τ_r while τ_r is already running on other time (donated or own). The scheduler S will therefore pick τ_s and execute a task to which τ_s donates directly or indirectly. This task will consume all the time τ_s donates to it because it is time consuming. Therefore, the only information that τ_s may learn from τ_r is that all donated time gets consumed. Because we limited donation to equally or higher classified tasks, τ_s is not authorized to learn whether τ_r or some task τ'_r consumed this time. If τ_s would be cleared to receive from such a task τ'_r, then $dom(\tau'_r) \leq dom(\tau_s)$ and $dom(\tau_r) = dom(\tau_s)$ would hold because of $dom(\tau_r) \leq dom(\tau'_r)$ (donation only to higher or equally classified tasks). Recall, the relation $\leq$ in the lattice $(L, \leq)$, $\leq$ is a partial order and hence transitive and antisymmetric. ■

IV. Uniprocessor Donation Ceiling Protocol

Introducing the uniprocessor donation ceiling protocol (UPDC), we now put together the previously established results to obtain a non-interference secure real-time locking protocol. More precisely, we mimic the behavior of the basic priority ceiling protocol (BPCP) [11] with additional protocol tasks and downward donation as time and priority inheritance mechanism. Please recall that more traditional implementations of BPCP remain valid as long as they exhibit the same timing behavior as UPDC.

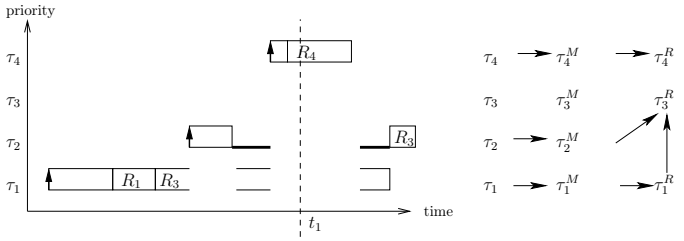

Fig. 2. UPDC's use of ceiling tasks and downward donation (arrows).

Figure 2 shows the additional protocol tasks and their use of downward donation. In addition to the tasks $\tau_i \in T$, we introduce two new types of tasks that run only on donated time and that implement the protocol steps of UPDC. Each resource accessing task τ_i is complemented with a mediator task τ_i^M whose responsibility it is to hide when precisely the resource is granted to τ_i. The second type of protocol tasks are ceiling tasks. Their role is to mediate when resources are allocated to τ_i, to execute τ_i's request on behalf of these tasks and to direct time to the appropriate resource holders to help free needed resources. All protocol tasks are time consuming. That is, we do not allow resource holders to suspend themselves or to engage in asynchronous IO. All protocol tasks are classified at a secrecy level l^P that is higher than the secrecy levels of all non-protocol tasks (i.e., $dom(\tau_i) \leq l^P$ for all τ_i).

To access a resource τ_i issues a request to τ_i^M with all the information that is necessary for τ_i^M (and other protocol tasks) to execute the resource access on τ_i's behalf. τ_i^M in turn forwards this request to its associated ceiling task, which decides whether to handle the request itself or to forward it to another ceiling task. Together with the requests, tasks donate their time and priority in a downward-donating fashion.

For UPDC, we introduce one ceiling task τ_c^R for each distinct ceiling priority of resources controlled by the protocol. The ceiling priority $\hat{R}$ of a resource R is the highest priority of all tasks that may access this resource. Because by definition of $\hat{R}$ a task cannot access resources with a ceiling priority lower than their own priority, we associate with τ_i^M the ceiling task τ_c^R that we have introduced for the lowest ceiling priority c that is larger than the priority of τ_i^M.

Whenever a ceiling task τ_c^R receives a request for a resource R (either from another protocol task (mediator or ceiling task) or when τ_i's request involves acquiring additional resources) it executes the request itself if $\hat{R}$ is smaller or equal to the ceiling priority c for which we introduced this task. Otherwise it forwards the request to the ceiling task introduced for the next highest resource ceiling priority.

Theorems 1 and 2 ensure that the resource requesting task τ_i cannot deduce which protocol task runs at which point in time. For this reason, τ_i cannot deduce directly whether at a specific point in time it is helping out a tasks, which holds a resource. However, by limiting the time it donates to τ_i^M to a value below the worst-case resource access time, it may

indirectly learn about this donation if τ_i^M would immediately return the data it has read from the resource. To avoid this illegal sampling of timing information, we let τ_i^M transform out this timing leak by returning from the resource request only after τ_i has donated an amount of time equal to the worst-case resource access time.

Because protocol tasks run only on donated time, they do not affect the real-time guarantees given by a non-interference secure scheduler for the original task set. For this reason and because we used the transformation primarily to prove non-interference and not to foster any particular implementation, we do not expect any performance impact with the exception of those introduced by the non-interference secure scheduler and by the timing leak transformation to the worst-case resource access time. Neither affect the real-time guarantees of BPCP.

Since all protocol tasks are strictly higher classified than non-protocol tasks, it is even possible to reuse the traditional analyses for the protocols discussed in this paper unless these analyses consider protocol-external blocking to improve schedulability. Under these constraints and given that none of the protocols we found to be secure helps out resource holders across processor boundaries, there is no reason for the scheduler to prohibit a task from running while it is requesting or accessing shared resources.

V. Confidentiality of Real-Time Locking

A. Priority Inheritance Protocol (PI)

PI is a uniprocessor resource locking protocol based on preemptive fixed-priority schedulers. Tasks are assumed to be time-wise independent (i.e., there are no precedence constraints or other temporal interrelationships). Whenever a task τ_i requests a resource that is held by a lower prioritized task τ_j, τ_j inherits the priority of τ_i unless the former releases the resource requested by the latter. PI does not per se prevent deadlocks. However, grouping resources [16] and similar precautions can prevent PI from entering a deadlocked state. Obviously, data confidentiality is at risk if a task may cause others to deadlock. Our non-interference results for PI are therefore limited to the non-deadlock case.

Our transformation of PI into a downward-donating protocol involves two protocol tasks τ_i^M and τ_i^R per non-protocol task τ_i (of course without loss of generality of behavior-mimicking implementations). As part of acquiring a resource R, τ_i donates its time via τ_i^M to τ_i^R. τ_i^R in turn accesses the resource or donates to τ_j^R for as long as τ_j^R holds resource R. All other settings are as described for UPDC with the exception that the PI worst-case resource access times have to be used by τ_i^M. We conclude from Theorems 1 and 2: *PI is confidentiality preserving*.

B. Basic Priority Ceiling Protocol (BPCP)

BPCP is described by the following three rules. The same assumptions apply as with PI.

1) *Scheduling Rule:* A task τ is scheduled preemptively in a priority-driven manner according to the current priority $\pi(\tau)$. When a job of τ is released, $\pi(\tau)$ is equal to the priority $prio(\tau)$ of the task τ.
2) *Allocation Rule:* τ blocks on held resources. Whenever τ requests a resource R that is free, one of the following two situations may occur:
   a) If $\pi(\tau)$ is greater than the ceiling priorities of all held resources, R is allocated to τ.
   b) If $\pi(\tau)$ is not greater than this ceiling priority, R is allocated to τ only if τ already holds a resource R' whose ceiling priority $\hat{R}'$ is equal to the maximum of the ceiling priorities of all held resources.
3) *Priority Inheritance Rule:* If τ blocks on a resource R held by τ', τ' inherits $\pi(\tau)$ until τ' releases all resources with a priority ceiling equal to or greater than $\pi(\tau)$. At this time, $\pi(\tau')$ drops to the value before it has acquired these resources.

To see that BPCP is confidentiality preserving, we establish the equivalence of UPDC and BPCP. We do so by comparing the rules that define the behavior of the latter:

1) *Scheduling Rule:* BPCP does not affect the priority of τ unless a higher prioritized task τ_h blocks on a resource R that τ holds. At this time $\pi(\tau)$ is raised to the priority of τ_h. UPDC parallels this behavior by executing τ's access to R in the ceiling task τ_c^R with $c = \hat{R}$. If τ_h blocks on R, it donates its time via τ_h^M and all ceiling task $\tau_{c'}^R$ with $prio(\tau_h) \leq c' \leq c$ to τ_c^R. Therefore, τ_c^R executes R on τ_h's priority.
2) *Allocation Rule:* UPDC characterizes the system ceiling priority $\hat{\Pi}$, which is the maximum of the ceiling priorities of held resources, only indirectly: $\hat{\Pi} = c$ holds if out of the set of ceiling tasks that are busy handling resource requests, τ_c^R is the ceiling task with the greatest ceiling priority. Ceiling tasks with higher ceiling priority are waiting for further requests.
   a) $\pi(\tau)$ is only greater than $\hat{\Pi}$ if the static priority of τ is greater. Therefore, the ceiling task $\tau_{c'}^R$, which UPDC associates with τ^M, and all ceiling tasks with a higher ceiling priority are free waiting for further requests and τ is granted R. Blocking resource holders violate this property. We therefore have to make the usual assumption that resource holders never suspend themselves or engage in asynchronous IO.
   b) Because τ_c^R executes τ_i's access to R on its behalf, UPDC grants resources that are acquired in a nested fashion only to the task that already holds a resource at the system ceiling priority. Notice, because a resource request R of a task τ_i passes and blocks the entire chain of ceiling tasks starting with the task associated with τ_i^M up to the ceiling task with ceiling priority $\hat{R}$, all resources that τ_i may acquire are either free or have a higher ceiling priority.
3) *Priority Inheritance Rule:* UPDC parallels BPCP's inheritance rule because tasks blocked on a resource R donate their priority (and time) to the ceiling task τ_c^R with $c = \hat{R}$. τ_c^R always runs on the time of the highest prioritized task that is blocked on R.

The above comparison shows that UPDC parallels all rules of BPCP. Hence, UPDC can be regarded as a *reformulation* of BPCP with protocol tasks and downward donation. It inherits all real-time properties of BPCP. This concludes our proof that *UPDC and BPCP are confidentiality preserving*.

C. Multiprocessor Priority Ceiling Protocol (MPCP)

MPCP as introduced in the classical work of Lehoczky et al. [17] builds upon a partitioned fixed-priority scheduler. As usual for multiprocessor locking protocols, MPCP distinguished between local and remote resources. Local resources are exclusively accessed by tasks pinned to the same processor. MPCP controls access to local resources by reverting to an instance of BPCP for every processor. Resources are said to be global if they are accessed by tasks that reside on different processors. MPCP executes global resource requests by blocking the requesting task on their home processors and by spawning new jobs on a dedicated synchronization processor where these new jobs execute the resource access. Resource accesses of the jobs on the synchronization processor are as well mediated by an instance of BPCP.

For local resources, MPCP inherits its non-interference properties from BPCP. Global resource accesses cannot influence the local execution on a processor. However, a blocking-aware schedulability analysis must consider the times when a non-interference secure scheduler prohibits local tasks from running while a higher prioritized task awaits the completion of a remote resource request [13]. MPCP, in contrast to the generalized multiprocessor priority ceiling protocol, executes only the newly spawned jobs on the synchronization processor. It is therefore impossible for these jobs to leak to tasks not currently involved in resource requests. Leakage between these jobs is prevented by means of the timing leak transformation of the mediator tasks τ_i^M on the synchronization processor, which execute these jobs on behalf of the non-protocol tasks. For these reasons, *MPCP is confidentiality preserving*.

D. Generalized MP Priority Ceiling Protocol (GMPCP)

GMPCP [17] generalizes MPCP in two aspects: it allows for more than one synchronization processor and it allows tasks to execute on the synchronization processors in addition to the jobs spawned for executing global resource requests. To make the timing of these spawned jobs independent of the execution of tasks that do not access resources, GMPCP raises the priority of spawned jobs to a priority band that is strictly higher than the priority of all other tasks. Increasing the number of synchronization processors constitutes no problem because all these processors execute spawned jobs as described above. However, unless all tasks on the synchronization processor are authorized to receive information from all resource requesting tasks, GMPCP is not confidentiality preserving because the newly spawned jobs influence the schedule of the tasks allocated to the synchronization processor. Spawned jobs arrive spontaneously when a task on another processor issues a global resource request, the traditional means of non-interference secure schedulers fail short in confining these tasks to reasonable execution slots.

E. Uni- and Multiprocessor Stack Resource Protocol

In contrast to the priority ceiling protocols, which raise the priority of a task only as a result of a resource requests by higher prioritized tasks, stack resource protocols immediately adjust the priority of resource holding tasks. As a result, tasks not currently involved in resource requests are prevented from running if their priority is in between the resource holding original priority and the priority to which this task is boosted. For this reason, we have to conclude that the stack resource protocols are not confidentiality preserving.

F. Clustered O(m) Locking Protocol (cOMLP)

Motivated by the goal to minimize the worst-case resource access time, which can be significant in the above protocols, in particular for global resources accesses, Brandenburg et al. [18] introduce a family of O(m) locking protocols for systems with c clusters with m_i processors in the i^{th} cluster. The protocols are based on a new inheritance mechanism: *priority donation*. The basic idea of priority donation is to let all jobs of all tasks help out resource holders for a bounded amount of time immediately after they are released. This helping occurs irrespective of whether or not the job will acquire resources. This mechanism and the additional constraint that a job is only allowed to access a resource if it is among the m_i highest prioritized jobs in its cluster ensures that at most $m = \sum_{0 \leq i < c} m_i$ jobs are simultaneously waiting for a resource. Therefore, by ordering these requests in first-come first-served order, a worst-case blocking time of O(m) can be guaranteed.

Priority donation works as follows. If a task τ_i preempts a resource holder τ_j when it is released, the resource holder inherits τ_i's priority for at most one critical section length. By transforming out potential timing leaks due to shorter priority donation times, *cOMLP becomes confidentiality preserving* by means of a similar construction as we have introduced for PI: τ_j donates its time and priority via the higher classified task τ_j^M to τ_j^R, which accesses the resource on τ_j's behalf. If τ_i preempts a job τ_j, which holds a resource, it first donates via τ_i^M to τ_i^R for exactly the length of one critical section (hiding shorter resource access times in τ_i^M). τ_i^R in turn donates to τ_j^R. Like before, we have to assume that resource holders do not suspend themselves or engage in asynchronous IO.

VI. Related Work

To the best knowledge of the authors, this is the first work on confidentiality-preserving real-time resource locking protocols. Nevertheless, there is a significant body of work that is related to the results we present here. Perhaps most closely related are the works by Son et al. [19], [20] and Xiao et al. [21] on real-time database concurrency control. The primary concern of these papers is not to guarantee worst-case bounds on resource access times but to minimize the likelihood of deadline misses of transactions in multi-level secure database systems. Hence, they fall into the class of non-predictable concurrency control mechanisms, which Audsley found to be not applicable for the hard real-time systems we address [15]. One interesting aspect of [19] is the trade-off between covert channel capacity and locking performance. This trade-off improves performance in systems where small amounts of leaked information are tolerable. In this paper, we

offer a more general solution, which applies also to systems that have to protect very small amounts of secret information.

In Section V, we refer to existing locking protocols, which address the real-time aspects of resource sharing. Audsley's review of resource control techniques [15] and the more recent works by Brandenburg et al. [16], [18] provide an excellent overview of activities in this field. Similarly, there is a significant body of work which focuses on the security aspects of non-real-time locking protocols. For instance, Reed et al. [22] proposed a non-blocking scheme for reader/writer synchronization assuming that all writers of a resource are classified to the same secrecy level. In our scheme, security with regard to resource timing channels is also guaranteed for differently classified writers.

The results presented in this work are orthogonal to the general question of how to prevent information leakage in real-time schedulers [12], [13].

VII. Conclusions

This paper presents the results of a novel information-flow analysis of seven real-time locking protocols. We first showed that downward donation preserves confidentiality in certain situations and then used this fact to transform existing locking protocols into downward-donating protocols, thereby showing that four protocols preserve the non-interference properties of their base schedulers.

Directions for future work include an investigation of further protocols, other time and priority donation mechanisms and other situations in which donation preserves the confidentiality guarantees established by the base schedulers. Work on non-interference secure dynamic priority schedulers and on locking protocols that are based on these schedulers would be highly interesting.

Appendix A
Proof of Theorem 1

To show that downward donation to equally classified tasks preserves confidentiality we chose a scheduler S, which is non-interference secure for non-donating tasks, and show by induction over the time parameter t that $purge(t, S_t^\phi, l) = purge(t, S_t^\psi, l)$ holds for all secrecy levels $l \in L$ and for all l-similar characteristic functions ϕ and ψ.

For each induction step, we apply the below transformation to obtain two new characteristic functions ϕ' and ψ', which exhibit the same scheduling behavior as seen by l-classified observers and which contain no donations in the first t time steps. The base case is trivial because we can assume that all processors idle in their initial states $S_0^\phi = S_0^\psi$.

Without loss of generality, let us assume that at time t no task τ in ϕ or ψ with $dom(\tau) \leq l$ donates its time and priority to another task and that no donations are present other than to equally classified tasks. The below transformation is a way to construct such pairs if observable tasks donate to equally classified tasks. Notice that under these constraints tasks τ that the l-classified observer may legitimately see can only donate to other tasks τ' that l can also see. This is because $dom(\tau) = dom(\tau')$ and hence $dom(\tau') \leq l$.

From the induction hypothesis we know that the scheduler is non-interference secure despite donations up to time $t-1$ because $purge(t-1, S_{t-1}^\phi, l) = purge(t-1, S_{t-1}^\psi, l)$ holds for both the original pair of characteristic functions (ϕ, ψ) and for the transformed pair (ϕ', ψ') where donations up to time $t-1$ have been removed.

If at time t there remains a task τ_s in ϕ or in ψ that donates time to another task τ_r then we eliminate this donation with the following transformation for τ_s and subsequently also for all other donating tasks to obtain the new pair ϕ' and ψ' where no tasks donate at time t. Because, as far as the donation-agnostic scheduler S is concerned, ϕ' and ψ' and ϕ and ψ exhibit the same behavior and because S is non-interference secure for non-donating task sets (including those characterized by ϕ' and ψ'), S is non-interference secure despite donations.

A. Transformation

We now describe our transformation which eliminates a donation from τ_s to τ_r without changing the behavior of tasks as far as S is concerned. To do so, we have to distinguish the following three cases:

1) τ_r *is blocked*: In this case, τ_s must block as well because if S considers $\phi_s(t, \ldots) = donate(\tau_r)$ at time t it will search for another task to run or leave the processor idle. $\phi'_s(t, \ldots) = block$ accommodates for this behavior. Because τ_r remains blocked, other tasks τ_o, which are authorized to observer τ_s and τ_r will not adjust their behavior because neither τ_s nor τ_r appear as executing task in the schedule passed to ϕ_o.
2) τ_r *consumes the received time*: In this case, we have to distinguish further whether (2a) the scheduler has already assigned τ_r to a processor or not (2b).
   - (2a) τ_s blocks on τ_r because τ_r already has received time from some other source. τ_r runs on its assigned processor and τ_s blocks. $\phi'_s(t, \ldots) = block$ accommodates for this change.
   - (2b) If τ_r has not yet been assigned to a processor S will run τ_r instead of τ_s and on τ_s' processor. To the scheduler it appears as if τ_s would run itself because it will account τ_r's execution to the execution budgets of τ_s. For all subsequent considerations, τ_r appears to be blocked. $\phi'_s(t, \ldots) = donate(-)$ and $\phi'_r(t, \ldots) = block$ accommodates for this point.
3) τ_r *donates to some other task* τ'_r: In this case, τ_r will not consume the time τ_s donates to it. Instead, it forwards this time to τ'_r. We can therefore change $\phi_r(t, \ldots) = block$ and repeat the above transformation with τ'_r instead of τ_r.

For all other tasks τ_x, $\phi_x(t, \ldots)$ remains unchanged (i.e., $\phi'_x(t, \ldots) = \phi_x(t, \ldots)$).

In the two situations discussed in case 2, the scheduler will assign τ_r to different processors depending on whether τ_r was considered prior to τ_s. Clearly, other tasks may react to this behavior provided they can learn about this change. In the case

that τ_s and τ_r are visible to tasks that the l-classified observer is authorized to see, l-similarity demands that the reaction is the same in ϕ and ψ. Because we quantify over all l-similar behaviors we consider all possible reactions. If tasks see τ_r and τ_s that are higher classified than l, their behavior may differ in the first place and again we quantify over all possible behaviors. What remains to be shown is that it is impossible for a task τ_i with $dom(\tau_i) \leq l$ to learn about the change in case 2 if $dom(\tau_s) \not\leq l$.

Because ϕ_i is invoked with $purge(t-1, S_{t-1}, dom(\tau_i))$, τ_i may adjust its behavior to a donation from τ_s to τ_r only if $dom(\tau_s) \leq dom(\tau_i)$, if $dom(\tau_r) \leq dom(\tau_i)$ or if in case 2a the scheduler selects another task τ_j from which τ_i is able to receive (i.e., for which $dom(\tau_j) \leq dom(\tau_i)$ holds). But then we can conclude from the transitivity of $\leq$ that $dom(\tau_s) \leq l$, $dom(\tau_r) \leq l$ respectively $dom(\tau_j) \leq l$ holds. Because $dom(\tau_s) \leq dom(\tau_r)$, the transitivity of $\leq$ allows us to collapse the first two situations, which contradicts our assumption that τ_s is not already visible to the l-classified observer. In the last case, S would not be able to hide variations in the execution or blocking behavior of τ_s unless $dom(\tau_s) \leq dom(\tau_j)$. That is, S would not be non-interference secure. But then, by transitivity of $\leq$, $dom(\tau_s) \leq l$, which again contradict our assumption that $dom(\tau_s) \not\leq l$.

ACKNOWLEDGMENTS

This work was in part supported by the European Union through PASR project "Open Robust Infrastructures", by the German Research Foundation (DFG) through the CRC 912 "Highly Adaptive Energy Efficient Systems" and the QuaOS project and by the State Saxony and the European Union through the ESF young researcher group IMData.

REFERENCES

[1] D. Demange and D. Sands, "All secrets great and small," in *ESOP '09: Proceedings of the 18th European Symposium on Programming Languages and Systems*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 207–221.

[2] ECRYPT II, "Yearly report on algorithms and keysizes (2010-2011)," Technical Report D.SPA.7 Rev 1.0, ICT-2007-216676, June 2011.

[3] U. Steinberg, J. Wolter, and H. Härtig, "Fast Component Interaction for Real-Time Systems," in *17th Euromicro Conference on Real-Time Systems*, Palma de Mallorca, Spain, July 2005.

[4] D. Denning, "A lattice model of secure information flow," in *Communications of the ACM*, vol. 19-5. New York, NY, USA: ACM Press, 1976, pp. 236–243.

[5] A. A. Matos and G. Boudol, "On declassification and the non-disclosure policy," in *CSFW '05: Proceedings of the 18th IEEE workshop on Computer Security Foundations*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 226–240.

[6] S. Vandebogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières, "Labels and event processes in the asbestos operating system," *ACM Trans. Comput. Syst.*, vol. 25, no. 4, p. 11, 2007.

[7] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in histar," in *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2006, pp. 19–19.

[8] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models," *Computer*, vol. 29, no. 2, pp. 38–47, Feb. 1996.

[9] J. Agat, "Transforming out Timing Leaks," in *ACM Principles of Programming Languages*, Boston, Mass., Jan 2000.

[10] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, and H. Hansson, "Worst-case execution-time analysis for embedded real-time systems," *International Journal on Software Tools for Technology Transfer (STTT)*, pp. 437 – 455, 2003.

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

[12] 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.

[13] M. Völp, C. J. Hamann, and H. Härtig, "Avoiding timing channels in fixed priority schedulers," in *Asian Conference on Computer and Communication Security (ASIACCS '08)*. Tokyo, Japan: ACM, March 2008.

[14] J. A. Goguen and J. Meseguer, "Security Policies and Security Models," in *IEEE Symposium on Security and Privacy*, Oakland, California, USA, 1982, pp. 11–20.

[15] N. C. Audsley, "Resource control for hard real-time systems: A review," University of York, Tech. Rep., 1991.

[16] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson, "A flexible real-time locking protocol for multiprocessors," in *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2007.

[17] R. Rajkumar, L. Sha, and J. P. Lehoczky, "Real-Time Synchronization Protocols for Multiprocessors," in *Real-Time Systems Symposium*. IEEE, 1988, pp. 259–269.

[18] B. B. Brandenburg and J. H. Anderson, "Real-time resource-sharing under clustered scheduling: mutex, reader-writer, and k-exclusion locks," in *EMSOFT*, 2011.

[19] S. H. Son, R. Mukkamala, and R. David, "Integrating security and real-time requirements using covert channel capacity," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, November 2000.

[20] C. Park, S. Park, and S. H. Son, "Multiversion locking protocol with freezing for secure real-time database systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, pp. 1141–1154, 2002.

[21] Y. Xiao, Y. Liu, and G. Liao, "A secure real-time concurrency control protocol for mobile distributed real-time databases," *IJCSNS International Journal of Computer Science and Network Security*, vol. 7, no. 1, January 2007.

[22] D. P. Reed and R. K. Kanodia, "Synchronization with eventcounts and sequencers," *Communinactions of the ACM*, vol. 22, no. 2, pp. 115–123, 1979.