Mixed-Trust Scheduling

Dionisio de Niz*, Bjorn Andersson*, Mark Klein*, John Lehoczky†, Amit Vasudevan*, Hyoseung Kim‡
*Software Engineering Institute
Carnegie Mellon University
{dionisio,baandersson,mk,avasudevan}@sei.cmu.edu
†Department of Statistics and Data Science
Carnegie Mellon University
jl16@andrew.cmu.edu
‡Electrical and Computer Engineering
University of California Riverside
hyoseung@ece.ucr.edu

***Abstract*—The verification of Cyber-Physical Systems is increasing in importance given the push to deploy complex autonomous features with life-threatening consequences. However, verifying CPS not only demands the verification of the application code but also of the supporting OS. Unfortunately, the complexity of traditional monolithic kernels has prevented their full verification. As a result, a new generation of verified hypervisors capable of hosting a full OS within a VM has been created. Notwithstanding, the verified properties of the hypervisor only apply to the code within it and do not extend to the code running the VM. In this paper we create a new scheduling model with tasks consisting of two parts, an unverified part that runs in the VM and a verified part that runs within the verified hypervisor. The unverified part can contain bugs and hence is considered untrusted. The verified part, on the other hand, is considered trusted. Due to this mixture we call this a mixed-trust task. A mixed-trust task is designed to run only its untrusted part if the execution does not run into any bugs. The untrusted part implements complex computations and has the support of a full-blown OS. However, if this part fails to finished (e.g., due to bugs) the trusted part is automatically activated by a timer within the trusted hypervisor to ensure some safety property (e.g. prevent a crash). The hypervisor ensures that the activation of the trusted part does not depend on the untrusted one. This execution model is implemented by the coordination of a verified non-preemptive scheduler in the hypervisor and a preemptive scheduler in the VM. Both the mixture of schedulers and the required independence of the hypervisor scheduler present key challenges that are addressed in this paper. In the paper we present the schedulability analysis for the mixed-trust scheduler, and its implementation based on the XMHF hypervisor and the ZSRM schedulers.**

I. Introduction

Certification authorities such as the FAA [18] allows the validation of different parts of a system with different degrees of rigor depending on their level of criticiality. This is only allowed if it is possible to prove that higher-criticality components are isolated from defects in lower-criticality components. Given that failure of the highest-criticality components can lead to fatal consequences, formal verification is highly recommended to provide provable guarantees. However, to preserve the level of rigor of this proof it is necessary to also formally verify the mechanism that isolates the highest-critical part from the rest.

The complexity of traditional monolithic kernels in general and of the virtual memory system in particular has prevented the verification of the code that implements process isolation. As a result, a new generation of micro-kernels and hypervisors has surged to cover this gap [20], [14], [11]. In this case, the verification is limited to the code that implements the isolation mechanisms and other simple micro-kernel and hypervisor services. Unfortunately, these services are limited and cannot support a full-scale application the way a full OS (such as Linux) can. Therefore, these works does not address the need to provide verified properties for complex real-life application.

This paper presents the timing verification work of a larger framework that allows the verification of large complex systems based on runtime verification [8], [1]. In this framework, small code components are added to the system to verify the input (e.g., sensing) and outputs (e.g., actuation) of the system ensuring that such outputs always lead to safe states (e.g., avoid crashes). It is worth noting that, in a CPS, verifying the behavior of a system depends not only on the values produced (outputs) but also the time when those outputs are produced. Hence, our framework includes a *temporal enforcer* that outputs a default safe action (e.g., *hover* in a quadrotor) if an output has not been produced on time to finish by the deadline. Clearly, if a provable guaranteed must be produced, the mechanism that triggers and host the temporal enforcer and the temporal enforcer itself must be verified. In this paper we present the real-time schedulability approach for tasks with verified temporal enforcers and the implementation of the scheduling scheme coordinating the scheduler within the verified hypervisor XMHF [20] and the ZSRM [9] scheduler in the VM.

In our real-time scheduling framework tasks have an unverified part followed by a verified part. The unverified part is expected to work most of the time (e.g., flying a drone through a mission) but may fail occasionally (say due to a bug) and, hence is considered untrusted. If this failure occurs, then the verified part takes over to preserve some safety invariants (e.g. prevents the drone from crashing). The latter part is considered trusted and is where the *temporal enforcer* from our runtime verification approach resides. As a result, these tasks are considered *mixed-trust* tasks. It is worth highlighting that the two parts are implemented as different subroutines (e.g., C functions) and, hence, the second part is not a continuation of the first one. This is a stark difference with mixed-criticality task models. However, both the trusted and untrusted parts are executed as a single job of a task and we need to ensure that they are executed periodically with a common period and must finish by their common deadline. This, together with the need to prevent trusted-part's dependencies from the untrusted part, presents important challenges that need to be addressed in the schedulability analysis.

Our runtime scheduling framework is composed of a verified hypervisor hosting a virtual machine running an unverified kernel. To provide the proper isolation between the trusted and untrusted environment we use two schedulers: (1) a simple verified non-preemptive fixed-priority scheduler in the hypervisor to host the verified part of the tasks and (2) a preemptive scheduler running in the kernel inside the virtual machine. This allows the runtime framework to provide a rich set of services (in a regular OS) to support complex applications maximizing mission objectives (e.g. wining an autonomous car race) while, at the same time, supports a verified isolation mechanism and basic services for the verified application code in charge of safety. The schedulability analysis of what we call a *mixed-trust scheduling* to analyzed the timing guarantees provided by the safe coordination of the two schedulers in the platform is the main topic of this paper. Our system model has two criticality levels where the untrusted part has a low-criticality and the trusted part high-criticality. The semantics of the criticality is a variant of the traditional mixed-criticality scheduling work with two important differences: (1) tasks have parts of different criticality levels instead of only one level. And (2) the isolation requirement for the high-critical part necessitates a separate scheduler and runtime environment. We present the analysis, enforcement mechanisms, and our implementation of the mixed-trust scheduler in a Raspberry Pi-3 board.

II. System Model

Our system is composed of a uni-core processor with a taskset $\Gamma = \{\mu_i | \mu_i = (T_i, D_i, \tau_i, \kappa_i)\}$ indexed in priority order, i.e., μ_i has higher priority than μ_j if $i < j$. In the task set μ_i is defined as a mixed-trust task with two execution segments τ_i and κ_i required to execute in that order and a period T_i and deadline D_i. The execution segment τ_i is considered to be untrusted and runs in the untrusted kernel inside the VM. On the other hand, the segment κ_i is considered trusted code (e.g. verified) and runs within the trusted hypervisor. For simplicity of presentation and to represent the fact that they are handled by different schedulers we consider these segments tasks and call the untrusted one a guest task (because it runs in the guest VM) and the trusted one a hyper task. These tasks are defined by:

$$\tau_i = (T_i, E_i, C_i) \tag{1}$$

and

$$\kappa_i = (T_i, D_i, \kappa C_i) \tag{2}$$

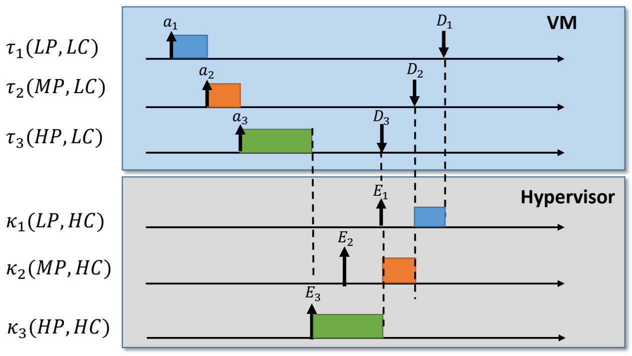


Figure 1: Mixed Trust Sample Timeline

Where T_i and D_i are the same as in μ_i (replicated here for convenience),C_i is the WCET of τ_i, and and κC_i is the WCET of κ_i. E_i determines how long the hyper task κ_i must wait after the arrival of a job of τ_i to be activated. It is worth noting that if a task tries to execute beyond C_i the task will be enforced not to exceed this budget. Tasks are assumed to have different priorities. For convenience we use $\kappa_{i,k}$ to identify the k's (hyper) job of the hypertask κ_i and $\tau_{i,k}$ to identify the k job of the guest task τ_i.

Under normal operation, the mixed-trust task μ_i only runs its guest task τ_i executing less than C_i and informing the scheduler of its completion. However, if it is detected that τ_i is taking too long to complete (e.g. due to an error or a security infiltration) then its execution is interrupted and κ_i is run within the hypervisor. To detect this, a timer is set to expire E_i time units after τ_i's arrival. The goal of the schedulability analysis is to compute E_i (if one exists) in order to ensure that all hyper-tasks can finish by their deadline D_i and all the guest tasks can finish before the timer E_i if they do not exceed their C_i. In our model, E_i is known as the enforcement timer. Figure 1 depicts a sample execution timeline for a mixed-trust taskset with three tasks.

III. Schedulability Analysis

The schedulability of a mixed-trust taskset is performed in three steps: we first calculate the worst-case response time (R_i^κ) of each hypertask κ_i running in non-preemptive fixed-priority scheduling. Then, we calculate the E_i timer for each guest task τ_i by simply subtracting R_i^κ from the deadline D_i. Finally, we calculate the response time of each guest task τ_i and verify that it is smaller than E_i.

1) Hyper-Task Response Time: To calculate the hyper-task response time we use previous results from the CANBus schedulability analysis [7]. In order to explain the mapping of this analysis to our model let us make the following observations about the non-preemptive scheduler used in our model analysis:

- **o1**. Even though a hyper task κ_i does not execute unless its corresponding guest task τ_i does not finish by the enforcement timer E_i, the worst case delay for κ_i occurs when all hyper-task always execute.

- **o2**. A high-priority task κ_i can be delayed by a lower-priority κ_k already running when κ_i arrived, due to its non-preemptive execution nature.
- **o3**. Once a task κ_i starts running, it experiences no further delays.
- **o4**. A job from a task κ_i can get extra carry-in preemptions at the beginning of its period from jobs of higher-priority tasks κ_j that were in turn delayed by lower-priority jobs from κ_k.

Now, the response time of a hyper-task κ_i is calculated in three steps:

1) A level-i non-preemptive busy period is calculated in order to explore all possible interleavings and find the worst-case response time of κ_i, in order to take into account potential carry-in preemptions as stated in **o4**.
2) The start time of each κ_i job in the busy period is calculated as the basis for the response time. The response time is then calculated by just adding the execution time to this starting time given observation **o3**.
3) The maximum response time among the jobs in the busy period is then calculated.

The level-i non-preemptive busy period is calculated with Equation 3.

$$t_i^\kappa = \max_{j \in \kappa L_i} \kappa C_j + \left\lceil \frac{t_i^\kappa}{T_i} \right\rceil \kappa C_i + \sum_{j \in \kappa H_i} \left\lceil \frac{t_i^\kappa}{T_j} \right\rceil \kappa C_j \tag{3}$$

where κL_i is the set of all hyper tasks with lower priority than κ_i and κH_i is the set of tasks with higher-priority than κ_i.

As pointed out in **o2**, Equation 3 takes into account the maximum preemption from one job of a lower-criticality task.

Then we can obtain the worst-case starting time of the q job with Equation 4.

$$W_{i,q}^\kappa = \max_{j \in \kappa L_i} \kappa C_j + (q-1)\kappa C_i + \sum_{j \in \kappa H_i} \left(\left\lfloor \frac{W_{i,q}^\kappa}{T_j} \right\rfloor + 1 \right) \kappa C_j \tag{4}$$

Finally the response time of job q is calculated by obtaining the longest starting delay of any job in the busy period and adding its execution time with Equation 5.

$$R_i^\kappa = \max_{x \in \left[1 \ldots \left\lceil \frac{t_i^\kappa}{T_i} \right\rceil\right]} (W_{i,x}^\kappa) + \kappa C_i - ((x-1)T_i) \tag{5}$$

Difference from CANBus. It is worth noting that the CANBus analysis only considers tasks with a single non-preemptive execution segment. In contrast, our tasks μ_i has two execution segments or subtasks: the guest task τ_i and the hyper task κ_i. The guest task runs in the VM under preemptive fixed-priority scheduling and the hyper task runs in the hypervisor under non-preemptive fixed-priority scheduling. More importantly, guest tasks only run if there is no hyper-task ready to run, i.e., the hyper-tasks are idle when the guest tasks run. This fact leads to a key difference that is reflected in the way the adapted CANBus equations to work in the mixed-trust model. Specifically, in a schedulable taskset, the level-i busy period of a hyper-task κ_i that has a corresponding guest task τ_i with non-zero WCET C_i (i.e., $C_i > 0$) ends before the second job of κ_i starts. This is because, a schedulable taskset will ensure that both the guest task τ_i and the hyper task κ_i have time to execute for their respective WCET C_i and κC_i. Hence, after the execution of the first job of κ_i there should be some time when τ_i executes for $C_i > 0$. Therefore, as pointed out before, during the time that τ_i executes there should not be any hyper-task ready to execute and this interval is consider idle time from the hypervisor point of view. This means that the hyper task level-i busy period ended as soon as this idle time started. Furthermore, this idle time not only ends the hyper task level-i busy period but the hyper task busy period at all levels.

Notwithstanding the single-job busy period observation in the previous paragraph, we keep the CANBus-based equations to allow tasksets without any guest task components.

We will now discuss how to choose E_i. In order for hyper tasks to be schedulable, clearly, we must choose E such that:

$$\forall \mu_i \in \Gamma, E_i \leqslant D_i - R_i^\kappa \tag{6}$$

We will later compute the guest response time of a task τ_i and denote it by R_i. In order for guest schedulability to hold, we must choose E such that:

$$\forall \mu_i \in \Gamma, R_i \leqslant E_i \tag{7}$$

Ideally, we would like to develop an algorithm that computes E for each task such that if there is an assignment for which the schedulability test is true, then our algorithm finds such an E-assignment. This is an interesting problem; though we will not address it here; we leave it for future research. Instead, in this paper, we use a rule-of-thumb. We assign E to be as large as possible while still satisfying Equation 6 and Equation 7. The rationale for this is that choosing E_i as large as possible gives as much space as possible for R_i; however, it may adversely affect R_l of a task τ_l with lower priority than τ_i. Based on this discussion, we choose to assign E with the following rule:

$$E_i = D_i - R_i^\kappa \tag{8}$$

2) Active-Period Exact Analysis of Guest Response Time: In order to calculate the exact response time of the guest we used a modified version of the active period from the CANBus analysis. However our analysis must account for the effects other tasks' guest and hyper tasks on τ_i and the effect of κ_i on τ_i. This in turn depends on a determining worst-case phasing for tasks relative to τ_i and κ_i.

We will argue in two steps, first considering the case in which κ_i does not exist. The argument is a slight variant of Theorem 1 in [15].

Lemma 1. *In the case when a guest task does not have an associated hyper task the longest response time for all jobs of* *guest task τ_i occurs in a level-i busy period initiated by the arrival of τ_i and the arrival of other tasks' guest or hyper tasks.*

Proof. Following the argument of Lehoczky [15], let [0, b] denote a level-i busy period. Assume that τ_i arrives at some point x_i after 0 during the busy period. Since the time before it starts,[0,x_i), is being used by higher priority guest tasks, higher priority hyper tasks, or lower priority hyper tasks, moving its start to zero cannot change its completion time and can only increase its response time.

Assume that τ_i arrives at 0, but some higher priority guest task, τ_j, is initiated after 0 while the prior hyper task does not become part of the busy period. Moving the initiation of τ_j to 0 will result in an increased (or unchanged) amount of work in every interval [0,t] for every t in [0,b) possibly increasing or leaving unchanged the response time of τ_i jobs. Similar arguments can be used if a higher or lower priority hyper task is initiated after 0. □

Now consider the case in which the guest task does have an associated hyper task. Aligning other task arrivals with the arrival of the k's job of τ_i guest task does not necessarily cause the worst-case response for τ_i's k's job. Sometimes aligning other arrivals with the $k-1$' job's hyper task of μ_i creates a busy period that includes the k's job of τ_i guest task and results in it having a worse response time.

Lemma 2. *In the case when a guest task does have an associated hyper task the longest response time for the job $\tau_{i,k}$ with the longest response time among all jobs of guest task τ_i occurs in a level-i busy period initiated by the arrival of either $\kappa_{i,k-1}$ or $\tau_{i,k}$ and the arrival of other tasks' guest or hyper tasks.*

Proof. The proof is very similar to the proof of above Lemma. Again let [0, b] denote a level-i busy period. Assume that the busy period includes $\kappa_{i,k-1}$ and $\tau_{i,k}$. Assume that $\kappa_{i,k-1}$ arrives at some point x_{i-1} after 0 during the busy period. Unlike the previous case moving the start of $\kappa_{i,k-1}$ to zero can reduce its response time since it is non-preemptible, but it can only increase the response times of all of the guest tasks in the busy period.

Assume that $\kappa_{i,k-1}$ arrives at 0, but some higher priority guest task, τ_j, is initiated after 0 while the prior hyper task does not become part of the busy period. Again moving the initiation of τ_j to 0 will result in an increased (or unchanged) amount of work in every interval [0,t] for every t in [0,b) and therefore increase or leave unchanged the response time of all guest task jobs in the busy period. (However, hyper tasks could benefit.)Similar arguments can be used if a higher or lower priority hyper task is initiated after 0. □

We first define a parameterized request-bound function in Equation 9. The notion of request-bound function has been used in previous work [3]. The request-bound function for a set of jobs from a given task τ_i, for a given time interval is the sum of the execution time of the jobs that have arrival times in this time interval. The request-bound function for a given task τ_i, for a given time interval is the maximum request-bound function that jobs of this task can generate in this time interval. The request-bound function for a given task τ_i, for a given time interval is the maximum request-bound function that jobs from this task can generate in this time interval. The request-bound function for a given task τ_i, for a given duration is the maximum request-bound function that this task can generate for a time interval of this duration. Recall that in our model, a task can generate a job but later the same job can "arrive" again (E time units later) to perform hypervisor execution. Therefore, from the perspective of request-bound function, this arrival of hypervisor execution is treated as the arrival of a job. The normal request-bound function takes only two parameters: a task and a duration. In our model, we will use a more specialized variant that takes two additional parameters, y (a phasing) and b (a 0-1 variable). We use the former parameter ($y \in \{E, A\}$) to indicate the phasing of the task τ_i; if $y = E$, then we are computing the request-bound function for the phasing when the level-i busy period starts at a time when a hypertask of τ_i arrives; analogously if $y = A$, then we are computing the request-bound function for the phasing when the level-i busy period starts at a time when a guesttask of τ_i arrives. We use the latter parameter ($b \in \{0, 1\}$) to indicate the whether the guest execution should be included in the counting of the request-bound function. If we would use $y = A$ and $b = 1$, then our notion measures the same quantity as the traditional request-bound function. Thus, our notion of request-bound function can be thought of as a generalization of the original notion of request-bound function.

The definition of request-bound function for our model is as given by the equation below:

$$\text{rbf}_i^y(t,b) = \begin{cases} \left\lceil \frac{t-(T_i-E_i)}{T_i} \right\rceil^+ C_i b + \left\lceil \frac{t}{T_i} \right\rceil \kappa C_i & \text{if } y = E \\ \left\lceil \frac{t}{T_i} \right\rceil C_i b + \left\lceil \frac{t-E_i}{T_i} \right\rceil^+ \kappa C_i & \text{if } y = A \end{cases} \tag{9}$$

We will use this notion of request-bound function to compute the response time of the guest execution of a given task τ_i. Then, if it holds for each task, that its computed guest response time is less than or equal to its E-parameter, then the taskset is schedulable (assuming that we have already checked hypertask schedulability). Therefore, our goal is now to present equations for computing the guest response time for a given task. We will do so by presenting an equation for the maximum duration of a level-i busy period. Then, compute the latest possible finishing time of a given job from a given task in this level-i busy period; then also show that arrival times of jobs can be moved to be as early as possible given the model; these two together (the finishing time and arrival time) allows us to compute the guest response time of a job. Since we know the maximum duration of a level-i busy period, we can compute an upper bound on the number of jobs of a given task in a level-i busy period; we can compute the maximum response time over all these jobs of the given task. This yields the guest response time. We will compute the guest response time for two cases: the case that the given task arrives in the beginning of the level-i busy period and the case that the given task arrives with a hypertask in the beginning of the level-i busy period. Given this high-level outline, we will now present the actual equations.

For each τ_i, for each $x \in \{E, A\}$, let $t_i^{g,x}$ denote the maximum level-i busy period such that this level-i busy period starts with a job of hypertask or guesttask of τ_i arriving (x indicates which). Then, in a similar spirit as Equation 3, we can, for $x \in \{E, A\}$, for a given task τ_i, compute $t_i^{g,x}$ as follows:

$$t_i^{g,x} = \left(\sum_{j \in L_i} \text{rbf}_j^E(t_i^{g,x}, 0) \right) + \text{rbf}_i^x(t_i^{g,x}, 1) + \sum_{j \in H_i} \max_{y \in \{E,A\}} \text{rbf}_i^y(t_i^{g,x}, 1). \tag{10}$$

Given a task τ_i and a level-i busy period, we refer to job q as the q^{th} job with a guest arrival in the level-i busy period. For each τ_i, for each $x \in \{E, A\}$, let $w_{i,q}^{g,x}$ denote the maximum finishing time of job q of task τ_i, relative to the start of the maximum level-i busy period, such that this level-i busy period starts with a job of hypertask or guest task of τ_i arriving (x indicates which). Then, in a similar spirit as Equation 4, we can, for $x \in \{E, A\}$, for a given task τ_i, for a given job index q of task τ_i, compute $w_{i,q}^{g,x}$ as follows:

$$w_{i,q}^{g,x} = \left(\sum_{j \in L_i} \text{rbf}_j^E(W_{i,q}^{g,x}, 0) \right) + qC_i + (q - 1 + I_{(x=E)})\kappa C_i + \sum_{j \in H_i} \max_{y \in \{E,A\}} \text{rbf}_j^y(w_{i,q}^{g,x}, 1). \tag{11}$$

In Equation 11, I_ϕ is an indicator function that returns 1 if the Boolean predicate ϕ is true and 0 otherwise.

For each τ_i, for each $x \in \{E, A\}$, let $R_{i,q}^{g,x}$ denote the maximum response time of job q of τ_i, relative to the start of the maximum level-i busy period, such that this level-i busy period starts with a job of hypertask or guest task of τ_i arriving (x indicates which). Then, in a similar spirit as part of Equation 5, we can, for $x \in \{E, A\}$, for a given task τ_i, for a given job index q of task τ_i, compute $R_{i,q}^{g,x}$ as follows:

$$R_{i,q}^{g,x} = w_{i,q}^{g,x} - ((q-1)T_i + I_{(x=E)}(T_i - E_i)) \tag{12}$$

For each τ_i, for each $x \in \{E, A\}$, let $R_{i,q}^{x}$ denote the maximum response time of τ_i, such that this level-i busy period starts with a job of hypertask or guest task of τ_i arriving (x indicates which). Then, in a similar spirit as part of Equation 5, we can, for $x \in \{E, A\}$, for a given task τ_i compute $R_i^{g,x}$ as follows:

$$R_i^{g,x} = \max_{q \in \left\{1 \ldots \left\lceil \frac{t_i^{g,x} - I_{x=E}(T_i - E_i)}{T_i} \right\rceil \right\}} R_{i,q}^{g,x} \tag{13}$$

Parameter	Default Value
Number of Tasks	10
Utilization	0.8
Tmin	100
$\frac{Tmax}{Tmin}$ ratio	100.0
$\frac{D}{T}$ ratio	1.0
$\frac{\kappa C}{C+\kappa C}$ ratio	0.1

Table I: Default Parameters

Finally, the max response time of a guest task over all phasings is obtained with Equation 14.

$$R_i^g = \max_{x \in \{E,A\}} R_i^{g,x} \tag{14}$$

A. Enforcement

Given that guest tasks are not trusted, their C_i needs to be enforced. In contrast to guest tasks, hyper-tasks are trusted and their κC_i does not need to be enforced. In addition, there can be two possible guest task termination options when the enforcement timer elapses: (i) the execution of the guest task τ_i is aborted and the corresponding hyper task κ_i is responsible for cleaning up its execution, or (ii) τ_i is deferred and its hyper task κ_i only executes temporary actions (e.g. safe actuation in a control task) allowing the guest task to finish in the next period.

B. Experiments

For our experiments define the following ranges:

- $U \in \{0.1, 0.2 \ldots, 1.0\}$
- $\frac{\kappa C}{C+\kappa C} \in \{0.1, 0.2, \ldots, 1.0\}$
- $\frac{TMAX}{TMIN} \in \{1, 10, 100, \ldots, 1000000\}$
- $\frac{D}{T} \in \{0.01, 0.1, 1.0\}$

For each combination generate 1000 tasksets and record schedulability and analysis time. The total utilization of the taskset is evenly divided into the number of tasks and the periods are chosen at random from the period range selected.

We perform five experiments to vary utilization $\frac{Tmax}{Tmin}$ ratio, number of tasks, $\frac{\kappa C}{C+\kappa C}$ ratio, and $\frac{D}{T}$ ratio.

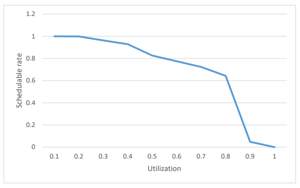
The default values for the parameters that do not vary are presented in Table I. Two observations are in order. First, the default number of task is set to 10 given that a larger number of task reduces the change having a schedulable taskset as can be seen in Figure 4a. And secondly, the default utilization is set to 80% also to reduce the influence of the utilization to dominate when varying the other parameters.

The graphs we will present are:

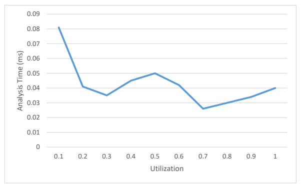
1) Execution Time Enforcement: All jobs of tasks τ_i are enforced not to exceed $C_i^{\zeta_i}$ units of execution. This is necessary, to ensure that a lower-priority higher-criticality τ_j does not suffer preemptions beyond those accounted in the schedulability equations.

IV. Related work

The contribution of this paper is to provide both security and real-time guarantees; and doing so by (i) combining a previously-proposed formally-proven hypervisor [20],

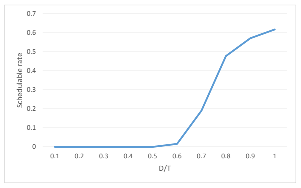


(a) Success rate

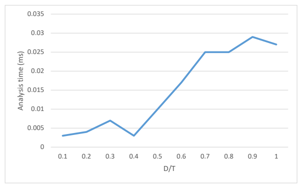


(b) Analysis time

Figure 2: As utilization grows

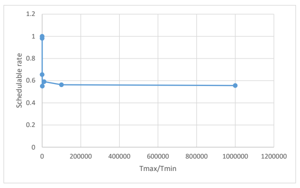


(a) Success rate

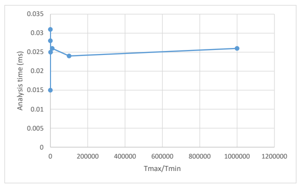


(b) Analysis time

Figure 6: as $\frac{D}{T}$ grows

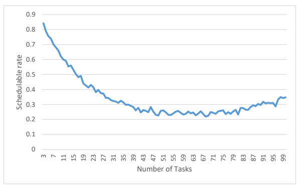


(a) Schedulability

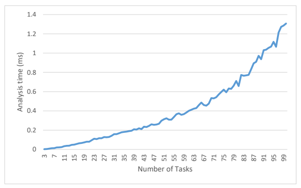


(b) Analysis time

Figure 3: As $\frac{Tmax}{Tmin}$ grows

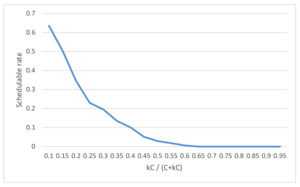


(a) Schedulability

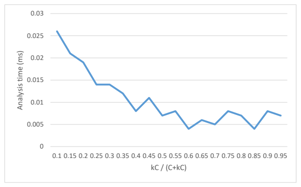


(b) Analysis time

Figure 4: as number of tasks grow



(a) Success rate



(b) Analysis time

Figure 5: as $\frac{\kappa C}{C+\kappa C}$ grows

(ii) extending this hypervisor to make it suited for real-time tasks, and (iii) present schedulability analysis for tasks running on the resulting system. This section describes previous work related to this contribution.

It is well-known that software of larger size tends to have more defects and it is also generally appreciated in the security community that it is desirable to formally prove correctness of security mechanisms—such arguments were made even back in the 1972 with ”The Anderson Report” which helped set the agenda for computer security research. There are several OS kernels that have been developed for security or with isolated parts; one of them is L4 which has been adapted for different purposes. The L4 kernel has been modified [13] to use hierarchical scheduling (see below) where the root scheduler uses time-triggered scheduling to decide which component should be active; however, there is also a background component (called background partition) that is always active. It is possible, however, for a task in the background component to have higher priority than a task in another component; in this case, a task in the background component can preempt a task in the other component. Also, because of the benefits of having small code base and isolation, researchers at Georgia Tech and Dresden [19] studied the problem of reducing the complexity of a kernel. They used a previously proposed approach (called Nizza) for re-architecting software to separate security-critical and non-security critical parts and applies this approach on three applications. They used a microkernel (L4) and a virtual machine running Linux. For similar reasons, researchers at NICTA created a small operating systems (OS) kernel (called seL4) and formally verified it [14]. The ideas were the following: (i) make the kernel small—move as many services as possible outside the kernel, (ii) verification is simplified by executing most kernel code non-preemptively and perform I/O—not with interrupts—but with pre-specified polling points, (iii) perform the verification with refinement, that is, show that each behavior of the concrete semantics of the implementation satisfies the abstract semantics, (iv) do the verification in two step: specification on the highest level, Haskell code as intermediate level, and C-code as the lowest level; show that the intermediate level is a refinement of the highest level and show that the lowest level is a refinement of the intermediate level, and (iv) use the theorem prover Isabelle/HOL.

The real-time systems community has studied hierarchical scheduling meaning that the system has schedulers on different levels. Typically, there is one root-level scheduler (sometimes called global scheduler) which decides at each instant which component/subsystem should be allowed to execute and then the selected component/subsystem has a local scheduler that decides which task in the component/subsystem should execute. The main driver behind the research in hierarchical scheduling is typically not security but instead the driver is typically to reduce the effort/cost of integrating components from different suppliers into a larger software system. The real-time systems community has developed schedulability analyses for hierarchical systems; these are verification procedures that take as input a model of a system and outputs a guarantee whether all tasks will meet their deadlines are run-time (see for example [10]). Typically, in hierarchical scheduling, one uses a method to compute the resource usage for each component and this becomes the timing interface of the component; then the schedulability test takes the timing interface of all components are input. One way to implement hierarchical scheduling is by letting the root scheduler be the scheduler in hypervisor and then let each component be a virtual machine and then a local scheduler is implemented by the guest operating system in a virtual machine. Typically hierarchical scheduling is used for systems where tasks in two different components are independent. It is noteworthy that the works in hierarchical scheduling do not solve our scheduling problem because we assume interaction between the guest task and the hypertask.

The real-time systems community has developed small OS kernels in order to improve security and also developed schedulability analysis for them. An example of this is Quest-V [16] which also has a corresponding schedulability analysis. Quest-V is intended to run on multicore processors and partition the resources, both processors and memory; and then run one guest operating system on each partition (called sandbox in [16]). In this way, software executing in one partition cannot write to memory belonging to another partition and cannot compete for processing resources that belong to another partition. A schedulability analysis for message passing between tasks in different partitions is also presented in [16]. In a similar spirit (but for a single processor system), researchers in Valencia [5] has created an OS kernel (called XtratuM) that provides time-partitions accordng to ARINC 653. Similar to other kernels mentioned above, the goal is to keep the size of the kernel small in order to achieve reliability; hence the kernel uses non-preemptive execution. The paper [5] does not offer schedulability analysis but there are techniques in hierarchical scheduling that offers that. Composite is the name of another operating system with similar goals (small kernel size and offering predictable timing). It is noted that managing resources shared across components places a special challenge [2]. Researchers have also noted the advantages of putting a hypervisor in hardware [12].

The real-time systems community has also considered confidentially; specifically information leakage between real-time tasks of different confidentiality levels when they are scheduled by a real-time scheduler [17]. Note that this is different from our work where we consider real-time requirements and integrity. From a distributed systems perspective, researchers have developed a security kernel that can tolerate some security violations [4]; it is implemented with RT-Linux under Linux.

The real-time systems community has also developed scheduling algorithms for real-time tasks where the tasks may have different criticalities and also different estimates on the worst-case execution time depending on the criticality level it is used for. The literature is vast—see [6] for an excellent survey. The work on mixed-criticality scheduling tends to ignore security aspects and tends to assume that the operating system is functioning. In our paper, however, we provide real-time guarantees even for the case that the guest operation system fails (because of a bug or security breach).

V. Conclusions

Software is increasing in complexity, to provide greater functionality and performance. Simultaneously, society has come to depend more and more on the services that computers perform. These are general trends but they have become particularly important in the area of autonomous systems (for example autonomous cars or UAVs) that perform perception and planning (that involves executing algorithms whose worst-case execution time or even termination are hard to prove) and must interact with the physical world in a safe way even if the complex function fails or does not finish within its expected worst-case execution time. In this paper, we have responded to this challenging situation with our proposal. Our proposal involves (i) a way of structuring software with one non-critical part (whose functional correctness is not trusted) and one critical part (whose functional correctness *is* trusted and its code base is small enough to be formally verified and it is structured to not depend on the non-critical part), (ii) a hypervisor that supports this way of structuring software, (iii) a task model that is suited for this way of structuring software, (iv) an exact schedulability test that takes a taskset described with this task model as input, and (v) an evaluation of this run-time system (the hypervisor and the Linux kernel together) showing that the hyperapps successfully perform functionality in the event that the part in the guest operation system fails.

Acknowledgment

Copyright 2018 IEEE. All Rights Reserved. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. DM18-0709

References

[1] Björn Andersson, Sagar Chaki, and Dionisio de Niz. Combining symbolic runtime enforcers for cyber-physical systems. In Shuvendu Lahiri and Giles Reger, editors, *Runtime Verification*, pages 68–84, Cham, 2017. Springer International Publishing.

[2] E. Armbrust, J. Song, G. Bloom, and G. Parmer. On spatial isolation for mixed criticality, embedded systems. In *WMC*, 2014.

[3] S. K. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. *Journal of Real-Time Systems*, 2003.

[4] M. Correia, P. Verissimo, and N.F. Neves. The design of a COTS real-time distributed security kernel. In *EDCC*, 2002.

[5] A. Crespo, I. Ripoll, and M. Masmano. Partitioned embedded architecture based on hypervisor: The XtratuM approach. In *EDCC*, 2010.

[6] R. Davis and A. Burns. Mixed-criticality systems—a review. In *Technical Report, University of York, Available at https://www-users.cs.york.ac.uk/burns/review.pdf*, 2018.

[7] Robert I. Davis, Alan Burns, Reinder J. Bril, and Johan J. Lukkien. Controller area network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, Apr 2007.

[8] Dionisio de Niz, Bjorn Andersson, and Gabriel Moreno. Safety enforcement for the verification of autonomous systems. In *Proceedings of SPIE*, 2018.

[9] Dionisio de Niz, Karthik Lakshmanan, and Ragunathan Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, RTSS '09, pages 291–300, Washington, DC, USA, 2009. IEEE Computer Society.

[10] A. Easwaran, I. Lee, I. Shin, and O. Sokolsky. Compositional schedulability analysis of hierarchical real-time systems. In *ISORC*, 2007.

[11] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: An extensible architecture for building certified concurrent os kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 653–669, Berkeley, CA, USA, 2016. USENIX Association.

[12] Z. Jiang, N.C. Audsley, and P. Dong. Bluevisor: A scalable real-time hardware hypervisor for many-core embedded systems. In *RTAS*, 2018.

[13] R. Kaiser and S. Wagner. Evolution of the PikeOS microkernel. In *First International Workshop on Microkernels for Embedded Systems*, 2007.

[14] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[15] John P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of the Real-Time Systems Symposium - 1990, Lake Buena Vista, Florida, USA, December 1990*, pages 201–209, 1990.

[16] Y. Li, R. West, Z. Cheng, and E. Missimer. Predictable commmunication and migration in the quest-v separation kernel. In *RTSS*, 2014.

[17] S. Mohan, M.-K. Yoon, R. Pellizzoni, and R. Bobba. Real-time systems security through scheduler constraints. In *ECRTS*, 2014.

[18] Special C. of RTCA. DO-178C, software considerations in airborne systems and equipment certification, 2011.

[19] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: Three case studies. In *Eurosys*, 2006.

[20] Amit Vasudevan, Sagar Chaki, Limin Jia, Jonathan M. McCune, James Newsome, and Anupam Datta. Design, implementation and verification of an extensible and modular hypervisor framework. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 430–444, 2013.