Mixed-Trust Computing for Real-Time Systems

Dionisio de Niz
SEI - Carnegie Mellon University, U.S.A.
dionisio@sei.cmu.edu

Bjorn Andersson
SEI - Carnegie Mellon University, U.S.A.
baandersson@sei.cmu.edu

Mark Klein
SEI - Carnegie Mellon University, U.S.A.
mk@sei.cmu.edu

John P. Lehoczky
Department of Statistics and Data Science - Carnegie Mellon University, U.S.A.
jl16@andrew.cmu.edu

Amit Vasudevan
SEI - Carnegie Mellon University, U.S.A.
avasudevan@sei.cmu.edu

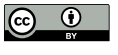
Hyoseung Kim
Department of Electrical and Computer Engineering - U.C. Riverside, U.S.A.
hyoseung@ece.ucr.edu

Gabriel A. Moreno
SEI - Carnegie Mellon University, U.S.A.
gmoreno@sei.cmu.edu

Abstract

Verifying complex Cyber-Physical Systems (CPS) is increasingly important given the push to deploy safety-critical autonomous features. Unfortunately, traditional verification methods do not scale to the complexity of this systems. One promising approach to tackle this problem is a runtime verification variant [3] where small pieces of code (called *enforcers*) are added to the system to watch the system output replacing it if it is deemed incorrect. Verifying these enforcers leads to a system-wide correctness guarantee. A system built this way combines trusted components (the enforcers) with untrusted ones (the rest of the system). In order for this combination to work in a real-time system, it is necessary to (i) protect the trusted code from failures of the untrusted code both in terms of memory and time, and (ii) use a coordination mechanisms between the trusted and untrusted parts in order to ensure that tasks meet their timing requirements. In this paper we present a new real-time scheduling framework to address these goals. This framework uses a new task model where each task is composed of an untrusted part guarded by a trusted one. We call them *mixed-trust tasks*. The untrusted part implements complex computations and has the support of a full OS running in a VM hosted by a trusted hypervisor. The trusted part, on the other hand, resides within the hypervisor. With this arrangement, if the untrusted part fails to finish (e.g., due to bugs) the trusted part is automatically activated to preserve safety properties (e.g., prevention of a crash). To support the new task model we present our new scheduling paradigm called *mixed-trust scheduling* that coordinates a fixed-priority non-preemptive scheduler protected by the hypervisor and a fixed-priority preemptive scheduler in the VM. We present a schedulability test and a procedure for configuring the parameters of the new task model. We also present our implementation based on the uberXMHF hypervisor running on a Raspberry Pi 3 and the Zero-Slack Rate-Monotonic (ZSRM) scheduler along with a coordination protocol designed to preserve trust. We present a set of experiments showing the system behavior during failures (transient and permanent) and a sample drone application to illustrate its use.

2012 ACM Subject Classification General and reference → General literature; General and reference



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Keywords and phrases Dummy keyword

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Certification authorities such as the FAA [23] allow the validation of different parts of a system with different degrees of rigor depending on their level of criticality. Formal verification has been recognized as a very important tool to provide correctness guarantees [2] for safety-critical components. Unfortunately, the verified property can be easily compromised if the verified components are not protected from unverified ones. Hence, **trust** requires that both verification and protection of components are jointly considered. This is the concept of trust we use in this paper.

One key limitation to building trust is the complexity of today's operating systems that makes them impractical (if at all possible) to verify. As a result, there has been a trend to achieve a trusted computing base (TCB) by developing small verified hypervisors (HVs) and microkernels, e.g., seL4 [19], CertiKOS [16], and uberXMHF [28, 29]. In these systems, trusted and untrusted components co-exist on a single hardware platform but in a completely isolated and disjoint manner. We thus call it *disjoint-trust computing*. Trusted components in the TCB are typically made small and simple due to the difficulty of verification. They are strictly isolated from untrusted parts hosted in a virtual machine (VM) where rich functionalities are implemented on full-scale guest OSs like Linux.

The fundamental limitation of disjoint-trust computing is that it does not allow the use of untrusted components for critical functionality that rely on verification to assure safety. This is because the verified components must be isolated from the untrusted ones in a disjoint-trust computing platform if they are to be trusted. For instance, this prevents the use of untrusted machine learning algorithms[1] to drive a car if such a functionality needs to be verified. Instead, a separate trusted component would need to be in charge of the driving, isolating it from any untrusted component. Unfortunately, the complexity of the critical functionality demanded today, e.g., autonomous driving, makes the verification of these components very difficult or impossible.

In this paper, we propose *real-time mixed-trust computing* (`RT-MTC`) which enables the combination of trusted and untrusted components within critical functionality. Untrusted components are monitored by verified components, a.k.a. *enforcers*, ensuring that the output of the untrusted components always lead to safe states (e.g., avoid crashes). To ensure trust, enforcers are protected by a (verified) micro-hypervisor [29]. These enforcers are known as *logical enforcers* [3, 12]. In addition, `RT-MTC` includes *temporal enforcers*, which are small, self-contained codeblocks that perform default safety actions (e.g., *hover* in a quadrotor) if untrusted components have not produced a correct output by the deadline, thus preserving the temporal requirement of the safe behavior. Temporal enforcers are contained as extensions within the base (verified) micro-hypervisor without jeopardizing the existing level of trust (e.g., using compositional verification offered by extensible micro-hypervisors [29]).

Our framework incorporates two schedulers: (i) a preemptive fixed-priority scheduler in the VM to run the untrusted components and (ii) a non-preemptive fixed-priority scheduler within the HV[2] to run trusted components. To verify the timing correctness of safety-critical applications in our mixed-trust framework, we propose a new task model and schedulability analysis. We also present

[1] For which the research community has not yet devised any effective verification method.

[2] Recent advances in formal verification [] have shown that single-threaded verification is more tractable. This simplifies the verification of both the scheduler and the code the scheduler run within the HV.

the design and implementation of a coordination protocol between the two schedulers in order to provide the protection and timing guarantees required by our framework. Lastly, we present an implementation of our proposed framework using uberXMHF [29], an open-source, compositionally verified micro-hypervisor framework supporting various security extensions. However, we note that in principle, our framework can also be instantiated with other verified micro-kernels or hypervisors provided they satisfy our requirements (see Section 2).

This work relies on innovations for code verification for the trusted parts that were presented in previous publications. Since that is out of the scope of this paper, we refer the readers interested in the compositional verification and isolation provided by uberXMHF to [29], the runtime verification conducted by logical enforcers to [3], and the formal verification of temporal enforcement code to [6].

The remainder of this paper is organized as follows. Section 2 presents our `RT-MTC` framework, an introduction to the runtime verification model that it supports, and the conditions that it must fulfill to preserve the verified properties of the model. Section 3 gives the system model. Section 4 presents schedulability analysis of mixed-trust tasks, including the evaluation of the schedulability analysis. Section 5 presents a fail-safe coordination protocol. Section 6 presents the implementation of mixed-trust scheduling. Section 7 discusses the related work and Section 8 concludes.

2 Mixed-Trust Real-Time Computing

The *Mixed-Trust Real-Time Computing* framework is designed to provide both logical and timing protection required to achieve trust in our previous runtime verification work [3]. In particular, a system in [3] is modeled as a state machine with a set of states $\mathbf{S}$ and a set of actions Σ; when we describe behavior, we let s in $\mathbf{S}$ be a state and α in Σ; be an action. The evolution of the system is modeled by the transition relation R, parameterized by the amount of time P that elapses during the transition, and the action applied at the start of the transition. Formally $R_P(\alpha) \subseteq \mathbf{S} \times \mathbf{S}$ is the relation such that if the action α is applied to the system at time t when it is in state s and subsequently the system is in state s' at time is $t + P$, then $(s, s') \in R_P(\alpha)$ — for a control-theoretic analysis see [24]. We then define $R_P(\alpha, s) = \{s' \mid (s, s') \in R_P(\alpha)\}$ as the set of states into which the system can transition after taking action α. We then identify ϕ as the set of safe states. Given these safe states we define a subset C_ϕ of ϕ-enforceable states as the largest set of states satisfying the following two conditions: $C_\phi \subseteq \phi \quad \text{and} \quad \forall s \in C_\phi \bullet \exists \alpha \in \Sigma \bullet R_P(\alpha, s) \subseteq C_\phi$.

We denote by $\mathit{SafeAct} : C_\phi \mapsto 2^\Sigma$ the mapping from ϕ-enforceable states to actions that will ensure that the system remains enforceable, i.e.,

$$\mathit{SafeAct}(s) = \{\alpha \mid R_P(\alpha, s) \subseteq C_\phi\}$$

The action α selected by the untrusted component in the system is then monitored and enforced by the logical enforcer. The logical enforcer, defined as $LE = (P, C_\phi, \mu)$, receives α from the untrusted component. It is therefore assumed that the logical enforcer executes with the same period P. $\mu(s) \in \mathit{SafeAct}(s)$ returns a set of enforcing actions. In each execution, the LE takes as input the current system state s and the system action α and produces an output action $\tilde{\alpha}$ defined as follows:

$$\tilde{\alpha} = \begin{cases} \alpha & \mathit{if}\ \alpha \in \mu(s) \\ \text{pick}(\mu(s)) & \mathit{otherwise} \end{cases} \tag{1}$$

where pick() selects one element from the set with whatever criteria. In the following we say that $\tilde{\alpha}$ is an LE-enforced action.

We now add to this model a temporal enforcer $TE = (E, C_\phi, \alpha_T)$ that executes periodically E time units after the untrusted component job arrives, takes the enforced action $\tilde{\alpha}$ from the LE and generates a temporally-enforced action $\hat{\alpha}$ before the end of the period as follows:

$$\hat{\alpha} = \begin{cases} \alpha_T & \mathit{if}\ \tilde{\alpha} = \perp \\ \tilde{\alpha} & \mathit{otherwise} \end{cases} \tag{2}$$

where (i) $\alpha_T \in \{\alpha | \alpha \in \mathit{SafeAct}(s) \forall s \in C_\phi\}$, that is, α_T is a safe action for any state in C_ϕ (i.e., the specific state s is not needed to calculate α_T) and (ii) $\perp$ denotes the absence of an action. In the following we say that $\hat{\alpha}$ is a TE-enforced action.

In this paper we present the software architecture, scheduling mechanisms, and schedulability analysis to execute a taskset where each task includes untrusted code guarded by both a logical and a temporal enforcer ensuring that not only the verification presented in [3] is preserved but also that the system can be **trusted** to remain in the safe statespace region ϕ according to our definition of trust.

We now define the conditions that our framework must satisfy to support this model. For the discussion of these conditions, we use *output* as the final action produced by the job once it has been evaluated by the logical and temporal enforcers. These conditions are defined as follow:

- **C1.** Each task must produce an output every period.
- **C2.** There is only one output per period produced either by the LE or the TE.
- **C3.** A task should not produce both an output produced by the TE and another by the LE within the same period.
- **C4.** An output produced by the task and validated by the LE must be the product of a computation that executes within a single period, i.e., that reads the state of the system (e.g., senses), computes an output, and generates the output within the same period.
- **C5.** The TE of a task must execute E time units after the arrival of the job it guards and finish before the end of the period.

To satisfy these conditions we not only need to create new runtime mechanisms, but the software also needs to be structured in a way that takes advantage of these mechanisms. This is the topic of our next section.

2.1 Software Architecture

Algorithm 1 shows the example behavior of a mixed-trust application. The **try** block shows the core of the infinite loop that periodically senses the state, computes an actuation, and issues the actuation. Within an iteration, the logical enforcer (LE) evaluates the computed actuation (α) and replaces it with a safe one ($\tilde{\alpha}$) if needed (as presented in (1)). However, this loop can fail if the code within the try block does not finish on time. Hence, a **catch** block is added to respond to a timeout (E time units after the start of the current period). If the timeout occurs, then the temporal enforcement actuation (α_T) is used to actuate. Note that it is not necessary to compute α_T based on the current state given that it is safe in any state within C_ϕ. Regardless of whether the actuation done by the try or the catch, it is immediately followed by a wait for the completion of the current period before executing another iteration. Note that in this algorithm there is no separate temporal enforcer but it takes the form of the **try-catch** construct effectively implementing (2).

Algorithm 1: Behavior of a Mixed-Trust Periodic Task

```
1  while true do
2      try:
3          s ← currentState()
4          α ← computeActuation(s)
5          α̃ ← LE(s, α)
6          actuate(α̃)
7      catch timeout(E):
8          actuate(α_T)
9      end
10     waitForNextPeriod()
11 end
```

Now, even if the LE and the TE are formally verified, Algorithm 1 can still fail to preserve trust in ϕ if (i) the behavior of the LE is modified (once modified we do not consider that the output

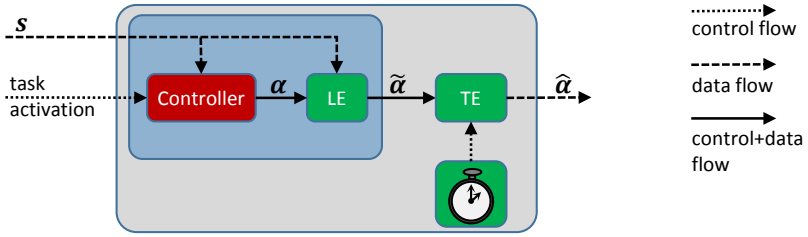


Figure 1 Software Architecture

is from the LE — **C2**), (ii) the system fails to issue one of the actuations $\tilde{\alpha}$ or α_T before the end of the period (**C1**), (iii) both $\tilde{\alpha}$ and α_T are issued within a period (**C3**), (iv) an $\tilde{\alpha}$ calculated in a previous period is issued (**C4**), or (v) the TE is modified (i.e., output is not considered to be a TE output — **C2**).

Based on the runtime assurance requirements and the failure possibilities presented above, we designed the software architecture presented in Figure 1. In this architecture, the green components are trusted and need to be protected from untrusted components (in red). Note that the LE requires the output of the controller (α) in order to calculate its output ($\tilde{\alpha}$) as presented in (1). Hence, while it can be (and must be) protected against logical behavior (code) modification, it cannot be protected against delays given that the untrusted controller can choose to delay its output at will. The TE, on the other hand, does not depend on $\tilde{\alpha}$ since it only uses it to decide whether or not to issue its safe action α_T. However, the TE still needs to be protected against logical behavior (code) modifications. Similarly, the communication of the $\tilde{\alpha}$ from the LE to the TE must also be protected against modification or falsification. Given this analysis, we define the following protection requirements:

- **P1.** Logical behavior protection. This requires protecting both the code and the related internal data. This is achieved through memory protection.
- **P2.** Temporal behavior protection. This requires protecting the activation time and the CPU bandwidth allocated in order to meet real-time deadlines.
- **P3.** Communication authentication. This implies that we can verify who is the source and ensuring that the source is protected (**P1**).
- **P4.** Communication logical integrity. This means that the message was not modified.
- **P5.** Communication temporal integrity. This means that the output generated is the product of a computation within a period.

It is worth noting that the protections listed above are the protections of trusted code from untrusted code from within the same task. Such requirements is a clear departure from other forms of protection between different tasks. This led us to name this new task a **mixed-trust task** whose timing characteristics will be formalized in Section 3. Before doing that, in the next section we first discuss the design of our runtime architecture in order to satisfy these inter-task protection requirements.

2.2 Runtime Architecture

As discussed in the introduction, we use a HV and its hosted VM to create a runtime environment to execute different parts of mixed-trust tasks. In order to design this runtime environment, we first design three protection domains to host the different parts, and a coordination protocol to preserve the temporal behavior of the overall mixed-trust task.

The domains we designed are:

Trusted Spatial protection Domain (TSD). This is where the LE executes. It offers trusted HV protection against memory modifications from the untrusted controller (or any other entity) but does not offer temporal protection.

Trusted Spatio-Temporal protection Domain (TSTD). This is where the TE resides. It offers trusted memory and temporal protection.

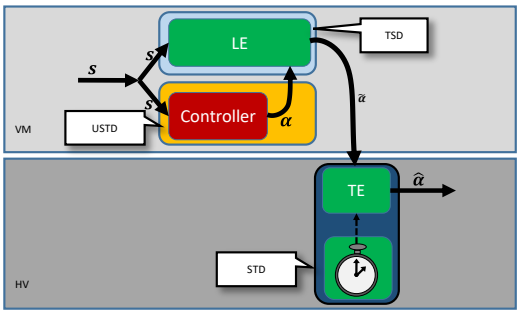


Figure 2 Runtime Architecture

Untrusted Spatio-Temporal protection Domain (USTD). This is where the untrusted controller resides. It offers untrusted VM spatial and temporal protection. This protection is untrusted because it is implemented in the VM, which is assumed to not be verified. However, it is expected its implementation is mature enough to work most of the time.

Figure 2 presents the location of these domains within the runtime architecture. This architecture allows us to (i) minimize the code added into the HV space, (ii) protect the LE and hence the integrity of the calculation of $\tilde{\alpha}$ (**P4**), (iii) validate the $\tilde{\alpha}$ origin by verifying the hypercall (syscall to hypervisor) origin (**P3** – not shown in the figure), (iv) provide trusted logical protection for the TE and the LE (**P1**), (v) provide trusted temporal protection for the TE (**P2**) and (vi) provide untrusted temporal and spatial protection (**P1**, and **P2**) to the controller.

In order to guarantee **P5**, we added a scheduler in the HV and a coordination protocol that synchronizes the scheduler in the VM with the one in the HV. Clearly, this coordination requires a new integrated analysis that will be presented in Section 4. Hence, we defer the discussion of the coordination protocol to Section 5. We now discuss the scheduling model.

3 System Model

Our system consists of a single-core processor[3] with a taskset $\Gamma = \{\mu_i | \mu_i = (T_i, D_i, \tau_i, \kappa_i)\}$. In the taskset, μ_i is a mixed-trust task with two execution segments, τ_i and κ_i, with period T_i and deadline D_i. The segment τ_i is considered to be untrusted and runs in the untrusted OS kernel inside the VM. The segment κ_i is considered to be trusted code and runs within the trusted HV. To represent the fact that these segments are handled by different schedulers, we consider them to be tasks and call τ_i the *guest task* (GT) and κ_i the *hyper task* (HT). These tasks are defined by:

$$\tau_i = (T_i, E_i, C_i) \tag{3}$$

$$\kappa_i = (T_i, D_i, \kappa C_i) \tag{4}$$

where T_i and D_i are the same as in μ_i, C_i is the worst-case execution time (WCET) of τ_i, and κC_i is the WCET of κ_i. Consider a particular job of μ_i, $(\tau_{i,q}, \kappa_{i,q})$. Ideally, $\tau_{i,q}$ will execute correctly taking no more than C_i time units and finishing within E_i time units of its activation. In this case, the job $\kappa_{i,q}$ is not activated. The logical enforcer (LE) verifies the correctness of $\tau_{i,q}$, while the timing enforcer (TE) verifies the timing. If the logical enforcer (LE) does not notify the HV that $\tau_{i,q}$ finished correctly and on time, then the corresponding HT $\kappa_{i,q}$ is activated and runs at a higher priority than any GT. The deadline for $\tau_{i,q}$, E_i, is chosen to ensure that $\kappa_{i,q}$ can finish by D_i, the μ_i deadline. We show how to calculate E_i in Section 4.

Under our mixed-trust scheduling paradigm, HTs running in the HV have strictly higher criticality than GTs in the VM. This leads to a scheduling hierarchy that strictly prioritizes all HTs over all

[3] For our current implementation we disable all cores but one in the Raspberry Pi 3. A multicore version is left for future work.

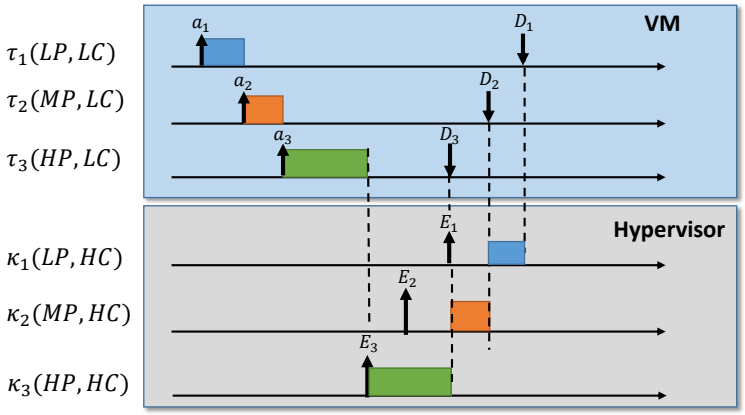


Figure 3 Mixed-Trust Sample Timeline

GTs. That is, the execution of a HT is not preemptible by any GT running in the VM, and a GT can be preempted by any HT that is ready to run. Furthermore, we assume that all HTs are non-preemptive, whereas GTs can be preempted.

Under normal operation, the mixed-trust task μ_i only runs its GT τ_i executing at most C_i time units, finishing within E_i time units and satisfying the LE, which then informs the scheduler of its correct completion. However, if any of these three conditions fails (e.g., due to an error in τ_i or the untrusted OS kernel or a security infiltration), then its execution is interrupted and κ_i is run within the HV. 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 the E_is in order to ensure that all GTs can finish by D_i if all GTs execute correctly, and all activated HTs can finish by D_i if their corresponding GTs do not complete correctly, that is the deadline of every task is met no matter whether any of the GTs execute correctly or not. Fig. 3 depicts a sample execution timeline for a mixed-trust taskset with three tasks. In this figure, HP/MP/LP represents high/medium/low priority on each scheduler, and HC/LC denotes high/low criticality.

4 Schedulability Analysis

The schedulability analysis of a mixed-trust taskset is performed in three steps: we first calculate the worst-case response time (R_i^κ) of each HT κ_i assuming non-preemptive fixed-priority scheduling[4]. Then, we calculate E_i for each GT τ_i by simply subtracting R_i^κ from the deadline D_i. Finally, we calculate the response time of each GT τ_i and check whether it is at most E_i.

4.1 Hyper Task Response Time

To calculate the HT response time, we use previous results on non-preemptive fixed priority scheduling (originally developed for the CAN bus) [11]. Specifically, the response time of a HT κ_i is calculated in three steps:

1. Inspired by [11], we define the level-i active period as a time interval in which the processor is busy at all times and (i) there is at most one job from a task with lower priority than the priority of task κ_i and if there is such a job, then it arrives before the beginning of the time interval, and (ii) for the rest of the time interval, there is only execution of jobs from tasks with priority higher than or equal to the priority of task κ_i. Our first step is to compute the maximum duration of a level-i active period. There are two reasons why we compute this maximum duration of level-i active period: (i) it allows us to compute the maximum number of jobs of task κ_i in the level-i active period, and (ii) we know that any execution outside the level-i active period cannot influence the response times of jobs of task κ_i in the level-i active period.

[4] We assume unique priorities across the paper.

2. For a given HT κ_i, the start time of a job from κ_i in the level-i active period is calculated. For each job of κ_i, the finishing time of this job is then calculated by just adding the execution time. This is because, once a task starts it cannot be preempted. Then, the response time of a job is calculated as the finishing time minus its arrival time.
3. For a given HT κ_i, the response time is computed from the maximum response time across all jobs of κ_i in the level-i active period.

Let t_i^κ denote the maximum duration of a level-i active period. Applying ideas in [11] on our model yields that t_i^κ can be calculated using:

$$t_i^\kappa = \max_{j \in \kappa L_i} \kappa C_j + \left\lceil \frac{t_i^\kappa}{T_i} \right\rceil \kappa C_i + \sum_{j \in \kappa H_i} \left\lceil \frac{t_i^\kappa}{T_j} \right\rceil \kappa C_j, \tag{5}$$

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

Given that a lower-priority task may be running when a higher-priority task arrives, (5) takes into account the maximum interference from one job of a lower-priority task.

Let $w_{i,q}^\kappa$ denote the latest starting time of the q^{th} job of κ_i in the level-i active period. Then, applying ideas in [11] on our model, we can obtain $w_{i,q}^\kappa$ using:

$$w_{i,q}^\kappa = \max_{j \in \kappa L_i} \kappa C_j + (q-1)\kappa C_i + \sum_{j \in \kappa H_i} (\left\lfloor \frac{w_{i,q}^\kappa}{T_j} \right\rfloor + 1)\kappa C_j. \tag{6}$$

The response time can then be calculated as follows. For the jobs in the level-i active period, we can move the arrival times of the jobs to be as early as possible without violating the T parameters; this may change the schedule but the duration of the level-i active period is non-decreasing and the starting time of each job is non-decreasing. Hence, it holds that the q^{th} job of each HT κ_j in the active period (including κ_i) arrives $(q-1)T_j$ time units after the level-i active period starts. For each job of a HT κ_i, we can add κC_i to its starting time and then subtract the arrival time of this job and this yields the response time of the job. With these ideas, we can compute the response time of κ_i as:

$$R_i^\kappa = \max_{q \in \{1 \ldots \left\lceil \frac{t_i^\kappa}{T_i} \right\rceil\}} (w_{i,q}^\kappa + \kappa C_i - (q-1)T_i). \tag{7}$$

It is worth noting that, in any schedulable taskset, the level-i active period of a HT κ_i includes only the execution of its first job if its corresponding GT τ_i has a $C_i > 0$. This is because if the taskset is schedulable, we verified that τ_i has time to run for C_i and hence no HT runs at that time. In other words, there is at least C_i time between two job executions (to completion) of a HT κ_i when τ_i executes. Notwithstanding, we keep the equation that considers active tasks with multiple job executions to allow tasksets without GTs.

Let us now discuss how to compute the E values. In order for HTs to be schedulable, clearly, we must choose E values such that:

$$\forall \mu_i \in \Gamma \quad E_i \leqslant D_i - R_i^\kappa. \tag{8}$$

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

$$\forall \mu_i \in \Gamma \quad R_i^g \leqslant E_i. \tag{9}$$

Ideally, we would like to develop an algorithm that computes E, for each task, such that if there exists an assignment of values of E to tasks such that the schedulability test deems the taskset schedulable, then our algorithm finds an assignment as well. In the appendix of the long version of this paper [1], we present such an algorithm. Unfortunately, its time-complexity is high. Therefore, we will calculate E_i in the following manner. For each τ_i, we assign E_i to be as large as possible while satisfying (8) and (9). The rationale for this is that choosing E_i to be as large as possible gives as much space as possible for R_i^g; however, it may adversely affect the R_l^g of a task τ_l with lower priority than τ_i. Based on this discussion, we choose to calculate E_i using (10).

$$E_i = D_i - R_i^\kappa. \tag{10}$$

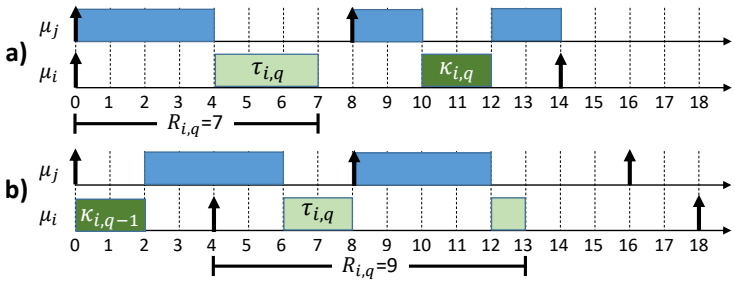


Figure 4 Aligning τ_i and τ_j leads to shorter $R_{i,q}$ (case (a)) than aligning κ_i and τ_j (case (b))

4.2 Guest Task Response Time

Recall that in the previous subsection, we defined the level-i active period. Since we now consider response times of GTs, we need a new notion that is similar but also incorporates interference from HTs. Therefore, we define the level-i *busy* period as a time interval such that at all times in the time interval, it holds that the processor is busy and there is only execution from jobs from HT or execution from jobs of GT of priority greater than or equal to the priority of τ_i.

Next, we will prove how to determine the worst-case phasings. We will argue in two steps, first considering the case in which the task under analysis does not have any HT. The proofs are variants of Theorem 1 in [20].

▸ **Lemma 1.** *In the case when a mixed-trust task μ_i does not have HT, the longest response time for all jobs of GT τ_i occurs in a level-i busy period initiated by the arrival of τ_i and the arrival of other higher-priority GTs or HTs and lower-priority HTs.*

Proof. Following the argument of Lehoczky [20], let $[0, b]$ denote a level-i busy period in which τ_i arrives at some point x_i after 0 during the busy period. Since the time before it arrives, $[0,x_i)$, is being used by higher-priority GTs, higher-priority HTs, or lower-priority HTs, moving its start to zero cannot change its completion time and can only increase its response time. Assume that a job of τ_i arrives at 0. Also assume that some higher-priority GT, τ_j, arrives after 0. From the definition of level-i busy period, it follows that for all jobs of τ_j that arrives strictly before 0, it holds that these jobs are not part of the level-i busy period and also the corresponding HT jobs are not part of the level-i busy period. Moving the arrival 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 HT arrives after 0. ◂

Now consider the case in which the GT does have an associated HT. Aligning other task arrivals with the arrival of GT τ_i does not necessarily cause the worst-case response for τ_i—see Fig. 4.

▸ **Theorem 2.** *In the case when a mixed-trust task does have a HT, the longest response time for all jobs of GT τ_i occurs in a level-i busy period for which the following holds:*

1. *for each mixed-trust with higher priority than μ_i it holds that either (i) the GT of the mixed-trust task arrives when the level-i busy period starts or (ii) the HT of the mixed-trust task arrives when the level-i busy period starts.*
2. *for the mixed trust task μ_i it holds that either (i) the GT of the mixed-trust task arrives when the level-i busy period starts or (ii) the HT of the mixed-trust task arrives when the level-i busy period starts.*
3. *for each mixed-trust task with lower priority than μ_i, it holds that the HT of the mixed-trust task arrives when the level-i busy period starts.*

Proof. The sequence of jobs in the level-i busy period will include a subsequence consisting of τ_i and κ_i jobs. If the subsequence begins with a job of τ_i, then Lemma 1 directly applies. Assume the subsequence begins with a job of κ_i.

Again let [0, b] denote a level-i busy period. Assume that κ_i arrives at some point in time after 0 during the level-i busy period. Unlike the previous case moving the start of κ_i to zero can reduce its response time since it is non-preemptible, but it can still only increase the response times of all of the GTs in the level-i busy period.

Assume that κ_i arrives at 0, but some higher-priority GT, τ_j, arrived after 0 but the previous job of κ_j finished executing before 0 (hence it is not part of the busy period). Again moving the arrival 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 GT jobs in the level-i busy period. Similar arguments can be used if a higher or lower priority HT arrive after 0. ◂

This theorem identifies the phasings that need to be explored to determine a GT's worst-case response time and motivate the equations that follow.

Previous work [5] has found the notion of a *request-bound function* useful in schedulability analysis. 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 that time interval. The request-bound function for a given task τ_i, for a given duration, is the maximum request-bound function that jobs of this task can generate in a time interval of this duration. Given its usefulness, we will now discuss how to create a request-bound function for our model; we will actually create a notion of request-bound function that takes additional parameters. Also, our notion of request-bound function gives the amount of execution of a mixed-trust task.

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 HV execution. Therefore, from the perspective of the request-bound function, this arrival of HV 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 mixed-trust task μ_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 HT of μ_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 GT of μ_i arrives. We use the latter parameter ($b \in \{0, 1\}$) to indicate whether the GT execution should be included in the execution counted in the request-bound function.

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

$$\text{rbf}_i^y(t, b) = \begin{cases} \left\lceil \frac{t-(T_i-E_i)}{T_i} \right\rceil^+ C_i b + \left\lceil \frac{t}{T_i} \right\rceil \kappa C_i & \text{if } y = E \\ \left\lceil \frac{t}{T_i} \right\rceil C_i b + \left\lceil \frac{t-E_i}{T_i} \right\rceil^+ \kappa C_i & \text{if } y = A \end{cases} \tag{11}$$

In the above definition, we use $\lceil x \rceil^+$ to mean $\max(0, x)$.

We will use this notion of request-bound function to compute the response time of the GT execution of a given mixed-trust task μ_i. Then, if it holds that the computed response time for each GT is less than or equal to its E parameter, then the taskset is schedulable (assuming that we have already checked HT schedulability). Therefore, our goal is now to present equations for computing the response time of a given GT. We will do so by presenting an equation for the maximum duration of a level-i busy period. Then, we compute the latest possible finishing time of a given job from a given task in this level-i busy period; then we 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) allow us to compute the response time of a GT 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 GT response time. We will compute the GT response time for two cases: the case that the given mixed-trust task arrives at the beginning of the level-i busy period and the case that the given mixed-trust task arrives with its HT aligned with the beginning of the level-i busy period. Given this high-level outline, we will now present the actual equations.

For each μ_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 the HT or the GT of μ_i arriving (x indicates which). Then,

in a similar spirit as (5), we can, for $x \in \{E, A\}$, for a given task τ_i, compute $t_i^{g,x}$ as follows:

$$t_i^{g,x} = \left(\sum_{j \in L_i} \text{rbf}_j^E(t_i^{g,x}, 0) \right) + \text{rbf}_i^x(t_i^{g,x}, 1) + \sum_{j \in H_i} \max_{y \in \{E,A\}} \text{rbf}_j^y(t_i^{g,x}, 1). \tag{12}$$

Given a task τ_i and a level-i busy period, we refer to job q as the q^{th} job with a GT 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 the HT or the GT of τ_i arriving (x indicates which). Then, in a similar spirit as (6), 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:

$$w_{i,q}^{g,x} = \left(\sum_{j \in L_i} \text{rbf}_j^E(w_{i,q}^{g,x}, 0) \right) + qC_i + (q - 1 + I_{(x=E)})\kappa C_i + \sum_{j \in H_i} \max_{y \in \{E,A\}} \text{rbf}_j^y(w_{i,q}^{g,x}, 1). \tag{13}$$

In (13), 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 the arrival of a job of the HT or the GT of τ_i (x indicates which). Then, in a similar spirit as part of (7), 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:

$$R_{i,q}^{g,x} = w_{i,q}^{g,x} - ((q-1)T_i + I_{(x=E)}(T_i - E_i)). \tag{14}$$

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 the arrival of a job of HT or GT of τ_i (x indicates which). Then, in a similar spirit as part of (7), we can, for $x \in \{E, A\}$, for a given task τ_i compute $R_i^{g,x}$ as follows:

$$R_i^{g,x} = \max_{q \in \left\{1 \ldots \left\lceil \frac{t_i^{g,x} - I_{x=E}(T_i - E_i)}{T_i} \right\rceil \right\}} R_{i,q}^{g,x} \tag{15}$$

Finally, the max response time of a GT over all phasings is obtained using:

$$R_i^g = \max_{x \in \{E,A\}} R_i^{g,x} \tag{16}$$

4.3 Monotonicity and how to solve the equations

We must address the solution of these equations to determine schedulability. We consider three cases depending on utilization.

1. $\sum_{\mu_i \in \Gamma} \frac{C_i + kC_i}{T_i} > 1$
   For this case, one can see that there is no solution to (12); the processor can be busy forever. Hence, q can be arbitrarily large; for arbitrarily large q, (13) implies that $w_{i,q}^{g,x}$ becomes infinite. This does not necessarily imply that the taskset is unschedulable but it can also be seen that for this case ($\sum_{\mu_i \in \Gamma} \frac{C_i + kC_i}{T_i} > 1$), for arbitrarily large q, $R_{i,q}^{g,x}$ becomes infinite as well. Thus, such tasksets are unschedulable. Hence, if, for a taskset, it holds that $\sum_{\mu_i \in \Gamma} \frac{C_i + kC_i}{T_i} > 1$, then we terminate the schedulability analysis and report *unschedulable*.
2. $\sum_{\mu_i \in \Gamma} \frac{C_i + kC_i}{T_i} < 1$
   For this case, one can see that (12) has a solution; hence, there is an upper bound on q. Then our schedulability analysis will terminate. Note that many of our equations are of the form z=f(z). This is true for (5), (6), (12), and (13). For each of these equations, it holds that the expression on the right-hand side is monotonically non-decreasing in the variable on the left-hand side. Hence, these equations can be solved with fixed-point iteration.

3. $\sum_{\mu_i \in \Gamma} \frac{C_i + kC_i}{T_i} = 1$
   For this case, it is difficult to determine whether (12) has a solution. We will pessimistically report *unschedulable* for such tasksets.

4.4 Generalizations

It is worth noting that a taskset is not required to have tasks with both GT and HT. More importantly, when a taskset contains no HT our scheduling equations reduce to fixed-priority preemptive response time analysis. Similarly, when a taskset contains no GT, then it reduces to fixed-priority non-preemptive schedulability analysis.

We can also observe these properties realized in a running system. For example, even if GTs have their corresponding HTs, the system will run like a "preemptive" scheduling system if all GTs finish without exceeding their C_i. On the other hand, if a VM crashes, the system will run like a pure non-preemptive scheduling system as will be shown in Figure 10 of Section 6.4.

4.5 Enforcement

Given that GTs are not trusted, their C_i values need to be enforced. This enforcement allows us to implement a graceful degradation scheme by preventing failing GTs from interfering with other non-failing GTs. Clearly, if a failure affects the kernel in the VM (e.g., due to a security attack) all the GTs will be compromised but the HV and the HT will be protected from the failure. In contrast to the GTs, the HTs are trusted and their κC_i does not need to be enforced. In addition, there can be two possible GT enforcement options when the enforcement timer elapses: (i) the execution of the job of a GT τ_i is aborted immediately and the corresponding HT κ_i is responsible for cleaning up its execution, or (ii) the job of the GT τ_i is deferred (suspended) and its HT κ_i executes only temporary actions (e.g., safe actuation in a control task), allowing the GT's job to complete in the next period. Our current implementation uses the latter option and we will call it *deferral GT enforcement*.

4.6 Experiments

This section presents experiments that show how taskset parameters influence schedulability. We found three reasons that impact schedulability:

1. When considering one task, its HT can experience interference from HT of other tasks. In addition, the HT of other tasks can also delay the execution of the GT of the task under consideration. This is a double-accounting effect that has no analog in classic fixed-priority scheduling.
2. A GT can experience a long delay because of execution of HT of all the other tasks. This reason is most impactful when the GT has very small period.
3. Consider the lowest-priority task and consider its GT. When its period is around $\sqrt{2}$ of the period of its higher priority task, then the schedulability deteriorates (just like it does for classic rate-monotonic preemptive scheduling).

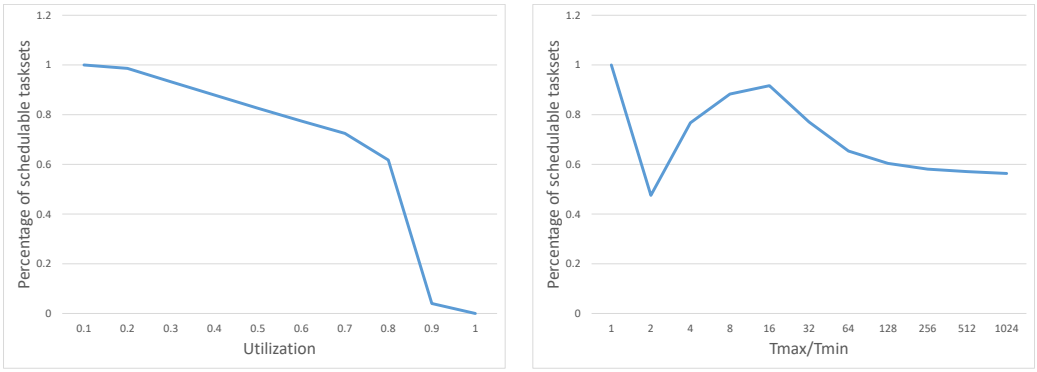
We now illustrate the schedulability conditions we just introduced with the following experiments. In these experiments we vary: (1) the taskset utilization, (2) the ratio between the maximum period and the minimum period, (3) the number of tasks in the taskset, (4) the ratio between the HT WCET and the sum of the GT and HT WCET, and (5) the ratio between the deadline and the period of the tasks. The ranges of variations are presented in Table 1.

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 1. Two observations are in order. First, the default number of tasks is set to 10 given that a larger number of tasks reduces the chance of having a schedulable taskset as can be seen in Fig. 6a. Second, the default utilization is set to 80% also to reduce the influence of the utilization to dominate when varying the other parameters.

Parameter	Range	Default
Number of Tasks	$\{3, 4, \ldots, 200\}$	10
U	$\{0.1, 0.2, \ldots, 1.0\}$	0.8
$\frac{\kappa C}{C+\kappa C}$	$\{0.1, 0.2, \ldots, 1.0\}$	0.1
$\frac{T_{max}}{T_{min}}$	$\{1, 2, 4, \ldots, 1024\}$	100.0
$\frac{D}{T}$	$\{0.1, 0.2, \ldots, 1.0\}$	1.0
Tmin		1000

Table 1 Parameter Ranges and Defaults



(a) As utilization grows

(b) As $\frac{Tmax}{Tmin}$ grows

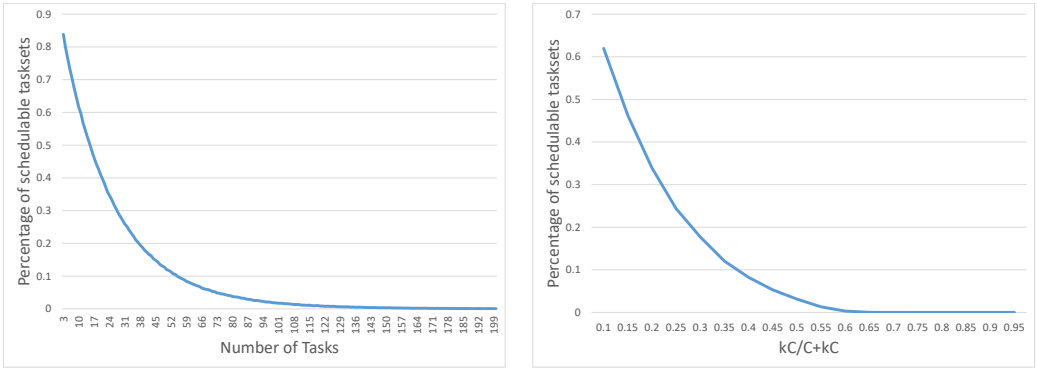
Figure 5 Success rate as utilization and $\frac{T_{max}}{T_{min}}$ grow

In the experiments each data point is the average of 100,000. Specifically, each dot is computed from 100,000 tasksets and we compute the percentage of schedulable tasksets.

Each taskset is generated with the selected number of tasks. Each task is assigned a utilization equal to the selected total utilization of the taskset divided by the number of tasks. Then the period of the task is chosen at random (with uniform distribution) from the period range selected.

Fig. 5a shows the percentage of schedulable tasksets as the taskset utilization grows from 10% to 100%. Note that the experiment shows a decline in the percentage of schedulable taskset just after 20%. This is due to reason 2.

Fig. 5b depicts the percentage of schedulable tasksets as the ratio of the minimum and maximum period grows. Here, we can see that when increasing the ratio, the success rate decreases and then increases and then decreases again. The initial decrease is caused by reason 3; the second decrease is caused by the reason 2.



(a) As number of tasks grow

(b) As $\frac{\kappa C}{C+\kappa C}$ grows

Figure 6 Success rate as as number of tasks and $\frac{\kappa C}{C+\kappa C}$ grow

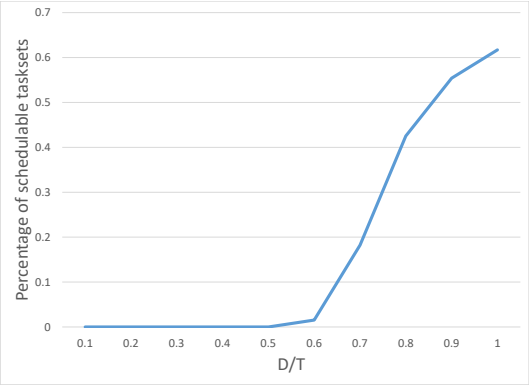


Figure 7 Success rate as $\frac{D}{T}$ grows

Fig. 6a presents the fraction of schedulable tasksets as the number of tasks in the taskset grows. The curve decreases exponentially reaching zero at about 115 tasks. This is because when the number of tasks increase, the maximum period divided by the minimum period among the tasks generated becomes larger and then reason 2 becomes more impactful.

Fig. 6b shows schedulable tasksets percentage as the ratio of HT WCET to the combined HT and GT WCET grows. The figure shows a quick drop in the percentage of schedulable tasksets as this ratio increases. This is because of reason 2.

Fig. 7 presents the percentage of schedulable tasksets as the ratio of deadline to period grows. As expected, when deadlines are much smaller than periods tasks are not schedulable. The situation starts to improve after a ratio of 0.6 when we start finding schedulable tasksets. Then the percentage of schedulable tasksets grows fast, and the rate of improvement reduces at the end of the ratio range.

5 Fail-Safe Mixed-Trust Scheduling Coordination Protocol

With the timing analysis as a background we can now discuss the coordination protocol between our schedulers. A key challenge that needed to be solved in our framework was the prevention of any dependency of trusted HV and HT code from the untrusted code, while still enabling the successful coordination of the HV and guest schedulers. The dependency of a higher-critical component from a lower-critical one is known as dependency inversion [14].

Preventing the dependency inversion problem necessitates three mechanisms: (1) a Secure HT Bootstrapping (*SHTBoot*) Protocol, (2) a Fail-Safe HT Triggering (*FSHTrigger*) mechanism, and (3) a Late-Output Prevention protocol (*LOP*).

5.1 Secure HT Bootstrapping (*SHTBoot*)

The objective of the SHTBoot protocol is to ensure that the HT can start and execute periodically according to its specification even if the VM is unable to bootstrap the GT. This is in fact necessary to properly implement the trusted temporal protection of the **TSTD** and the protection requirement **P2** (discussed in Section 2). We leverage secure boot mechanism provided by uberXMHF to ensure that the micro-hypervisor framework is the first to get control when the system is powered on. The SHTBoot protocol starts the HTs and GTs independently out of bootstrapping task tables stored in the HT and VM storage, respectively. To synchronize the periodic activation of a GT with the corresponding enforcement timer of its HT, the guest scheduler requests the start time of the next period from the HV and uses this time to start the first job of the GT, aligning the periodic activations of the GTs and HTs as required.

5.2 Fail-Safe HT Triggering Protocol (*FSHTrigger*)

The objective of the FSHTrigger is to prevent a failure in the VM from disabling or corrupting the periodic activation of the HTs. This is in fact the activation side of the protection requirement **P2**.

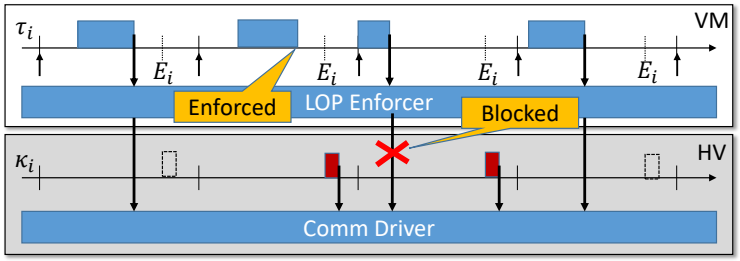


Figure 8 Late-Output Prevention Protocol

To implement the FSHTrigger, a general strategy followed is to program separate timers for the VM and the HV down to the hardware level so that untrusted VM code can program its own timer but cannot program the HV timer. This way when a task μ_i is started, its E_i timer is programmed within the HV and will always trigger no matter what code executes in the VM (even malicious code actively trying to disable the timer). As a result, the HT can always run on time and complete its safety action by the deadline. We leverage the peripheral isolation provided by uberXMHF to isolate the HV and guest timers so that the guest cannot access the HV timer.

Even though an HT must be isolated from failures of the corresponding GT, some of their timing parameters need to be synchronized: (1) initial release offset, which is needed to program an enforcement timer exactly E_i units after the arrival of each job, and (2) job completion time, which is to disable the HT if it completed before E_i. The creation time synchronization is performed by SHTBoot. For the completion-time synchronization we rely on the logical enforcer in the VM to send the completion signal to the HV. More specifically, the logical enforcer (LE) verifies that the output produced is safe (or replaces it with a safe one) and sends the completion signal. To guarantee that the LE only sends the completion signal when a safe output is generated, the HV protects the LE memory and we assume that the LE code's trustworthiness has been asserted (e.g. through verification). This satisfies the protection requirement **P1** and **P3**. The only possible failure is then a denial-of-service, i.e., **P2** is not guaranteed. In particular, if the GT τ_i takes longer than C_i to complete, the task will be enforced and the LE will neither complete nor send the completion signal to the HV before E_i. However, this failure is part of the assumption of the FSHTrigger protocol since the absence of the completion signal will trigger the HT and issue the safe output. In other words, the HT that host the TE preserves it temporal behavior (**P2**).

5.3 Late-Output Prevention (LOP) Protocol

A late output may occur when a GT job is allowed to complete and generates an output (e.g., actuation commands) E_i time units or more after its arrival. Recall the deferral GT enforcement approach explained in Section 4.5. In this case, a job can be suspended once E_i time units have elapsed after its arrival, and resumed in the next period, allowing it to send its output in the second period (violating requirement **C4**). Preventing this output is important because the logic in the application algorithm (e.g. control algorithm) assumes that it is computed within the execution of a single job, perhaps using inputs (sensing) from the beginning of the period that are only valid for output (e.g. actuation) during this same period (**C4**). To solve this problem, the output and completion signals are bundled in a single call and all the output is mediated by an LOP enforcer in the VM kernel scheduler, making sure that the output is discarded if it is sent after E_i. We also protect the LOP enforcer memory (in protection domain **TSD**) in the kernel and assume that the trust in its code has been established (e.g. through verification). As a result, the LOP enforcer can only fail by not sending the output. The use of the LOP enforcer separately from the LE allows us to separate the logical from the temporal correctness, simplifying its verification but preserving both properties.

Figure 8 depicts an example of the execution timeline of the LOP protocol, where the second job of GT τ_i is budget-enforced and does not send the output and completion signal in that period. The absence of this signal triggers the HT (red rectangle), which in turn sends the safe output. When the GT is resumed in the next period, it produces an output but the LOP blocks it along with the

completion signal. Hence, the HT is activated again to send the safe output.

6 Implementation

Our scheduling mechanisms are implemented by the combination of the budget enforcement of the ZSRM kernel scheduler [13] running in a VM and a non-preemptive fixed-priority scheduler implemented within the verifiable and extensible uberXMHF micro-HV [28, 29, 27] running in a Rasberry Pi-3 platform. In order to prevent failures in the VM from affecting the code within the micro-HV, uberXMHF implements two-stage hardware memory page-tables and protections that cannot be modified by the guest OS running inside the VM. The two-stage hardware page tables isolate the micro-HV memory where the HTs reside. The guest OS memory isolation and protections have been formally verified as presented in [28, 29]. uberXMHF employs a compositional verification methodology that allows the addition of security sensitive functionality as modularly protected and verifiable components (called *uberapps*). We used this facility to implement the HV scheduler as a uberapp.

6.1 Hyper-Task-Aware Budget Enforcement

ZSRM is a Linux kernel module scheduler that works on top of the fixed-priority preemptive scheduler of the Linux kernel. The original ZSRM budget enforcement is performed by shadowing the priority queues of the fixed-priority scheduler to keep track of the current top-priority active task and the amount of CPU time this task consumes. Tasks become ready to run when they are created, and they go to sleep by calling the ZSRM API `wait_next_period()` when they finish their periodic job execution. At this time, the shadow priority queues in ZSRM are updated, and the top-priority active task in the queue is scheduled and marked as the `current` task. Similarly, when a task period elapses, ZSRM wakes the task up and makes a scheduling decision. Whenever a task is scheduled, an enforcement timer is set to expire with the maximum remaining budget of that task. If the task either finishes or is preempted by another task before the enforcement timer expires, the timer is canceled and reprogrammed for the next task. On the other hand, if the timer expires, the task is suspended until its next periodic timer expires. The budget accounting is implemented by recording a starting timestamp when a task becomes the `current` task and a finishing timestamp when the task is either preempted or completes its periodic execution. Then, subtracting the starting timestamp from the finishing one, gives us the CPU time used by the task. We accumulate this time for all the intervals that a task is considered to be the currently executing task in each period. Then the budget minus this accumulated CPU time consumption is used to program the enforcement timer every time the task becomes the currently executing task. The implementation of this budget enforcement mechanism was formally verified in [6].

Given that HT preemptions are invisible to the VM and ZSRM scheduler, the ZSRM budget enforcement fails to account for them. To handle this, we use an event logger within the HV to record the timestamps of the activation and completion of the HTs. Then, these events are requested from the HV when the budget enforcement timer triggers. At this time, we use the HT timestamps to discount the HT preemption time from the budget. Finally, if this discount reduces the budget consumption to less than its C_i budget, the budget enforcement timer is reprogrammed to allow the GT to keep executing for the difference.

6.2 Implementation Overhead

Table 2 shows the average overhead of our implementation in the Raspberry Pi-3 running at 1200 MHz. A few observations are in order: (i) The context switch includes the shadow queue manipulation in the ZSRM kernel module but not the kernel scheduling queues. (ii) The budget enforcement includes the activation of a kernel thread that is used to prevent modifying the priority queues of the kernel from inside the timer interrupt context, as well as the hypercall to obtain the hypertask preemptions. The hypercall to get the hypertasks preemptions is also shown separately.

Procedure	Avg Overhead (ns)	WC Overhead (ns)
Job Arrival with context Switch	12368	14105
Job Arrival, no context switch	7789	9052
Budget Enforcement	26263	42894
wait_next_period()	7789	9894
Get Hypertask Preemptions	7473	11315

Table 2 Overhead

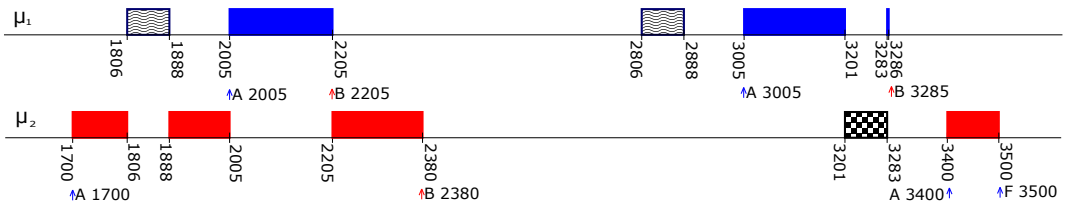


Figure 9 Mixed-Trust Taskset Execution Timeline (timestamps in 10^{-4} seconds)

6.3 Spurious Temporal Failure Illustration

We ran a taskset of two tasks with our implementation to illustrate the mixed-trust scheduler. This taskset allowed us to obtain a plot of the tasks with timestamps captured from both the kernel scheduler and the HV scheduler. Both schedulers query the same hardware timer to read their timestamps, allowing us to have an integrated timeline without incurring context-switch penalties.

Figure 9 shows the execution of two mixed-trust tasks in a timeline reconstructed from timestamps for the following events: (i) **arrival (A)** that marks when the job of the task becomes ready to execute; (ii) **job finishing (F)** by calling `wait_next_period()` from a guest application task to the kernel scheduler when its job ends normally; (iii) **budget enforcement (B)**; (iv) **resume** marks the start of a colored rectangle showing when the job starts to execute; and (v) **paused** presented as the end of a colored rectangle. It is worth noting that the activation of the HTs is presented as a small rectangle marked by the **resume** and **paused** events and filled with wavy and checkered patterns. The timeline shows different types of preemptions as follows. From 1806 to 1888, μ_1's HT (κ_1) preempts μ_2's GT (τ_2). Then from 2005 to 2205, τ_2 is preempted again this time by τ_1. At 2205, τ_1 is budget-enforced letting τ_2 to run until it is enforced at 2380. Given that τ_1 did not signal completion, its HT κ_1 executes from 2806 to 2888. Then the next job of τ_1 arrives at 3005 and executes until it is preempted by the HT κ_2 at 3201. κ_2 finishes its execution at time 3283 allowing τ_1 to resume execution until it is enforced at 3285. The last job execution shown in the timeline is a normal that start at 3400 and finishes at 3500.

6.4 Permanent Failure Illustration

In this section we present an experiment to show how our approach handles a complete failure of the kernel in the VM. More specifically, we start two tasks (μ_1 and μ_2) in this case running with a periodicity of $T_1 = 1$ and $T_2 = 2$ seconds, respectively, with their respective HTs designed to run for only 10 ms each. We let the task μ_1 run for four periods and μ_2 for three periods without faults (not even timing faults so their HT do not trigger). The third job of τ_2 uses a semaphore to signal a third task (not shown) whose only role is to wait for this signal and invoke a system call in our scheduler specifically designed to test a full kernel failure (this scheme allows τ_2 to properly finished given that the third task has the lowest priority). This call, in turn, calls the kernel `panic()` function. The `panic()` function is designed to completely stop the kernel when a part of the kernel reaches a point where it is not possible to continue executing due to unrecoverable failures. This basically simulates a hard crash. In order to capture the timestamps, we send them to a serial port through the HV. However, because the serial port driver implementation is slow, it creates some disruption in the timestamps. The resulting trace is presented in Figure 10.

A few observations about the trace in Figure 10 are in order. First, the call to `panic()` occurs

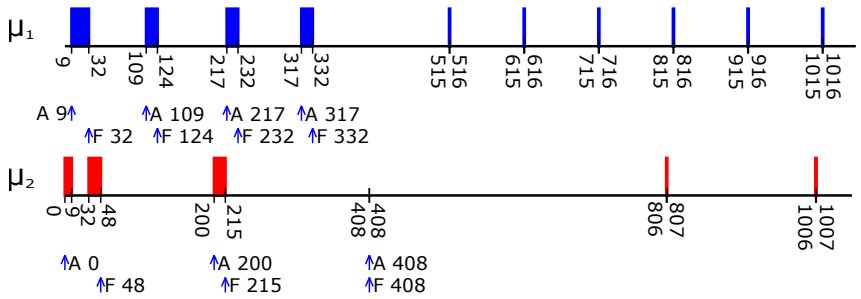


Figure 10 Experiment with Permanent VM crash (timestamps in 10^{-2} seconds)

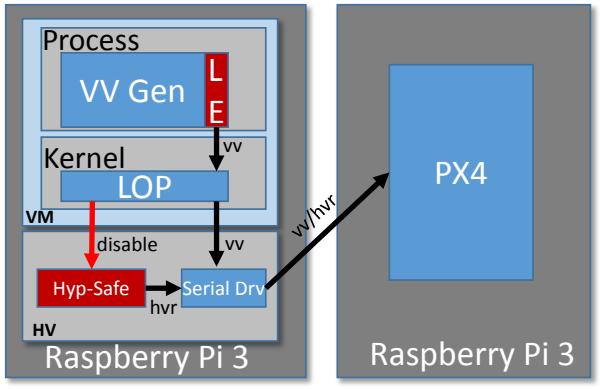


Figure 11 Drone Application Architecture

after τ_2 finishes at time 408. Second, after this time no more arrival (**A**) or finishing (**F**) events occur from either of the tasks. Third, as expected both HTs (κ_1 and κ_2) continue executing periodically. Finally, the first execution of both HTs after the `panic()` call occurs almost two periods from the last GT executions. In particular, the first execution of κ_1 after the `panic()` call occurs at 515 that is almost two periods from the last arrival of its GT at 317. Similarly, for κ_2 its corresponding first arrival after the `panic()` call is at 806 and its last GT arrival at 408. This is expected because the HT execution is scheduled at the end of the period of a task. This means that when the GT executes at the beginning of the period, the execution of the HT of the following period will happen almost two periods apart, even though an output is produced in every period.

6.5 Illustrative Application

We implemented a sample mixed-trust application of a drone mission. This application consists of two components: (i) the mission controller, which generates velocity vectors (VV) that the drone must fly to follow a route, and (ii) the Pixhawk[5] flight controller running the PX4 autopilot[6] in off-board mode, which makes the drone fly in the direction and speed of the last VV received. The mission controller runs in its own processor sending a VV message every 50 ms to the processor where the flight controller runs. A logical enforcer in the mission controller prevents the drone from violating spatial constraints (e.g., a virtual fence or the collision volume of other drones [7]). In addition, we added an HT to the mission controller can take a safe action and continue to send VV messages to the flight controller in case the GT fails.

Figure 11 depicts this structure, which, for simplicity of presentation, shows only one mixed-trust

[5] `https://pixhawk.org/`
[6] `http://px4.io/`

task. This task has a guest task that generates velocity vectors (*VV Gen*) and a hypertask (*Hyp-Safe*) that generates the safe drone action *hover* (*hvr*), which is a null VV. The figure also shows the LOP mediation of the messages sent by *VV Gen* ensuring that (a) no late outputs are allowed, and (b) when no output is generated by *VV Gen* the *Hyp-Safe* HT generates the *hover* action. The mission controller was implemented using a version of DronecodeSDK[7] that we modified to handle serial communications through the serial driver in the HV, and to use the bundled output and completion signals for the LOP. We ran this application using hardware-in-the-loop simulation (i.e., actual mission and flight computers connected to a drone simulation), which allowed us to observe the physical consequences. We tested both temporary failures and hard failures where we verified that both the LOP and the HTs properly prevent drone failures.

7 Related work

Previous work has recognized that small operating systems kernels can be more reliable [18] and can be formally verified [19]. In this context, decomposing an application can provide security benefits as well [26]. But they do not provide schedulability analysis. Previous work on hierarchical scheduling (e.g., [15]) studies run-time systems with two schedulers and they present theories that provide real-time guarantees but they do not consider a task that spans different components. Operating systems works considering real-time requirements have also been presented [21, 9, 4, 17, 30] to achieve isolation and some offer offline schedulability tests but not for the task model that we consider (where a task can span two operating systems) and they do not target formal verification of operating system code.

Previous work [22, 8] combining real-time and security are not based on a runtime verification framework that requires the integration of trusted and untrusted components.

Works on mixed-criticality (see [10] for an excellent survey) share our goal of monitoring run-time behavior and taking action when behaviors that are abnormal are detected; however, we are not aware of any work on a mixed-criticality scheduler that considers our task model and uses a formally verified kernel.

Simplex [25] is an architecture for designing controllers. It comprises one complex controller, a simple controller, and two sets of states. The first set describes the safe states; the second set describes when there is a need to transition between controllers. The complex controller is allowed to operate when the plant is in the second set. If the plant leaves this set, then the simple controller takes over. With this architecture, the complex controller can be optimized for performance and does not need to be verified; the simple controller, however, is verified to make sure that the plant is always in a safe state. One can think of the simple controller in Simplex as somewhat analogous to our HT.

8 Conclusions

In this paper we presented a real-time mixed-trust framework for mixed-trust tasks composed of an untrusted part (a.k.a. guest task (GT)) that runs in a VM guarded by a temporal enforcer (a.k.a. the hyper task (HT)) running in the trusted HV that hosts the VM. With this arrangement if the untrusted GT fails to complete on time, the HT is automatically triggered executing a safe computation to keep the system safe (e.g., avoiding a crash). We presented the new real-time schedulability analysis and parameter configuration algorithms along with experiments that explore the sensitivity of our analysis to different parameters. We also presented the implementation of our framework based on the coordination of a non-preemptive fixed-priority scheduler within the uberXMHF HV and the ZSRM scheduler in the VM. We also discussed the three coordination mechanisms used to preserve trust even in the presence of failures and presented experiments to show the behavior of our system during transient and permanent failures. Finally, we presented a sample drone application to illustrate the use of our framework. The contributions presented in

[7] `https://www.dronecode.org/`

this paper enable the use of complex non-deterministic software (e.g., machine learning) in safety-critical systems (including dependencies on large OS and libraries) while still preserving provable logical and timing guarantees. Furthermore, informed by runtime verification, our framework avoids taking actions that, while preserving timing properties, break logical properties, e.g., killing a job disregarding the consequences of the absence of actuation. This is, in fact, the result of the co-design of a logical and timing runtime verification scheme that gave birth to our mixed-trust scheduling framework.

Acknowledgment

Copyright 2019 Carnegie Mellon University, Hyoseung Kim and John P. Lehoczky. 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. NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.
Please see Copyright notice for non-US Government use and distribution. Carnegie Mellon® is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.
DM19-0109

References

1 Mixed-Trust Scheduling. `https://www.dropbox.com/s/8uthfqkjarsbbre/mixed_trust_scheduling-tr.pdf?dl=0`, 2018.

2 RTCA Special Committee 205. Formal methods supplement to DO-178C and DO-278A, 2011.

3 B. Andersson, S. Chaki, and D. de Niz. Combining symbolic runtime enforcers for cyber-physical systems. In *RV*, 2017.

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

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

6 S. Chaki and D. de Niz. Formal verification of a timing enforcer implementation. *ACM TECS*, 2017.

7 M. C. Consiglio, J. P. Chamberlain, C. A. Munoz, and K. D. Hoffler. Concepts of integration for UAS operations in the NAS. In *Congress of the International Council of the Aeronautical Sciences (ICAS)*, 2012.

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

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

10 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.

11 R.I. Davis, A. Burns, R.J. Bril, and J.J. Lukkien. Controller area network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 2007.

12 D. de Niz, B. Andersson, and G. Moreno. Safety enforcement for the verification of autonomous systems. In *Proceedings of SPIE*, 2018.

13 D. de Niz, K. Lakshmanan, and R. Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *RTSS*, 2009.

14 H. Ding, L. Arber, L. Sha, and M. Caccamo. The dependency management framework: a case study of the ION cubesat. In *ECRTS*, 2006.

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

16 R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *OSDI*, 2016.

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

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

19 G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *SOSP*, 2009.

20 J.P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *RTSS*, 1990.

21 Y. Li, R. West, Z. Cheng, and E. Missimer. Predictable commmunication and migration in the Quest-V separation kernel. In *RTSS*, 2014.

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

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

24 D. Seto, J. P. Lehoczky, L. Sha, and K. G. Shin. On task schedulability in real-time control systems. In *17th IEEE Real-Time Systems Symposium*, pages 13–21, Dec 1996. doi:10.1109/REAL.1996.563693.

25 L. Sha. Using simplicity to control complexity. *IEEE Software*, 2001.

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

27 A. Vasudevan and S. Chaki. Have your PI and eat it too: Practical security on a low-cost ubiquitous computing platform. In *IEEE Euro Symposium on Security and Privacy*, 2018.

28 A. Vasudevan, S. Chaki, L. Jia, J. M. McCune, J. Newsome, and A. Datta. Design, implementation and verification of an eXtensible and Modular Hypervisor Framework. In *2013 IEEE Symposium on Security and Privacy, SP*, 2013.

29 A. Vasudevan, S. Chaki, P. Maniatis, L. Jia, and A. Datta. überspark: Enforcing verifiable object abstractions for automated compositional security analysis of a hypervisor. In *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

30 S. Xia, J. Wilson, C. Lu, and C.D. Gill. RT-Xen: Towards real-time hypervisor scheduling in Xen. In *EMSOFT*, 2011.