Mixed-Trust Computing for Real-Time Systems

D. de Niz[1], B. Andersson[1], M. Klein[1], J. Lehoczky[1], A. Vasudevan[1], H. Kim[2], G. Moreno[1]
[1]Carnegie Mellon University [2]University of California, Riverside

***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 these systems and do not provide systematic methods to protect verified properties when not all the components can be verified. To address these challenges, this paper proposes a real-time mixed-trust computing framework that combines verification and protection. The framework introduces a new task model, where an application task can have both an untrusted and a trusted part. The untrusted part allows complex computations supported by a full OS with a real-time scheduler running in a VM hosted by a trusted hypervisor. The trusted part is executed by another scheduler within the hypervisor and is thus protected from the untrusted part. If the untrusted part fails to finish by a specific time, the trusted part is activated to preserve safety (e.g., prevent a crash) including its timing guarantees. This framework is the first allowing the use of untrusted components for CPS critical functions while preserving logical and timing guarantees, even in the presence of malicious attackers. We present the framework design and implementation along with the schedulability analysis and the coordination protocol between the trusted and untrusted parts. We also present our Raspberry Pi 3 implementation along with experiments showing the behavior of the system under failures of untrusted components, and a drone application to demonstrate its practicality.**

I. Introduction

Certification authorities (e.g., FAA [28]) allow the validation of different parts of a system with different degrees of rigor depending on their level of criticality. Formal methods have been recognized as important to verify safety-critical components [4]. Unfortunately, a verified property can be easily compromised if the verified components are not protected from unverified ones. Thus, **trust** requires that both verification and protection of components are jointly considered. This is the notion of trust used in this paper.

A key challenge to building trust is the complexity of today's operating systems (OSs) making them impractical to verify. Thus, there has been a trend to minimize the trusted computing base (TCB) by developing small verified hypervisors (HVs) and microkernels, e.g., seL4 [24], CertiKOS [21], and uberXMHF [32], [33]. 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 this approach *disjoint-trust computing*. Trusted components in the TCB are typically made small and simple due to the difficulty of verification. They are 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 in critical functionality whose safety must be assured through verification. This is because the verified components must be isolated from the untrusted ones if they are to be trusted. For instance, this prevents the use of untrusted machine learning algorithms (for which no effective verification method exists) to drive a car if such 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 practically impossible.

In this paper, we present the *real-time mixed-trust computing (RT-MTC)* framework. Unlike disjoint-trust computing, it gives the flexibility to use untrusted components even for CPS critical functionality. In this framework, untrusted components are monitored by verified components ensuring that the output of the untrusted components always lead to safe states (e.g., avoiding crashes). These monitoring components are known as *logical enforcers* [7], [16]. To ensure trust, these enforcers are protected by a verified micro-hypervisor [33]. To preserve the timing guarantees of the system, RT-MTC uses *temporal enforcers*, which are small, self-contained codeblocks that perform a default safety action (e.g., *hover* in a quadrotor) if the untrusted component has not produced a correct output by a specified time. Temporal enforcers are contained within the verified micro-hypervisor without jeopardizing the existing level of trust (e.g., using compositional verification offered by extensible micro-hypervisors [33]).

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[1] 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 the design and implementation of a coordination protocol between the two schedulers to preserve the synchronization between the trusted and untrusted components while preventing dependencies that can compromise the trusted component. Lastly, we present an implementation of our proposed framework using uberXMHF [33], an open-source, compositionally verified micro-hypervisor framework, and the ZSRM scheduler [17]. However, we note that in principle, our framework can also be instantiated with other verified micro-kernels or hypervisors provided they satisfy our

[1] Simplify HV logical verification by removing task interleavings [32], [33].

978-1-7281-3197-9/19/$31.00 ©2019 IEEE

requirements (see Section II).

This work relies on innovations for code verification for the trusted parts that were presented in previous publications. Since this is out of the scope of this paper, we refer the readers interested in the compositional verification and isolation provided by uberXMHF to [33], the runtime verification conducted by logical enforcers to [7], and the formal verification of temporal enforcement code to [10].

The remainder of this paper is organized as follows. Section II 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 III defines the system model. Section IV presents schedulability analysis of mixed-trust tasks, including the evaluation of the schedulability analysis. Section V presents a fail-safe coordination protocol. Section VI presents the implementation of mixed-trust scheduling. Section VII discusses the related work and Section VIII concludes.

II. Real-Time Mixed-Trust Computing (RT-MTC)

To discuss our RT-MTC framework we first summarize the logical model presented in [7]. Then, we extend it to accommodate the temporal enforcement and describe the architecture.

A. Logical Model

A system in [7] 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 $\alpha \in \Sigma$ be an action. The evolution of the system is modeled by the transition relation R_P, where P is the time between consecutive transitions, also known as *period.* Formally $R_P(\alpha) \subseteq \mathbf{S} \times \mathbf{S}$ is the relation such that if the action α is applied to the system in state s and it transitions to state s', then $(s, s') \in R_P(\alpha)$. Without loss of generality we also require that the system always performs an action in each period P to match the continuous evolution of physical processes. This can include an action where the source state is equal to the destination state (i.e., a null action). 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: $C_\phi \subseteq \phi$ and $\forall s \in C_\phi \bullet \exists \alpha \in \Sigma \bullet R_P(\alpha, s) \subseteq C_\phi$.

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 monitored and enforced by the logical enforcer. The logical enforcer, defined as $LE = (P, C_\phi, \mu)$, receives α from the untrusted component. Thus, we assume the logical enforcer executes with the same period P, and $\mu(s) \subseteq \mathit{SafeAct}(s)$ 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:

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

where $\mathrm{pick}()$ selects one element from the set with an arbitrary criteria. We say that $\tilde{\alpha}$ is an LE-enforced action.

We now extend this model by adding 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:

$$\hat{\alpha} = \begin{cases} \alpha_T & \mathit{if}\ \tilde{\alpha} = \bot \\ \tilde{\alpha} & \mathit{otherwise} \end{cases} \qquad (2)$$

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) $\bot$ denotes the absence of an action. Thus, we say that $\hat{\alpha}$ is a TE-enforced action. Finally, our system is assumed to start in C_ϕ.

B. Logical Model Required Conditions

We now define the conditions that our framework must enforce to prevent an untrusted component from causing behaviors not present in this model (see Appendix A in [18] for justification). For the discussion of these conditions, we let *output* denote the final action produced by the job once it has been evaluated by the LE and TE. These conditions are defined as follow:

- **C1.** Each task must produce an output every period.
- **C2.** There is only one output per period.
- **C3.** The output produced by a task in a period is either from LE or TE.
- **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.

C. 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 and issues an actuation. Within an iteration, the LE evaluates the computed actuation α and replaces it with a safe one ($\tilde{\alpha}$) if needed (as 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 issued by the **catch** block, effectively implementing (2). 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 which block performed the actuation, it is immediately followed by a wait for the completion of the current period before executing another iteration. 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 is from the LE — **C3**), (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 (**C2**), (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 — **C3**).

Algorithm 1: Behavior of a Mixed-Trust Periodic Task

```
while true do
    try:
        $s \leftarrow$ currentState()
        $\alpha \leftarrow$ computeActuation($s$)
        $\tilde{\alpha} \leftarrow LE(s, \alpha)$
        actuate($\tilde{\alpha}$)
    catch timeout(E):
        actuate($\alpha_T$)
    end
    waitForNextPeriod()
end
```

Based on the runtime assurance requirements and the failure possibilities presented above, we designed the software architecture presented in Fig. 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) modifications, 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 its arrival to decide whether or not to issue its safe action α_T.[2] 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 arrival time and the CPU bandwidth allocated in order to meet real-time deadlines.
- **P3.** Communication authentication. This means that we can verify the identity of the sender and ensure that the sender itself is protected (**P1**).

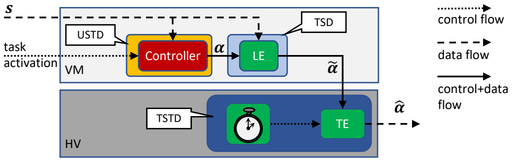

Fig. 1: Architecture

- **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 are a clear departure from other forms of protection between different tasks. Hence, we name this new task **mixed-trust task** whose timing characteristics will be formalized in Section III.

As discussed in Section I, 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 protection against memory modifications from untrusted components 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.

Untrusted Spatio-Temporal protection Domain (USTD). This is where the untrusted component resides. It offers untrusted spatial and temporal protection because it is implemented in the unverified VM.

The location of these domains within the architecture is shown in Fig. 1. 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 untrusted component.

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 IV. Hence, we defer the discussion of the coordination protocol to Section V. We now discuss the system model.

III. System Model

Our system model considers a uniprocessor system with a taskset $\Gamma = \{\mu_i | \mu_i = (T_i, D_i, \tau_i, \kappa_i)\}$ with unique priorities.

[2] TE can be activated by either the arrival of $\tilde{\alpha}$ from LE or the timeout E time units after the task arrival.

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: $\tau_i = (T_i, E_i, C_i)$, $\kappa_i = (T_i, D_i, \kappa C_i)$, 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 after its arrival. 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 by a timer set to expire E_i time units after τ_i arrives running 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 IV.

Under our mixed-trust scheduling paradigm, HTs are scheduled in a higher-priority band than GTs in the VM. 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.

Note that if a timing error is detected while a τ_i of a μ_i is running, 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 E_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.

IV. Schedulability Analysis

The schedulability analysis of a mixed-trust taskset is performed in three steps: (1) calculation of the worst-case response time (R_i^κ) of each HT κ_i assuming non-preemptive fixed-priority scheduling; (2) calculation of E_i for each GT τ_i by simply subtracting R_i^κ from the deadline D_i; and (3) calculation of the response time of each GT τ_i and check whether it is at most E_i.

A. 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) [15]. Specifically, the response time of a HT κ_i is calculated in three steps:

1) 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 executing job from a task with lower priority than κ_i's arriving before the beginning of the active period, and (ii) for the rest of the active period, there is only execution of jobs from tasks with priority higher than or equal to κ_i's. We then compute the maximum duration of a level-i active period, because: (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.
2) The start time of each job from κ_i in the level-i active period is calculated along with their finishing time. The finishing time of this job is calculated by just adding the execution time, since 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 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. Following [15], we calculate t_i^κ as the smallest solution:

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

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, (3) 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 $\kappa_{i,q}$ in the level-i active period. Then, (from [15]) we calculate $w_{i,q}^\kappa$ as the smallest solution of:

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

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 κ_i jobs to be as early as possible; this may change the schedule but neither the duration of the level-i active period nor the starting time of each κ_i job decreases. Hence, it holds that each $\kappa_{i,q}$ in the active period arrives $(q-1)T_i$ time units after the level-i active period starts. For each job of κ_i, we can add κC_i to its starting time and then subtract the arrival time of this job, which yields the response time of the job. Then we calculate the response time of κ_i as:

$$R_i^\kappa = \max_{q \in \{1 \ldots \left\lceil \frac{t_i^\kappa}{T_i} \right\rceil\}} (w_{i,q}^\kappa + \kappa C_i - (q-1)T_i). \quad (5)$$

Note 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 for tasksets without GTs.

In the rest of the paper, we use $E_i = D_i - R_i^\kappa$ (see Appendix C in [18] for further discussion).

B. Guest Task Response Time

To calculate the response times of GTs, we need a new notion of the busy period similar to the active period of the previous subsection but that also incorporates interference from HTs and higher-priority GTs. Therefore, we define the level-i *busy* period as a time interval such that at all times the processor is busy only with execution from jobs from HTs or execution from jobs of GT of priority greater than or equal to the priority of τ_i.

We now present a theorem that identifies the phasings that need to be explored to determine a GT's worst-case response time and motivates the schedulability equations.

Theorem IV.1. *The longest response time for all GT jobs of task τ_i of a mixed-trust task μ_i occurs in a level-i busy period initiated by the arrival of either τ_i or κ_i and the arrival of higher-priority GTs or HTs of other mixed trust tasks, μ_j.*

Proof. Following the argument of Lehoczky [25], let $[0, b]$ denote a level-i busy period (BP). Assume the BP is initiated by the arrival of higher-priority GTs or HTs of other mixed trust tasks, μ_j. Also assume that the first job of μ_i in the BP is a job of τ_i, and it arrives at some point, $x_i > 0$. Higher priority execution occupies $[0, x_i)$. This is followed by alternating intervals of τ_i and higher priority execution until τ_i finishes, ending the BP. Since τ_i execution cannot influence when higher priority jobs execute, reducing x_i to 0 leaves τ_i's completion time unaltered but moves its arrival time earlier thereby lengthening its response time. Additionally, if $E_i < b$, the arrival of κ_i will prevent τ_i from ever finishing, effectively resulting in an infinite response time and thus no need to check any further in the BP.

Again, assume the BP is initiated by the arrival of higher-priority GTs or HTs of other mixed-trust tasks, μ_j, but that the first job of μ_i in the busy period is a job of κ_i, and it arrives at some point, x_i, after 0. Once again we reduce x_i to 0 and observe the effect on the τ_i job that follows κ_i. The high priority and non-preemptability of κ_i might cause κ_i to preempt or delay some high priority execution in $[0, x_i)$. However, τ_i's completion time again will remain unaffected, while its response time is lengthened. One can see this by examining Fig. 2.b and moving the start time of κ_i from 0 to 2. We can also see the carry-in effect in Fig. 2.b. μ_j is delayed by $\kappa_{i,q-1}$ thereby delaying the execution of $\tau_{i,q}$. The finishing time of $\tau_{i,q}$ remains the same if $\kappa_{i,q-1}$ starts anywhere between 0 and 2.

Now assume that τ_i arrives at 0 and that for one μ_j, $j \neq i$ its first τ_j job, has a priority higher than τ_i and arrives after 0. All other higher-priority tasks, τ_k or κ_k, arrive at the start of the BP. The finish time, F_i, of τ_i is equal to C_i plus the execution time of higher priority jobs that execute before F_i. Moving the arrival of τ_j to 0 will either increase or leave unchanged the amount of higher priority execution before F_i thereby increasing F_i. If κ_i arrives at 0 moving the arrival of τ_j to 0 will also either increase or leave unchanged the finish time of all τ_i jobs in the BP. The same argument can be used if μ_j's first job is κ_j. □

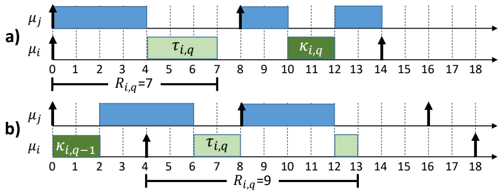

Fig. 2: Aligning τ_i and τ_j yields shorter $R_{i,q}$ than κ_i and τ_j

Previous work [9] has found the notion of a *request-bound function* useful in schedulability analysis. We will now discuss how to create a request-bound function for our model that takes additional parameters. Our request-bound function gives the amount of execution of a mixed-trust task.

Recall that in our model, a task μ_i can generate a τ_i job and a κ_i job E_i 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:

$$\text{rbf}_i^y(t, b) = \begin{cases} \left\lceil \frac{t-(T_i-E_i)}{T_i} \right\rceil^+ C_i b + \left\lceil \frac{t}{T_i} \right\rceil \kappa C_i & \text{if } y = E, \\ \left\lceil \frac{t}{T_i} \right\rceil C_i b + \left\lceil \frac{t-E_i}{T_i} \right\rceil^+ \kappa C_i & \text{if } y = A, \end{cases} \tag{6}$$

where $\lceil x \rceil^+ = \max(0, \lceil x \rceil)$.

We will use this request-bound function to compute the response time of the GT execution of a mixed-trust task μ_i. Then, if the computed response time for each GT is less than or equal to its E parameter, then the taskset is deemed 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 first present 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. 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 GT 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, similar to (3), we compute $t_i^{g,x}$ as the smallest solution of:

$$\begin{aligned} t_i^{g,x} = &\left(\sum_{j \in L_i} \mathrm{rbf}_j^E(t_i^{g,x}, 0)\right) + \mathrm{rbf}_i^x(t_i^{g,x}, 1) \\ &+ \sum_{j \in H_i} \max_{y \in \{E,A\}} \mathrm{rbf}_j^y(t_i^{g,x}, 1). \end{aligned} \tag{7}$$

where L_i and H_i contains the tasks with lower and higher priority (respectively) than τ_i. Given τ_i and 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, and $x \in \{E, A\}$, 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, similar to (4), we compute $w_{i,q}^{g,x}$ as the smallest solution of:

$$\begin{aligned} w_{i,q}^{g,x} = &\left(\sum_{j \in L_i} \mathrm{rbf}_j^E(w_{i,q}^{g,x}, 0)\right) + qC_i + (q - 1 + I_{(x=E)})\kappa C_i \\ &+ \sum_{j \in H_i} \max_{y \in \{E,A\}} \mathrm{rbf}_j^y(w_{i,q}^{g,x}, 1). \end{aligned} \tag{8}$$

In (8), I_ϕ is an indicator function that returns 1 if 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 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, similar to (5), we compute $R_{i,q}^{g,x}$ as:

$$R_{i,q}^{g,x} = w_{i,q}^{g,x} - ((q-1)T_i + I_{(x=E)}(T_i - E_i)). \tag{9}$$

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, similar to (5), we compute $R_i^{g,x}$ as:

$$R_i^{g,x} = \max_{q \in \left\{1 \ldots \left\lceil \frac{t_i^{g,x} - I_{x=E}(T_i - E_i)}{T_i} \right\rceil \right\}} R_{i,q}^{g,x}. \tag{10}$$

Finally, the response time of a GT is:

$$R_i^g = \max_{x \in \{E,A\}} R_i^{g,x}. \tag{11}$$

Note that a taskset is not required to have tasks with both GT and HT. 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. These conditions can also occur in a running system, e.g., 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 system as will be shown in Fig. 8 of Section VI-C.

C. Solving the equations

We consider three cases depending on utilization.

1) $\sum_{\mu_i \in \Gamma} \frac{C_i + kC_i}{T_i} > 1$
   For this case, we terminate the schedulability analysis and report *unschedulable* because for this case, there is no finite level-i busy period.
2) $\sum_{\mu_i \in \Gamma} \frac{C_i + kC_i}{T_i} < 1$
   For this case, (7) has a solution; thus, there is an upper bound on q. Note that (3),(4),(7), (8) are of the form z=f(z) and the right-hand side is monotonically non-decreasing in the variable on the left-hand side—we solve these 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 (7) has a solution—we pessimistically report *unschedulable*.

D. Budget 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 HTs will be protected from the failure. In contrast to the GTs, the HTs are trusted and their κC_is do 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*.

E. Experiments

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

1) When considering a task, its HT can experience interference from HT of other tasks. Also, the HT of other tasks can also delay the execution of the GT of the task under consideration. This double-accounting effect has no analog in classic fixed-priority scheduling.
2) A GT can experience a long delay because of the execution of HTs of all the other tasks. This situation 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).

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
Tmin		1000

TABLE I: Parameter Ranges and Defaults

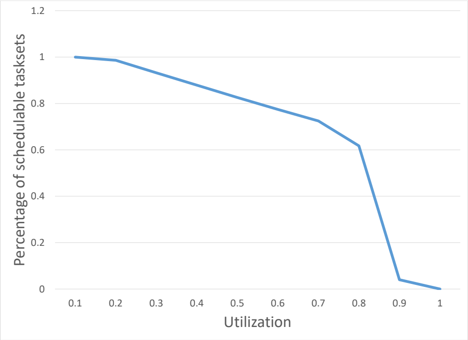

Fig. 3: Success rate as utilization grows

We now illustrate the schedulability conditions 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, and (4) the ratio between the HT WCET and the sum of the GT and HT WCET. See Table I. We perform four experiments to vary utilization, $\frac{Tmax}{Tmin}$ ratio, number of tasks, and $\frac{\kappa C}{C+\kappa C}$ 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 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. 5. Second, the default utilization is set to 80% also to reduce the influence of the utilization to dominate when varying the other parameters. In the experiments each data point 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. 3 shows the percentage of schedulable tasksets as the taskset utilization grows from 10% to 100%.

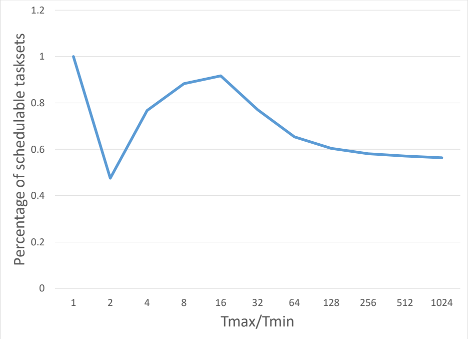

Fig. 4: Success rate as $\frac{Tmax}{Tmin}$ grows

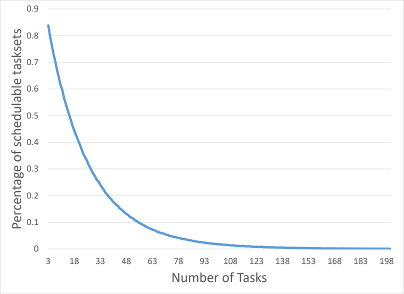

Fig. 5: Success rate as number of tasks grow

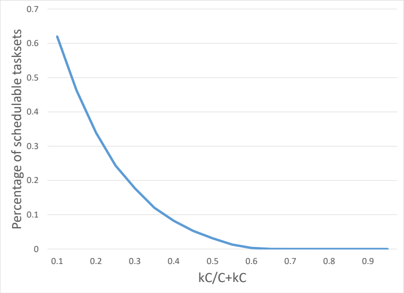

Fig. 6: Success rate as $\frac{\kappa C}{C+\kappa C}$ grows

Note that the experiment shows a decline in the percentage of schedulable taskset just after 20%. This is due to reason 2. Fig. 4 depicts the percentage of schedulable tasksets as the ratio of the maximum and minimum 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. Fig. 5 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 increases, the ratio of the maximum period to by the minimum period among the tasks generated becomes larger and then reason 2 becomes more impactful. Fig. 6 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.

V. Fail-Safe Mixed-Trust Scheduling Coordination Protocol

With the timing analysis as 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 [19]. 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*).

A. 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 necessary to properly implement the trusted temporal protection of the **TSTD** and the protection requirement **P2** (discussed in Section II). We leverage the secure boot mechanism provided by uberXMHF [31] 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 arrival 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 arrivals of the GTs and HTs as required.

B. 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 arrival of the HTs. This is the activation side of the protection requirement **P2**. To implement the FSHTrigger, the 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 [31] to isolate the HV and guest timers so that the guest cannot access the HV timer.

An HT must be isolated from failures of the corresponding GT, however, 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 the GT completed before E_i. The creation time synchronization is performed by SHTBoot. For the completion-time synchronization we rely on the logical enforcer (LE) in the VM to send the completion signal to the HV. More specifically, the 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 assured (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 suspended 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 hosts the TE preserves its temporal behavior (**P2**).

C. Late-Output Prevention (LOP) Protocol

A late output may occur when a GT job is allowed to complete and generate an output (e.g., actuation commands) E_i time units or more after its arrival. Recall the deferral GT enforcement approach explained in Section IV-D. 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 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 logical correctness from the temporal correctness, simplifying its verification but preserving both properties.

VI. Implementation

Our scheduling mechanisms are implemented by the combination of the budget enforcement of the ZSRM kernel scheduler [17] running in a VM and a non-preemptive fixed-priority scheduler implemented within the verifiable and extensible uberXMHF micro-HV [32], [33], [31] running in a Raspberry Pi-3 (RPi) platform with only one active core matching our system model. 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 protection have been formally verified as presented in [32], [33]. 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.

A. Hyper-Task-Aware Budget Enforcement

The ZSRM budget enforcement is performed by shadowing the priority queues of the fixed-priority scheduler within a

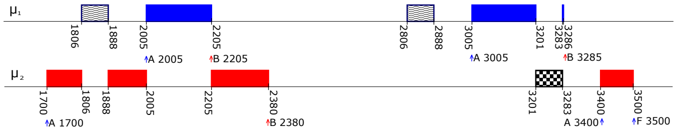

Fig. 7: Mixed-Trust Taskset Execution Timeline (in 10^{-4} secs)

kernel module 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 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.

Given that HT preemptions are invisible to the VM and the 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 used to discount the HTs preemptions from the budget when the budget timer triggers, reprogramming the timer accordingly. See Appendix B in [18] for overhead measurements.

B. Spurious Temporal Failure Illustration

We plot the run of a two-task taskset with timestamps captured from both the kernel and the HV schedulers for the case when a GT try to execute beyond C_i. Both schedulers read the same hardware timer counter register (as timestamps), allowing us to have an integrated timeline without incurring context-switch penalties. Fig. 7 shows the timeline plot reconstructed from timestamps for the following events: (i) **arrival (A)** marking when the job becomes ready to execute; (ii) **guest job finishing (F)** by calling the `wait_next_period()` of the VM scheduler when the job ends normally; (iii) **budget enforcement (B)**; (iv) **resume** marking the start of a colored rectangle showing when the job starts to execute; and (v) **paused** presented as the end of a colored rectangle. 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.

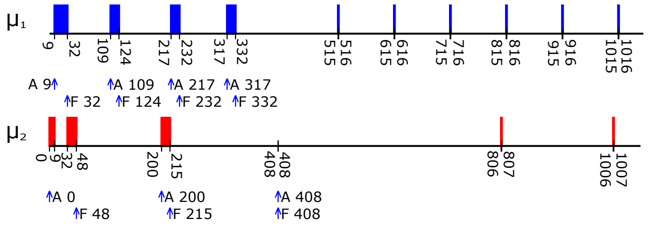

Fig. 8: Permanent VM Crash Experiment (in 10^{-2} secs.)

C. 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) with periods $T_1 = 1$ 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 finish given that the third task has the lowest priority). This call, in turn, calls the kernel `panic()` function designed to stop the kernel in unrecoverable failures (simulating a 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 Fig. 8. A few observations about the trace are in order. First, the call to `panic()` occurs 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.

D. Illustrative Application

We implemented a sample mixed-trust application of a drone mission. This application consists of two components:

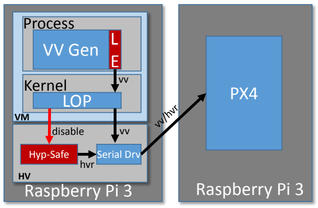

Fig. 9: Drone Application Architecture

(i) the mission controller, which generates velocity vectors (VV) that the drone must fly to follow a route, and (ii) the Pixhawk [1] flight controller running the PX4 autopilot [2] 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 [11]). In addition, we added an HT to the mission controller to take a safe action and continue to send VV messages to the flight controller in case the GT fails. Fig. 9 depicts this structure, which, for simplicity of presentation, shows only one mixed-trust 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 [3] 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 spurious failures and hard failures where we verified that both the LOP and the HTs properly prevent drone failures.

VII. Related work

Previous work recognized that small operating systems kernels can be more reliable [23] and can be formally verified [24]. In this context, decomposing an application can provide security benefits as well [30]. But they do not provide schedulability analysis. Previous work on hierarchical scheduling (e.g., [20]) 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 [26], [13], [8], [22], [34] 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 [27], [12] 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 scheduling (see [14] for an excellent survey) share our goal of monitoring run-time behavior and taking action when behaviors that are abnormal are detected. We are not aware of any work on a mixed-criticality scheduler that considers our task model and uses a formally verified HV.

Simplex [29] is an architecture comprising a complex controller, a simple controller, and two sets of states. The first set describes 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. Other frameworks (e.g., [5], [6]) mitigate the impact of attackers by rebooting, assuming that attacks do not happen instantaneously, but do not protect against bugs in unverified code.

VIII. Conclusions

The safe use of untrusted components in CPS critical functions requires protection and verification; this needs to guarantee logical and timing correctness. We presented the first framework that satisfies these requirements—we call our framework **real-time mixed-trust computing** (RT-MTC). The framework achieves this by (i) using trusted components to monitor and replace unsafe untrusted component outputs with safe ones (we call these monitoring components *logical enforcers*) and (ii) protecting the logical and temporal behavior of trusted components. Enforcers are protected from logical behavioral modification by preventing modifications to their memory (by untrusted components). However, to protect them from temporal behavior modifications it is necessary for an enforcer not to rely on output from untrusted ones in order to execute. Hence, in our framework we introduced a *temporal enforcer* that produces a safe output if the guarded untrusted component does not produce one by a pre-specified time. The untrusted component and its logical enforcer run in a *guest task* (GT) in a VM that runs on a trusted HV and the temporal enforcer runs in a *hyper task* (HT) within the HV. Together they form a *mixed-trust task*. A protocol was designed to coordinate the execution of a GT and its corresponding HT without forcing HT to depend on its GT. We also presented a new schedulability analysis for the mixed-trust task model and experiments to evaluate its performance. We showed the practicality and utility of our framework by (i) implementing it with the open source uberXMHF HV and ZSRM scheduler, (ii) demonstrating its ability to preserve the logical and timing correctness even in the presence of transient and permanent failures in the VM, and (iii) modifying the open-source drone-controller PX4 to insert enforcers that guarantee safety properties, testing it under both transient and permanent faults.

Acknowledgment

Copyright 2019 Carnegie Mellon University, Hyoseung Kim and John 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. The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation. 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. Internal use:* Permission to reproduce this material and to prepare derivative works from this material for internal use is granted, provided the copyright and No Warranty statements are included with all reproductions and derivative works. External use:* This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other external and/or commercial use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu. * These restrictions do not apply to U.S. government entities. Carnegie Mellon® is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University. DM19-0389

References

[1] https://pixhawk.org/.

[2] http://px4.io/.

[3] https://www.dronecode.org/.

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

[5] Fardin Abdi, Chien-Ying Chen, Monowar Hasan, Songran Liu, Sibin Mohan, and Marco Caccamo. Guaranteed physical security with restart-based design for cyber-physical systems. In *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*, ICCPS '18, pages 10–21, Piscataway, NJ, USA, 2018. IEEE Press.

[6] Fardin Abdi, Rohan Tabish, Matthias Rungger, Majid Zamani, and Marco Caccamo. Application and system-level software fault tolerance through full system restarts. In *Proceedings of the 8th International Conference on Cyber-Physical Systems*, ICCPS '17, pages 197–206, New York, NY, USA, 2017. ACM.

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

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

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

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

[11] 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.

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

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

[14] 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*, 2019.

[15] 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.

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

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

[18] Dionisio de Niz, Bjorn Andersson, Mark Klein, John Lehoczky, Amit Vasudevan, Hyoseung Kim, and Gabriel Moreno. Mixed-Trust Computing for Real-Time System — extended version. https://www.andrew.cmu.edu/user/dionisio/xchange/mixed-trust-scheduling-tr.pdf, 2019.

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

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

[21] 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.

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

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

[24] 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.

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

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

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

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

[29] L. Sha. Using simplicity to control complexity. *IEEE Software*, 2001.

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

[31] 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.

[32] 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.

[33] 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.

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