symmetry

MDPI

Article

Scheduling Randomization Protocol to Improve Schedule Entropy for Multiprocessor Real-Time Systems

Hyeongboo Baek [1] **and Chang Mook Kang** [2,*]

[1] Department of Computer Science and Engineering, Incheon National University (INU), Incheon 22012, Korea; hbbaek@inu.ac.kr
[2] Department of Electrical Engineering, Incheon National University (INU), Incheon 22012, Korea
* Correspondence: mook@inu.ac.kr; Tel.: +82-32-835-8437

Received: 08 March 2020; Accepted: 14 April 2020; Published: 6 May 2020

Abstract: Because most tasks on real-time systems are conducted periodically, its execution pattern is highly predictable. While such a property of real-time systems allows developing the strong schedulability analysis tools providing high analytical capability, it also leads that security attackers could analyze the predictable execution patterns of real-time systems and use them as attack surfaces. Among the few approaches to foil such a timing-inference security attack, TaskShuffler as a schedule randomization protocol received considerable attention owing to its simplicity and applicability. However, the existing TaskShuffler is only applicable to uniprocessor platforms, where the task execution pattern is quite simple to analyze when compared to multiprocessor platforms. In this study, we propose a new schedule randomization protocol for real-time systems on symmetry multiprocessor platforms where all processors are composed of the same architecture, which extends the existing TaskShuffler initially designed for uniprocessor platforms.

Keywords: real-time systems, security, schedule randomization protocol, multiprocessor platforms

1. Introduction

The primary concerns in designing safe critical real-time systems are to develop a methodology to effectively allocate limited computing resources (e.g., memory and CPU (central processing unit)) to multiple real-time tasks (e.g., motor control and sensing), and to derive a mathematical analysis mechanism. This mechanism would ensure every operation conducted by real-time tasks are completed within predefined time units (called deadlines) to satisfy real-time requirements [1]. The former and latter, which are respectively referred to as real-time scheduling algorithms and schedulability analysis, have been extensively studied over the past several decades in the field of real-time systems [2–4].

Unlike conventional research, which mostly focuses on the timing guarantees under various computing environments with different operational constraints, recent studies have focused on the security aspect because the modern real-time systems are exposed to unknown security attacks. For instance, modern real-time systems are increasingly being connected to unsecured networks such as the Internet, which allows the sophisticated adversaries to launch security attacks on UAVs (unmanned aerial vehicles) [5], industrial control systems [6], and automobiles [7,8].

Predictability, which is a key property of real-time systems, facilitates the development of several effective schedulability analysis mechanisms, but causes an increase in the success ratio of security attacks because of easy timing inferences [9–11]. Because most of the real-time tasks are conducted periodically, the execution patterns can be highly predictable, and most of the strong schedulability analysis mechanisms such as deadline analysis (DA) [12] and response-time analysis (RTA) [13] exploit such a property to judge whether each real-time operation can be completed within the deadline on the target environment. Such a seemingly advantageous property plays a double-edged sword

because the security attackers can analyze the predictable execution patterns of real-time systems and use them as attack surfaces. For example, recent studies have shown that an adversary can launch cache-based side-channel attacks by collecting information about important tasks or set up new covert channels [9,14]. The success ratio of such attacks is quite high because the periodic execution of real-time tasks results in repeated (based on the hyper-period of all real-time tasks on the system) scheduling patterns. After observing the real-time task schedules on the target systems, the adversary can predict future schedules of the system.

The main challenge in preventing such timing inference-based security attacks on real-time systems is making the initially predictable schedule unpredictable, while simultaneously satisfying the real-time constraint. Among the few proposed mechanisms that have been used to address the problem of the uniprocessor system, TaskShuffler (a schedule randomization protocol) has received considerable attention owing to its simplicity and applicability [15]. TaskShuffler exploits the notion of priority inversion budget, defined as the time interval between the finishing time of worst-case operation and deadline, of each task. Priority inversion budget value of each task implies that the task can complete its execution even if the other tasks execute for the amount of worst-case inversion budget instead of the task. Utilizing the calculated priority inversion budget value of each task, the TaskShuffler effectively selects random tasks at each schedule point and dynamically manages the value to induce uncertainty and satisfy the real-time constraint simultaneously.

As multiprocessor platforms have been increasingly adopted modern real-time systems to conduct highly resource-consuming tasks, the state-of-the-art techniques need to be tailored for multiprocessor platforms. For example, the latest embedded platforms developed for autonomous driving car consist of multiple CPUs to execute heavy-load tasks such as multiple sensing and image processing (e.g., NVIDIA Drive AGX Pegasus with 8-core “Carmel" CPUs based on ARM architecture) [16]. However, the existing TaskShuffler is only applicable to uniprocessor platforms, where the task execution pattern is quite simple to analyze when compared to multiprocessor platforms.

In this study, we propose a new schedule randomization protocol for real-time systems on symmetry multiprocessor platforms where all processors are composed of the same architecture, which extends the existing TaskShuffler initially designed for uniprocessor platforms. To develop a schedule randomization protocol for multiprocessor platforms, we need to address the following issues:

(i) How to define the problem of improving the security (i.e., uncertainty of schedules) of real-time systems and satisfying the schedulability simultaneously on multiprocessor platforms, differentiating it from the uniprocessor case (Section 2),
(ii) How to calculate the priority inversion budget value for each task on multiprocessors (Section 4.1),
(iii) How to utilize the calculated priority inversion budget values in randomized schedules effectively, to improve uncertainty (Section 4.2), and
(iv) How to satisfy the real-time requirement after applying the proposed schedule randomization protocol (Section 4.2).

To address point (i), we first recapitulate the underlying idea and purpose of the existing TaskShuffler designed for the uniprocessor case. Then, we define the problem for the multiprocessor case, which is addressed in this study. To address point (ii), we investigate each task’s surplus computing power allowed for lower-priority tasks without missing its corresponding deadline. To address point (iii), such lower-bound computing power is effectively utilized by the new schedule randomization protocol proposed in this study. In addition, we demonstrate that if the task set is schedulable with the fixed-priority (FP) preemptive scheduling, then it is schedulable with the proposed schedule randomization protocol; this addresses point (iv). Using experimental simulations, we then discuss various factors affecting the uncertainty of new schedules.

2. Problem Definition

TaskShuffler assumes that the attacker knows task sets’ parameters as well as scheduling policy of the target system [15]. The attacker aims at gleaning sensitive data such as the victim task’s private

key in shared resources such as DRAM (dynamic random access memory). Figure 1 briefly presents a scenario where the attacker launches a cache side-channel attack exploiting the scheduling pattern of real-time systems. The attacker first hijacks a task $task_A$ consecutively executing with a victim task $task_V$; it assumes that $task_A$ is relatively easy to hijack compared to $task_V$ because $task_A$ is not related to the security operation. Then, the attacker fills all cache sets with $task_A$'s data before $task_V$ executes. Thereafter, $task_V$ operates a cryptographic operation with a private key. Here, all cache sets were already filled with $task_A$' data, and thus some cache sets are replaced by $task_V$' data. Later, $task_A$ reads cache sets and measures the latencies. Some cache sets used by $task_V$ result in slow latencies because of cache misses for $task_A$. The attacker collects such timing information and reasons the location of the private key in the shared memory.

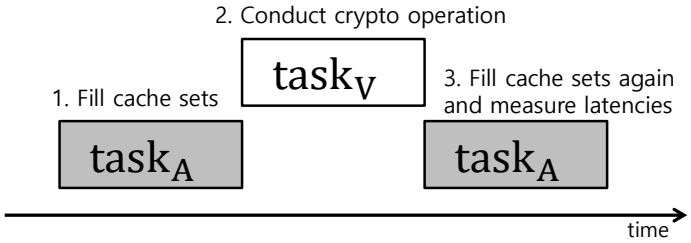


Figure 1. Cache side-channel attack scenario.

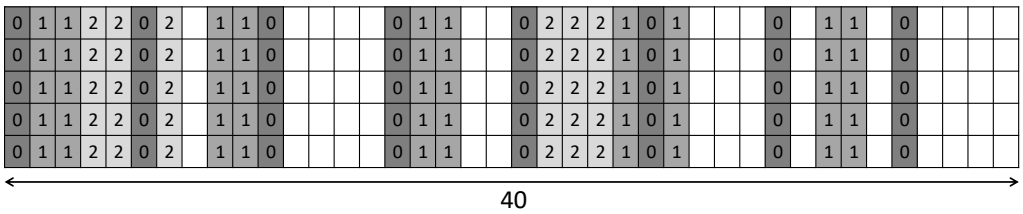
To make such an attack scenario feasible, the attacker should monitor the execution pattern of the target system for a long time to catch the proper timing to launch the attack. Because the same execution pattern of real-time systems is (mostly) repeated, the success ratio of the attack would increase. Figure 2 presents the scheduling pattern of $\tau = \{\tau_0(T_0 = 5, C_0 = 1, D_0 = 5), \tau_1(8, 2, 8), \tau_2(20, 3, 20)\}$ scheduled by a fixed-priority scheduling (τ_0 and τ_2 have the highest and lowest priorities, respectively) without (Figure 2a) and with (Figure 2b) TaskShuffler on a uniprocessor platform. It shows which task executes in 200 time slots, and the number in each rectangle presents the task index executed at the time slot. As shown in Figure 2a, the scheduling pattern is repeated every 40 time units because the least common multiple of periods of tasks in τ is 40. On the other hand, the scheduling pattern is obfuscated by TaskShuffler (most importantly) without schedulability loss (Figure 2b). That is, every task both in Figure 2a,b completes its execution without any deadline miss. It implies that TaskShuffler could improve potential durability against timing-inference attacks by obfuscating the scheduling pattern of real-time systems without schedulability loss.

In this study, we aim to develop a new schedule randomization protocol for multiprocessor platforms by extending the existing TaskShuffler initially designed for uniprocessor platforms. To achieve this, we first recapitulate the underlying idea and purpose of the existing TaskShuffler designed for the uniprocessor case. Then, we define the problem for the multiprocessor case, which is addressed in this study.

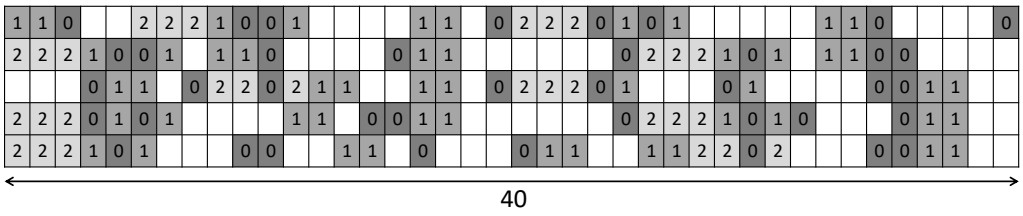
The key property of real-time systems is that most of the tasks operate repetitively at periodic intervals. This indicates that the same scheduling pattern for a certain period of time (e.g., one hyper-period of all tasks) can be exhibited in the next period. Although such property aids in developing better-performing analysis techniques that can easily judge the schedulability of real-time systems, an adversary can launch timing-inference attacks by collecting information about important tasks or set up new covert channels.

The TaskShuffler compensates for such shortcoming of real-time systems by reducing the predictability of schedules for real-time tasks. Therefore, even if an observer can record the exact schedule for a certain time period, the same will not be exhibited in the next period under the TaskShuffler. The underlying idea of the existing TaskShuffler is to pick up a random task from the ready queue, unlike most of the real-time systems where the highest priority task is selected for scheduling. Such counterintuitive mechanisms lead to the priority inversion problem and deadline misses, placing system safety at risk. To solve this problem, the TaskShuffler only allows limited

priority inversion for each task. That is, it restricts the use of priority inversion so that every task meets the original real-time constraint (i.e., meeting deadlines). To this end, it calculates the lower-bound amount of priority inversion that each task can tolerate, using the TaskShuffler protocol. When the priority inversion limit is reached during execution, the lower-priority task stops running, and the highest priority task is selected to be scheduled.



(a) Fixed-priority scheduling



(b) Fixed-priority scheduling with TaskShuffler

Figure 2. Scheduling pattern of $\tau = \{\tau_0(T_0 = 5, C_0 = 1, D_0 = 5), \tau_1(8, 2, 8), \tau_2(20, 3, 20)\}$ scheduled by fixed-priority scheduling without and with TaskShuffler on a uniprocessor platform.

Then, the TaskShuffler achieves the following goal G.

G. Suppose that a task set τ is scheduled by a given FP scheduling algorithm S, and its schedulability is guaranteed by a given schedulability analysis A. Then, τ's schedulability is still guaranteed by a given schedulability analysis A' when it is scheduled by the FP algorithm S' incorporating the TaskShuffler protocol of uniprocessors.

Let $\lambda(S)$ be the set of schedulable task sets scheduled by FP scheduling, and $\lambda(A)$ be the set of task sets each of whose schedulability is guaranteed by a given schedulability analysis supporting FP scheduling. In addition, $\lambda(S)'$ denotes the set of schedulable task sets scheduled by FP scheduling incorporating TaskShuffler, and $\lambda(A')$ is the set of task sets each of whose schedulability is guaranteed by a given schedulability analysis supporting FP scheduling incorporating TaskShuffler. Figure 3a shows the regions of task sets covered by $\lambda(S)$, $\lambda(A)$, $\lambda(S')$, and $\lambda(A')$ for uniprocessor platforms. $\lambda(S) = \lambda(A)$ in Figure 3a indicates a well-known fact that every schedulable task set $\tau \in \lambda(S)$'s schedulability is guaranteed by the given exact schedulability analysis (e.g., RTA supporting FP scheduling) [13]. Then, the TaskShuffler targets task sets belonging to $\lambda(A)$ $(= \lambda(S))$ and applies the TaskShuffler protocol; note that task sets out of $\lambda(A)$ has nothing to do with the goal G. It utilizes a specialized schedulability analysis A' to support the FP scheduling S' incorporating the TaskShuffler protocol. When A' is conducted for each task $\tau_i \in \tau$, the lower-bound amount of priority inversion for each task is calculated, and the TaskShuffler exploits it without compromising the schedulability of task sets $\tau \in \lambda(A)$. Therefore, it achieves $\lambda(S) = \lambda(A) = \lambda(A') = \lambda(S')$ as shown in Figure 3a.

For multiprocessor platforms, there is no exact schedulability analysis for our system model [17], which provides $\lambda(S) \neq \lambda(A)$ as shown in Figure 3b. Therefore, we target $(\lambda(A) \subsetneq \lambda(S))$ in which tasks' schedulabiliy are guaranteed by the DA schedulability analysis and aim at achieving the goal G on multiprocessors.

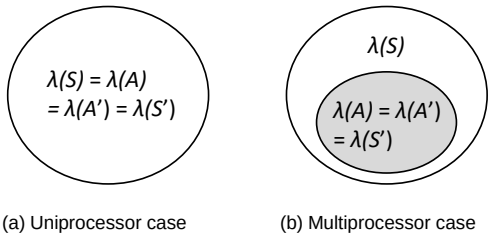


Figure 3. The regions covered by each technique.

3. System Model

We consider a task set τ the Liu and Layland task (and system) model (considered as the de-fecto standard in the field of real-time scheduling theory) in which every task $\tau_k \in \tau$ is scheduled by global, preemptive and work-conserving FP real-time scheduling algorithm on m identical multiprocessors [1]. A scheduling algorithm is called global, preemptive and work-conserving if a task migrates from one processor to another, a lower-priority task is preempted by a higher-priority one, and the processor is never idle when there are jobs to be executed, respectively. In FP scheduling, the priority is assigned to each task such that all jobs invoked by the same task have the same priority. The Liu and Layland model assumes that tasks are independent, no synchronization is needed, and resources (except processors or cores) are always available. According to the task model, we assume that all processors share the common cache and main memory. A task τ_k periodically invokes a job J_k in every T_k time units, and each job is supposed to execute for at most C_k time units (also known as the worst-case execution time (WCET)) to declare its completion. Every job invoked by τ_k should complete its execution in D_k time units as a real-time constraint. j-th job invoked by a task τ_k is denoted by J_k^j, and it is invoked at the release time r_k^j and should finish its execution within the absolute deadline $d_k^j = r_k^j + D_k$. We use the notation J_k when it indicates an arbitrary job of a task τ_k. A job J_k is said to be schedulable when it finishes its execution before its absolute deadline d_k^j. Further, a task τ_k is said to be schedulable when all jobs invoked by τ_k are schedulable, and a task set τ is said to be schedulable when all tasks $\tau_k \in \tau$ are schedulable. $lp(\tau_k)$ and $hp(\tau_k)$ represent the set of tasks whose priorities are lower than that of τ_k and that whose priorities are higher than that of τ_k, respectively. We consider *online* scheduling algorithms and *offline* schedulability analysis (in online scheduling algorithms and offline schedulability analysis, the priorities of pending or executing jobs (i.e., scheduled by the given scheduling algorithm) are determined after the system starts, while the schedulability of the given task set is judged before the system starts.)

4. Schedule Randomization Protocol for Multiprocessors

The key idea of our schedule randomization protocol is to select random jobs from the ready queue rather than the originally prioritized ones that are supposed to be selected by a given algorithm. This operation inevitably causes priority inversions for some tasks, which can induce an additional execution delay when compared to the existing schedule. This can result in a deadline miss in the worst case, even if the task is deemed schedulable by a given schedulability analysis. To avoid such situations, we should employ bounded priority inversions (calculated by the schedule randomization protocol) so that every task completes its execution before its corresponding deadline.

In this section, we first calculate the upper bound of allowed priority inversions for each task (in the first subsection), and then present how to effectively utilize such calculated priority inversions budget values in a schedule randomization protocol for multiprocessors.

4.1. Priority Inversion Budget Calculation

As an offline step of our schedule randomization protocol for multiprocessors, the maximum number of time units that are allowed for jobs of tasks in $lp(\tau_k)$ to execute when a job of the task τ_k

waits for another job to complete should be calculated (we use a subscript 'k' if the notation is related to a task whose schedulability will be judged. We use a subscript 'i' if the notation is related to higher priority tasks of τ_k.). We refer such time units as the priority inversion budget, V_k, which will be utilized in the online step of randomization protocol. We define the worst-case inversion budget as follows.

Definition 1. *(Priority inversion budget V_k) The priority inversion budget V_k of a task τ_k is defined as the maximum amount of time units in $[r_k, d_k)$ that is allowed for jobs of tasks in $lp(\tau_k)$ to execute while a job J_k of a task τ_k waits on multiprocessors, which ensures schedulability of τ_k.*

Thus, V_k can be lower-bound by calculating the allowable limit for delay in execution of J_k caused by lower-priority jobs in $[r_k, d_k)$ without missing deadlines. To achieve this, we exploit the underlying mechanism of a well-known schedulability analysis called DA, which uses two notions of interference defined as follows.

Definition 2. *(Worst-case interference I_k on τ_k) The worst-case interference I_k on a task τ_k in an interval $[r_k, d_k)$ on multiprocessors is defined as the maximum cumulative length of all the intervals in which J_k is ready to execute but cannot be scheduled because of higher-priority jobs.*

Definition 3. *(Worst-case interference $I_{k\leftarrow i}$ of τ_i on τ_k) The worst-case interference $I_{k\leftarrow i}$ of a task τ_i on a task τ_k in an interval $[r_k, d_k)$ on multiprocessors is defined as the maximum cumulative length of all the intervals in which J_k is ready to execute but cannot be executed on any processor when the job J_i of the task τ_i is executing.*

Using the definition of I_k, V_k is calculated as follows.

$$V_k = D_k - (C_k + I_k). \tag{1}$$

For a job J_k to interfere at a time unit in $[r_k, d_k)$, there are m jobs at the time unit. By the definition of $I_{k\leftarrow i}$, I_k is calculated as follows.

$$I_k = \frac{\sum_{\tau_i \in hp(\tau_k)} I_{k\leftarrow i}}{m}. \tag{2}$$

To upper bound $I_{k\leftarrow i}$, we use the concept of workload of a task τ_i in an interval of length ℓ, which is defined as the maximum amount of time units required for all jobs released from τ_i in the interval of length ℓ. As shown in Figure 4, the left-most job (called the carry-in job) of τ_i starts its execution at t (i.e., the beginning of the interval) and finishes it at $t + \ell$. That is, it executes for C_i without any delay. Thereafter, the following jobs are released and executed without any delay. By calculating the number of jobs executing for C_i and the other jobs executing for a part of C_i, the workload of τ_i in an interval of length ℓ (i.e., $I_{k\leftarrow i}$) is calculated by [18].

$$I_{k\leftarrow i} = \left\lceil \frac{D_k + D_i - C_i}{T_i} \right\rceil C_i + \min\left(C_i, D_k - \left\lceil \frac{D_k + D_i - C_i}{T_i} \right\rceil T_i\right). \tag{3}$$

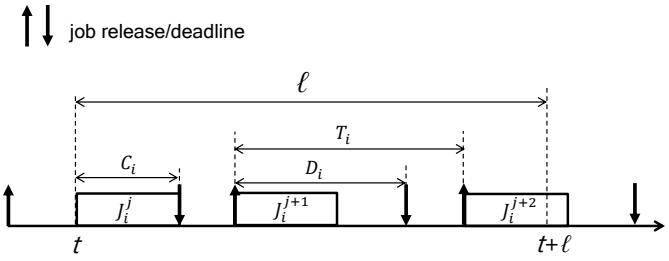


Figure 4. The worst-case scenario in which the workload of τ_i is maximized in an interval of length ℓ.

Figure 5 illustrates the underlying idea of DA with an example in which the target task τ_k's schedulability is judged considering its four higher priority tasks $\tau_i \in hp(\tau_k)$ on $m = 3$. In the interval $[r_k^j, d_k^j)$, there is only one job J_k^j of τ_k, and there is no deadline miss if J_k^j's execution is not hindered by more than $D_k - C_k + 1$. As seen in Figure 5, $I_{k \leftarrow i}$ can be larger than $D_k - C_k + 1$, and some portion of that inevitably executes in parallel with J_k^j since we assume that each job cannot execution in parallel on more than one processor. Thus, DA limits the amount of $I_{k \leftarrow i}$ to $D_k - C_k + 1$, which refines Equation (2) as follows.

$$I_k = \frac{\sum_{\tau_i \in hp(\tau_k)} \min\left(I_{k \leftarrow i}, D_k - C_k + 1\right)}{m}. \tag{4}$$

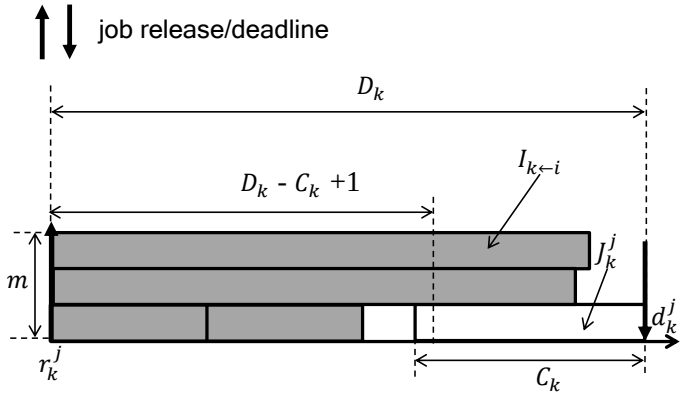


Figure 5. An example of deadline analysis (DA) for four higher-priority tasks τ_i and a target task τ_k

Theorem 1. *Suppose that $\tau_k \in \tau$ scheduled by a given fixed-priority scheduling algorithm is deemed schedulable by DA, and tasks in $lp(\tau_k)$ do not delay τ_k more than V_k. Then, τ_k is still schedulable with the randomization protocol.*

Proof. By the definition τ_k is schedulable if every job J_k^j released by τ_k at r_k^j can finish its execution within D_k. From Equation (1), we have

$$C_k + I_k + V_k = D_k. \tag{5}$$

The worst-case execution for J_k^j is upper bounded by C_k, and the time in which J_k^j is hindered by higher-priority jobs is upper bounded by I_k according to Equation (4). Therefore, if the tasks in $lp(\tau_k)$ do not delay τ_k for more than V_k, then every job J_k^j of τ_k can finish its execution within D_k. □

4.2. Schedule Randomization Protocol

Based on the mechanism to calculate the priority inversion budget of each task τ_i explained in the previous section, we illustrate how the new schedule randomization protocol for multiprocessors operates in this section. Let $Q_r = (J_{(1)}, J_{(2)}, \cdots, J_{(|Q_r|)})$ be the ready queue in which active jobs are sorted in decreasing order of priority. It implies that $J_{(1)}$ and $J_{(|Q_r|)}$ are the highest- and lowest-priority jobs in Q_r, respectively. We assume that TaskSuffler operates only when Q_r is greater than m because all active jobs will be selected for schedule, otherwise.

The TaskSuffler for multiprocessor conducts Algorithm 1 for $Q_r > m$ at every scheduling decision. It first adds $J_{(1)}$ to the candidate list, L_c (Line 1). If its remaining inversion budget $v_{(1)}$ is equal to zero, then it returns $J_{(1)}$ and $(m-1)$ highest-priority jobs in Q_r (Lines 2–4). Otherwise, with letting $J_{(i)}$ be the job in the current iteration, it iterates the following for $J_{(2)}$ through $J_{(|Q_r|)}$ (Lines 5–11). If the remaining

inversion budget $v_{(i)}$ of $J_{(i)}$ is larger than zero, it adds $J_{(i)}$ to the candidate list L_c and considers the next job (Lines 6–7). Otherwise, it adds $J_{(i)}$ to the candidate list L_c and stops the iteration (Lines 8–9). Then, if $|L_c|$ is smaller than or equal to m, it returns all jobs in L_c and $(m - |L_c|)$ highest-priority jobs in Q_r. Otherwise, it returns randomly selected m jobs in $|L_c|$. After Algorithm 1 is conducted, the selected m jobs execute until the next scheduling decision.

Algorithm 1 TaskSuffler for multiprocessors.

1: $L_c \leftarrow J_{(1)}$
2: **if** $v_{(1)} \leq 0$ **then**
3: **return** $J_{(1)}$ and $(m-1)$ highest-priority jobs in Q_r
4: **end if**
5: **for** each $J_{(i)} \in Q_r$ for i from 2 to $|Q_r|$ **do**
6: **if** $v_{(i)} > 0$ **then**
7: add $J_{(i)}$ to L_c
8: **else**
9: add $J_{(i)}$ to L_c and **goto** Step 12
10: **end if**
11: **end for**
12: **if** $|L_c| \leq m$ **then**
13: **return** all jobs in L_c and $(m - L_c)$ highest-priority jobs in Q_r
14: **else**
15: **return** randomly selected m jobs in L_c
16: **end if**

Let τ_b denote the task of the lowest-priority job among selected jobs, and τ^u denote the set of tasks that were not selected. The next schedule decision is made at

$$t' = t + min(v_i | \tau_i \in (hp(\tau_b) \cap \tau^u)), \tag{6}$$

unless a new job arrives or any of the selected jobs finishes its execution before t'. Thus, the remaining inversion budget of every released job belonging to a task $\tau_i \in (hp(\tau_b) \cap \tau^u)$ is deducted by one at each time unit until the next schedule decision will be made, a new job arrives, or any of the selected jobs finishes its execution before t'. The remaining priority inversion budget v_i of each job J_i is set to V_i when J_i is released.

Theorem 2. *Suppose that $\tau_k \in \tau$ scheduled by a given fixed-priority scheduling algorithm is deemed schedulable by DA schedulability analysis, then it is still schedulable under the schedule randomization protocol.*

Proof. By Theorem 1, τ_k is schedulable if its execution is not hindered for more than V_k by lower-priority tasks. τ_k's execution is interfered by lower-priority tasks only when τ_k's priority is lower than that of τ_b, and the amount of interference from lower-priority tasks cannot be larger than V_k since the next scheduling decision is made before v_k becomes zero owing to Equation (6). □

To implement TaskShuffler to the existing scheduler, the system should be capable for tracking tasks' remaining execution times and priority inversion budgets. Such monitoring ability is already commonly available on many real-time systems, whose aim is to guarantee that they do not exceed their execution allowances [19,20]. Utilizing such the ability, TaskShuffler may impose additional scheduling decisions according to the policy of TaskShuffler, compared to the vanilla scheduling algorithms (e.g., rate monotonic). It naturally increases scheduling costs such as preemption (or context switching) and migration costs. The system designers should consider how much such scheduling costs will happen for their target systems when TaskShuffler is considered to improve the schedule entropy without schedulability loss.

4.3. Schedule Entropy for Multiprocessors

Because our goal is to improve the uncertainty in scheduling to foil the fixed schedule pattern of real-time scheduling, we need to evaluate the improvement in the uncertainty of our proposed schedule randomization protocol. To overcome this issue, we use the concept of schedule entropy initially designed for a uniprocessor [21–23]. The underlying idea of schedule entropy is to measure randomness (or unpredictability) of schedule at each time unit, called slot entropy, and derive the summation of all slot entropies over a *hyper-period* L (defined as the least common multiple of T_i of all tasks $\tau_i \in \tau$). The slot entropy $H_\tau(t)$ at a time slot t for a task set τ is calculated as follows.

$$H_\tau(t) = -\sum_{\tau_i \in \tau} Pr(\tau_i, t) log_2 Pr(\tau_i, t), \tag{7}$$

where $Pr(\tau_i, t)$ is the probability mass function of a task τ_i appearing at time t. $Pr(\tau_i, t)$ is obtained empirically by observing multiple hyper-periods [15]. Then, the schedule entropy H_τ is calculated by the summation of all slot entropies over a hyper-period L as follows.

$$H_\tau = \sum_{t=0}^{L-1} H_\tau(S_t). \tag{8}$$

According to the considered task model, we assume that all processors share the common cache and main memory. Then, it also implies that which processor is assigned to a task τ_i does not affect the level of scheduling entropy. That is, the level of scheduling entropy is influenced by whether τ_i is scheduled or not in a time slot. Such the assumption is the limitation of our study, and thus it should be considered as potential future work.

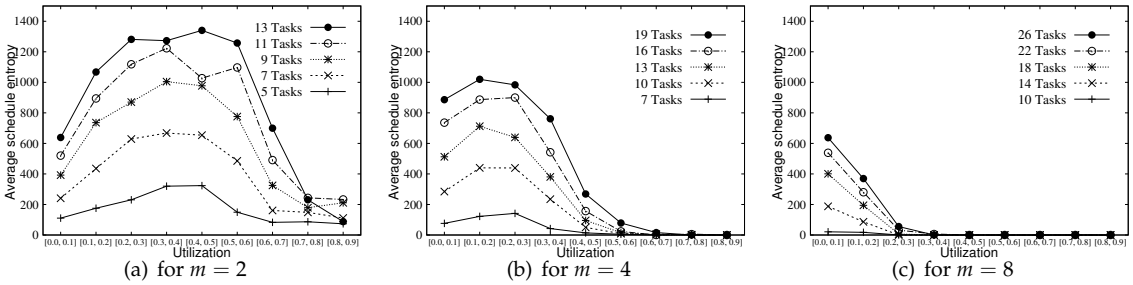
5. Evaluation

In this section, we evaluate the performance of our schedule randomization protocol with randomly generated synthetic task sets to understand the effect of our approach on various factors more fully.

We randomly generated even task sets from nine system utilization groups, $[0.01 + 0.1 \cdot i, 0.09 + 0.1 \cdot i]$ for $i = 0, \cdots, 8$, that is, 100 instances per group. The system utilization of a task set is defined as the sum of the utilizations of all tasks ($\sum_{\tau_i \in \tau}(C_i/T_i)$) in the task sets. We considered three different numbers of processors $m = 2, 4$ and 8. Each system utilization group has five sub-groups, each of which has a fixed number of tasks; for $m = 2, 4$ and 8, the number of tasks for five sub-groups are $\{5, 7, 9, 11, 13\}$, $\{6, 9, 12, 15, 18\}$ and $\{7, 11, 15, 19, 23\}$. Each task period T_i was randomly selected from {20, 40, 80, 160, 320, 640, 1280, 2560}, and each WCET C_i was selected from $min([1, 50], T_i)$. A total of 100 task sets were generated for each sub-group, and thus $7 \cdot 3 \cdot 5 \cdot 100 = 10{,}500$ task sets were generated in total. As our goal is to improve the uncertainty of the schedule without compromising schedulability, we only selected the task sets whose schedulability is guaranteed by DA schedulability. We used rate-monotonic (RM) scheduling as our base scheduling algorithm in which our proposed schedule randomization was applied. To obtain the converged schedule entropy of each task set, we conducted a simulation for 10,000 hyper-periods of the task set as it ensures less than 0.01% (recommended to obtain the converged schedule entropy [15]) of difference in schedule entropy between those from 9999 and 10,000 hyper-periods.

Figure 6 shows the plot of average schedule entropy of each sub-group's task sets (i.e., 100 task sets for each sub-group) over varying system utilization groups for $m = 2, 4$ and 8 (Due to limited space on x-axis, $[0.01 + 0.1 \cdot i, 0.09 + 0.1 \cdot i]$ for $i = 0, \cdots, 8$ is represented as [0.0, 0.1], [0.1, 0.2], $\cdots$, [0.7, 0.8], [0.8, 0.9].). As shown in Figure 6a, the schedule entropy of a task set is high on average when it consists of a large number of tasks (System utilization group [0.8, 0.9] exhibits an exceptional result because of heavy utilization of its task set, which result that most of the task sets exhibit zero schedule entropy). This is because a higher number of tasks with our schedule randomization protocol possibly

provides larger options for tasks to be randomly selected in every time slot. For example, the schedule entropy $H_\tau(t)$ at a time slot t of a task set τ having two tasks $\tau_i \in \tau$ each of whose $Pr(\tau_i, t)$ is 1/2 is 1, while $H_\tau(t)$ is 2 if τ has four tasks $\tau_i \in \tau$ each of whose $Pr(\tau_i, t)$ is 1/4. This implies that a larger number of tasks for each task set can improve schedule entropy. In addition, Figure 6a demonstrates that task sets included in the groups whose system utilization is low or high (e.g., [0.0, 0.1], [0.1, 0.2], [0.7, 0.8] or [0.8, 0.9]) result in relatively low average schedule entropy while the others (e.g., [0.2, 0.3]–[0.6, 0.7]) results in high average schedule entropy. Because the task sets with low system utilization have less WCET, no process operates (i.e., processors are idling) in most of the time slots. When it comes to high utilization, most of the tasks have low V_i values; therefore, the chances of tasks to be randomly selected are quite low. Therefore, a low schedule entropy results in both cases.



(a) for $m = 2$ (b) for $m = 4$ (c) for $m = 8$

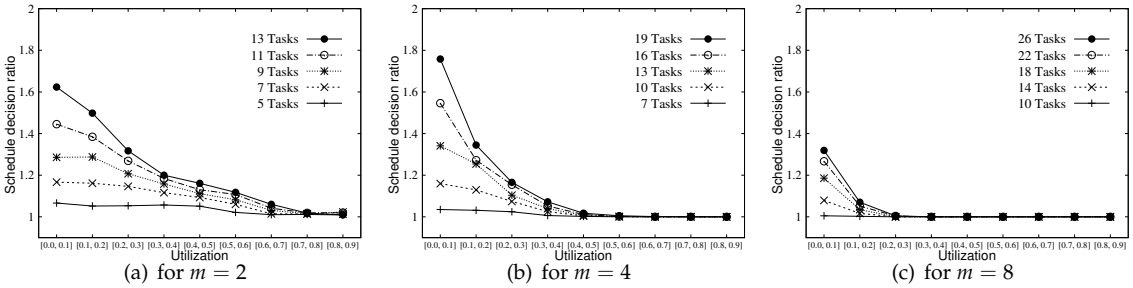
Figure 6. Average schedule entropy for $m = 2, 4$ and 8.

From Figure 6a–c, we can observe that the average schedule entropy of task sets decreases as the number of processors m increases. This is mainly due to the underlying pessimism of the DA schedulability analysis in calculating I_k in Equation (4). As Figure 5 implies, the DA schedulability analysis assumes that the execution corresponding to $I_{k \leftarrow i}$ is performed as much as possible in the interval $[r_k^j, r_k^j + D_k - C_k + 1)$ to upper bound I_k. However, this does not happen in most cases since two jobs released consecutively from the same task may execute at intervals from each other, which implies that the amount of execution contributing to I_k is overestimated under the DA schedulability analysis. Such pessimism of the DA schedulability analysis increases for a larger value of m. As our schedule randomization protocol is based on the DA schedulability analysis to derive V_i, task sets with a lower value of V_i (from a larger value of m) results in a lower average schedule entropy.

While Figure 6 presents the average schedule entropy over varying system utilization groups, Tables 1–3 show the maximum schedule entropy obtained from each setting. As the minimum schedule entropy of every setting is zero, Tables 1–3 also represent the range of schedule entropy that can be obtained from each setting. "-" in the tables represents a value lower than 0.1, and we exclude rows of the tables if all values corresponding to the row are "-". The trends shown in Figure 6a–c also appear in Tables 1–3, but the maximum schedule entropy of task sets in the high system utilization group are relatively high compared to the average schedule entropy of those task sets. This indicates that a very low number of task sets in the high system utilization group shows exceptionally high schedule entropy from a certain task parameter setting following the considered task set generation method, and the other task sets are with zero schedule entropies.

One may wonder whether much additional scheduling overhead is required to apply our schedule randomization protocol into the existing scheduling algorithm. Note that the schedule decision on preemptive scheduling algorithm (basically considered in our paper) without the schedule randomization protocol is made only when a job finishes its execution or a new job is released. In addition, an additional schedule decision stemming from the schedule randomization protocol is made when v_i (i.e., the remaining priority inversion budget value of V_i) of a job becomes zero. Figure 7 shows the ratio between the number of schedule decisions made by naive RM and that made by RM with the schedule randomization protocol. As shown in Figure 7, for task sets with high system utilization or for larger m shows less schedule decision ratio. With high system utilization or larger m,

each task has a lower priority inversion budget as aforementioned, and the schedule randomization protocol works rarely with such settings. Overall, a high schedule decision ratio implies a larger scheduling overhead, and thus the system designers should carefully consider the trade-off between scheduling overhead and the degree of security they want to achieve for their target system.



(a) for $m = 2$ (b) for $m = 4$ (c) for $m = 8$

Figure 7. Schedule decision ratio for $m = 2, 4$ and 8.

Table 1. m = 2.

	5 Tasks	7 Tasks	9 Tasks	11 Tasks	13 Tasks
[0.0, 0.1]	434.2	612.9	727.2	801.1	967.8
[0.1, 0.2]	597.0	853.8	1357.3	1575.6	1599.9
[0.2, 0.3]	930.4	1269.4	1811.5	1901.7	2166.2
[0.3, 0.4]	1166.0	2008.1	2082.4	2280.5	2784.5
[0.4, 0.5]	2049.6	2038.7	2152.4	2112.5	2659.2
[0.5, 0.6]	1626.3	1754.0	2221.2	2211.9	2674.5
[0.6, 0.7]	1659.7	1084.8	1583.9	2372.3	2529.1
[0.7, 0.8]	981.1	1978.5	1802.6	2280.7	1915.0
[0.8, 0.9]	1332.2	2229.9	2043.6	2183.4	1952.1

Table 2. m = 4.

	6 Tasks	9 Tasks	12 Tasks	15 Tasks	18 Tasks
[0.0, 0.1]	257.3	695.5	845.2	1147.1	1321.0
[0.1, 0.2]	377.6	977.1	1441.6	1556.8	1602.6
[0.2, 0.3]	669.6	1023.0	1584.3	2196.7	1712.7
[0.3, 0.4]	409.5	997.6	1416.2	1752.8	1875.8
[0.4, 0.5]	428.2	652.6	1567.5	1327.4	1462.2
[0.5, 0.6]	480.8	416.3	430.4	735.3	1184.2
[0.6, 0.7]	-	401.7	67.4	-	1208.2

Table 3. m = 8.

	7 Tasks	11 Tasks	15 Tasks	19 Tasks	23 Tasks
[0.0, 0.1]	144.9	583.9	861.4	878.6	1014.1
[0.1, 0.2]	126.8	357.6	789.5	811.4	1196.0
[0.2, 0.3]	45.3	-	343.3	470.8	728.3
[0.3, 0.4]	-	-	-	643.5	-

6. Related Work

The problem of information leakage for real-time systems has been addressed in several studies. Mohan et al. considered real-time tasks with different security levels and focused on information leakage on shared computing resources (e.g., RAM and cache). They proposed a mechanism for FP scheduling to flush the status of shared resources conditionally, which reduces the chances of an attacker from obtaining sensitive information of the resources [24]. Because they incorporated a security mechanism into the existing FP scheduling, an additional timing overhead was inevitably

required. Therefore, they proposed a new sufficient schedulability analysis to accommodate that fact. In [25], an exact schedulability was proposed to improve the analytical capability. In addition, this work was extended to mixed-criticality systems in [26]

The aforementioned studies addressed the issues only for non-preemptive scheduling; therefore, Pellizzoni et al. extended this work to preemptive scheduling [27]. They extended it to a more general task model, and proposed an optimal priority assignment method that determines the task preemptibility.

Another approach to improve the security of real-time systems is to randomize the schedules without compromising schedulability. TaskShuffler addressed in this study was proposed by Yoon et al. for a preemptive FP scheduling algorithm [15]. The goal of TaskShuffler is to improve uncertainty in schedule and reduce the success ratio of timing inference attacks simultaneously. Kr¨uger et al. proposed an online schedule randomization protocol for time-triggered systems [28]. While the aforementioned two approaches are applicable to uniprocessor platforms, we focus on multiprocessor platforms.

7. Conclusions

In this study, we aimed to develop a new scheduling randomization protocol for symmetry multiprocessors to improve the security and conserve schedulability of real-time systems simultaneously, by extending the existing TaskShuffler initially designed for uniprocessors. To this end, we first define the problem of improving the security of real-time systems and satisfying the schedulability simultaneously on multiprocessor platforms, differentiating it from the uniprocessor case. Then, we employed DA schedulability to derive priority inversion budget values for each task, and proposed an algorithm to effectively utilize the calculated priority inversion budget values in randomized schedules to improve uncertainty. Based on the simulation results, we investigated the effect of our approach on various factors. Non-preemptive [29] or partitioned [2] scheduling will be considered in our future work.

As our study adopts a fundamental task model (of the Liu and Lyland) assuming no consideration of task dependency, resource consideration and scheduling costs (e.g., migration or preemption cost), it cannot be directly applied to actual real-time systems without relieving the assumptions. We also leave it as our promising future work.

Author Contributions: Conceptualization, H.B.; software, H.B.; data curation, H.B.; writing—original draft preparation, H.B. and C.M.K.; writing—review and editing, C.M.K.; supervision, C.M.K.; project administration, H.B.; funding acquisition, H.B. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by Incheon National University (International Cooperative) Research Grant in 2019.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Liu, C.; Layland, J. Scheduling Algorithms for Multi-programming in A Hard-Real-Time Environment. *J. ACM* **1973**, *20*, 46–61. [CrossRef]
2. Brandenburg, B.B.; Gul, M. Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations. In Proceedings of the IEEE Real-Time Systems Symposium (RTSS), Porto, Portugal, 29 November–2 December 2016; pp. 1–15.
3. Melani, A.; Bertogna, M.; Bonifaci, V.; Marchetti-Spaccamela, A.; Buttazzo, G. Schedulability Analysis of Conditional Parallel Task Graphs in Multicore Systems. *IEEE Trans. Comput.* **2016**, *66*, 339–353. [CrossRef]
4. Biondi, A.; Buttazzo, G.C.; Bertogna, M. Schedulability Analysis of Hierarchical Real-Time Systems under Shared Resources. *IEEE Trans. Comput.* **2016**, *65*, 1593–1605. [CrossRef]
5. Shepard, D.; Bhatti, J.; Humphreys, T. Drone Hack: Spoofing Attack Demonstration on a Civilian Unmanned Aerial Vehicle. *GPS World* **2012**, *23*, 30–33.
6. Russotto, B.F. W32.stuxnet Dossier. Available online: http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf. (accessed on 6 March 2020).

7. Koscher, K.; Czeskis, A.; Roesner, F.; Patel, S.; Kohno, T.; Checkoway, S.; McCoy, D.; Kantor, B.; Anderson, D.; Shacham, H.; et al. Experimental Security Analysis of a Modern Automobile. In Proceedings of the IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 16–19 May 2010; pp. 447–462.
8. Checkoway, S.; Mccoy, D.; Kantor, B.; Anderson, D.; Shacham, H.; Savage, S.; Koscher, K.; Czeskis, A.; Roesner, F.; Kohno, T. Comprehensive Experimental Analyses of Automotive Attack Surfaces. In Proceedings of the Usenix Security Symposium, San Francisco, CA, USA, 8–12 August 2011; pp. 1–16.
9. Son, J.; Alves-Foss, J. Covert Timing Channel Analysis of Rate Monotonic Real-Time Scheduling Algorithm in MLS Systems. In Proceedings of the IEEE Information Assurance Workshop, West Point, NY, USA, 21–23 June 2006; pp. 361–368.
10. Yoon, M.K.; Mohan, S.; Choi, J.; Sha, L. Memory Heat Map:Anomaly detection in real-time embedded systems using memory behavior. In Proceedings of the 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 8–12 June 2015; pp. 1–6.
11. Zadeh, M.M.Z.; Salem, M.; Kumar, N.; Cutulenco, G.; Fischmeister, S. SiPTA: Signal processing for trace-based anomaly detection. In Proceedings of the ACM & IEEE International Conference on Embedded Software, Jaypee Greens, India, 12–17 October 2014; pp. 1–10.
12. Bertogna, M.; Cirinei, M.; Lipari, G. Schedulability Analysis of Global Scheduling Algorithms on Multiprocessor Platforms. *IEEE Trans. Parallel Distrib. Syst.* **2009**, *20*, 553–566. [CrossRef]
13. Joseph, M.; Pandya, P. Finding response times in a real-time system. *Comput. J.* **1986**, *29*, 390–395. [CrossRef]
14. Volp, M.; Hamann, C.J.; Hartig, H. Avoiding timing channels in fixed-priority schedulers. In Proceedings of the ACM Symposium on Information, Computer and Communication Security, Tokyo, Japan, 18–20 March 2008; pp. 1–10.
15. Yoon, M.; Mohan, S.; Chen, C.; Sha, L. TaskShuffler: A Schedule Randomization Protocol for Obfuscation against Timing Inference Attacks in Real-Time Systems. In Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS), Vienna, Austria, 11–14 April 2016; pp. 1–12.
16. NVIDIA. Drive AGX Pegasus. Available online: https://developer.nvidia.com/drive/drive-agx (accessed on 8 April 2020).
17. Fisher, N.; Goossens, J.; Baruah, S. Optimal Online Multiprocessor Scheduling of Sporadic Real-Time Tasks is Impossible. *Real-Time Syst.* **2010**, *45*, 26–71. [CrossRef]
18. Bertogna, M.; Cirinei, M. Response-Time Analysis for globally scheduled Symmetric Multiprocessor Platforms. In Proceedings of the IEEE Real-Time Systems Symposium (RTSS), Tucson, AZ, USA, 3–6 December 2007.
19. Bernat, G.; Burns, A. New results on fixed priority aperiodic servers. In Proceedings of the IEEE Real-Time Systems Symposium (RTSS), Phoenix, AZ, USA, 1–3 December 1999; pp. 68–78.
20. Zabos, A.; Davis, R.; Burns, A.; Harbour, M.G. Spare capacity distribution using exact response-time analysis. In Proceedings of the Conference on Real-Time and Network Systems, Paris, France, 26–27 October 2009; pp. 97–106.
21. Cachin, C. Entropy Measures and Unconditional Security in Cryptography. Ph.D. Thesis, ETH Zurich, Zurich, Switzerland, 1997; pp. 1–143.
22. Shannon, C. A mathematical theory of communication. *Bell Syst. Tech. J.* **1948**, *27*, 379–423. [CrossRef]
23. Dodis, Y.; Smith, A. Entropic security and the encryption of high entropy messages. In Proceedings of the International Conference on Theory of Cryptography, Cambridge, MA, USA, 10–12 February 2005; pp. 1–22.
24. Mohan, S.; Yoon, M.K.; Pellizzoni, R.; Bobba, R. Real-Time Systems Security through Scheduler Constraints. In Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS), Madrid, Spain, 8–11 July 2014; pp. 129–140.
25. Baek, H.; Lee, J.; Lee, Y.; Yoon, H. Preemptive Real-Time Scheduling Incorporating Security Constraint for Cyber Physical Systems. *IEICE Trans. Inf. Syst.* **2016**, *E99-D*, 2121–2130. [CrossRef]
26. Baek, H.; Lee, J. Incorporating Security Constraints into Mixed-Criticality Real-Time Scheduling. *IEICE Trans. Inf. Syst.* **2017**, *E100.D*, 2068–2080. [CrossRef]
27. Pellizzoni, R.; Paryab, N.; Yoon, M.; Bak, S.; Mohan, S.; Bobba, R. A generalized model for preventing information leakage in hard real-time systems. In Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Seattle, WA, USA, 13–16 April 2015; pp. 271–282.

28. Krüger, K.; Völp, M.; Fohler, G. Vulnerability Analysis and Mitigation of Directed Timing Inference Based Attacks on Time-Triggered Systems. In Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS), Barcelona, Spain, 3–6 July 2018; pp. 1–17.
29. Baek, H.; Jung, N.; Chwa, H.S.; Shin, I.; Lee, J. Non-Preemptive Scheduling for Mixed-Criticality Real-Time Multiprocessor Systems. *IEEE Trans. Parallel Distrib. Syst.* **2018**, *29*, 1766–1779. [CrossRef]