TaskShuffler: A Schedule Randomization Protocol for Obfuscation Against Timing Inference Attacks in Real-Time Systems

Man-Ki Yoon*, Sibin Mohan†, Chien-Ying Chen* and Lui Sha*
*Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801
†Information Trust Institute, University of Illinois at Urbana-Champaign, Urbana, IL 61801
Email: {mkyoon, sibin, cchen140, lrs}@illinois.edu

***Abstract*—The high degree of predictability in real-time systems makes it possible for adversaries to launch timing inference attacks such as those based on side-channels and covert-channels. We present TaskShuffler, a schedule obfuscation method aimed at randomizing the schedule for such systems while still providing the real-time guarantees that are necessary for their safe operation. This paper also analyzes the effect of these mechanisms by presenting *schedule entropy* – a metric to measure the uncertainty (as perceived by attackers) introduced by TaskShuffler. These mechanisms will increase the difficulty for would-be attackers thus improving the overall security guarantees for real-time systems.**

I. Introduction

Increased computational power and connectivity in modern real-time systems exposes hitherto unknown security flaws. Threats to such systems are growing, both in number as well as sophistication, as demonstrated by recent attacks [1], [10], [21], [23]. A successful attack on systems with real-time properties can have disastrous effects, from loss of human life to damages to machines and/or the environment. Hence, there is a need to develop effective protection mechanisms that foil attacks on such systems.

An important characteristic of real-time systems, especially those with real-time constraints is that they are predictable by design. Designers ensure that such systems have narrow operational ranges and fixed modes of execution for safety guarantees. Hence, any perturbations to these operational modes can result in the safety of such systems being compromised. On one hand, recent studies [14], [26]–[29], [31] have shown that the security of real-time systems can be improved by taking advantage of these very properties, viz., *predictability* (or *regularity*) in their behavior. Such security mechanisms try to detect abnormal deviations from expected patterns.

On the other hand, adversaries could examine and analyze these predictable patterns of execution in real-time systems and use them as attack surfaces. For instance, an attacker who can determine the timing properties of the system (e.g., the schedule of the system) will be able to launch targeted attacks such as timing side channels to collect information about important tasks [5], [9], [13], [16] or even set up new covert channels [22], [25]. Such attacks can succeed due to the *limited uncertainty* in the repeating schedules generated by a set of periodic, real-time tasks (e.g., Figure 5(a) in Section III-C). Once an attacker is able to gauge the schedule for one *hyper-period*, he can predict, *in a very precise fashion*, the future schedules of the system.

In this paper we propose a novel *schedule randomization protocol* that we call TaskShuffler. TaskShuffler *reduces the determinism perceived by adversaries in real-time schedules*. It achieves this by generating highly unpredictable, randomized schedules that still *meet the real-time requirements of the system*. To this end, TaskShuffler retrofits priority inversions [19] – at each point where a scheduling decision is to be made TaskShuffler selects a *random job* (from those that are ready), irrespective of its priority, subject to some constraints. In the presence of such mechanisms, adversaries are unable to deduce the exact order of execution of real-time tasks even if they are able to completely reconstruct the prior behavior of the system [5]. Hence, they face increased difficulty while initiating timing inference attacks. Yet, designers can ensure that the real-time guarantees of the system are not diluted.

We also introduce the concept of *Schedule Entropy* to measure the increased uncertainty in the real-time schedule. It is based on the Shannon Entropy [20] of the probability distribution of hyper-period schedules. While it is infeasible to obtain the exact schedule entropy for complex systems, we present an approximation calculated from partial observations.

In summary, the main contributions of this paper are:

1) We introduce TaskShuffler– an algorithm that randomizes the schedules of a given task set (using fixed priority scheduling[1]) while guaranteeing its schedulability;
2) A metric to measure the uncertainty introduced by the schedule randomization and present an approximated upper-bound calculated from empirical observations; and
3) A discussion of the effects of various factors on the randomness in the new schedules along with an extensive evaluation.

II. System and Adversary Model

We consider a uniprocessor system consisting of a set of N periodic tasks $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_N\}$. Each task τ_i is characterized by (e_i, p_i, d_i), where e_i is the worst-case execution time, p_i is the minimum inter-arrival time (or period) between successive releases and d_i is the relative deadline. Task priorities are assigned according to the Rate Monotonic (RM) algorithm [12]. $\texttt{Pri}(\tau_i)$ represents the priority of τ_i.

[1] Although we limit ourselves to fixed-priority scheduling in this paper, the concepts can be extended to dynamic priority algorithms.

978-1-4673-8641-8/16/$31.00 ©2016 IEEE

$\texttt{Pri}(\tau_j) < \texttt{Pri}(\tau_i)$ if τ_j has a lower priority than τ_i. Let $hp(\tau_i)$ be the set of tasks that have higher priorities than τ_i and $lp(\tau_i)$ be the set of tasks with lower priority. We assume that priorities are distinct and that $d_i = p_i$ for all i. Each task τ_i releases a new *job* at each invocation. Since $d_i = p_i$, there can be at most one job per task at any time instant (if schedulable) and thus we use the same symbol τ_i to denote its jobs for notational simplicity – hence, we use the terms, task and job, interchangeably.

We assume that the task set Γ is *schedulable* by a fixed-priority preemptive scheduling. That is, the worst-case response time of task i, $wcrt_i$, is less than or equal to the deadline, d_i, where $wcrt_i$ is calculated by the iterative response time analysis [2]. Note that since the task set is schedulable, the following inequality is satisfied for any task $\tau_i \in \Gamma$:

$$wcrt_i = e_i + \sum_{\tau_j \in hp(\tau_i)} \left\lceil \frac{r_i^k}{p_j} \right\rceil e_j \leq d_i, \tag{1}$$

where $r_i^0 = e_i$ and $wcrt_i = r_i^{k+1} = r_i^k$ for some k. Finally, we assume that there is no synchronization or precedence constraints among tasks and that $e_i, p_i, d_i \in \mathbb{N}^+$.

A. Adversary Model

We assume that an adversary knows the timing parameters of the task set as well as the scheduling policy. The main objective of the adversary is to launch security attacks that make use of *deterministic schedules*. For example, an attacker may want to track the execution profile of a target task in an attempt to launch a side-channel attack on it. Figure 1 illustrates an example scenario, in which the attacker task tries to extract the victim task's sensitive information (e.g., a cryptographic key), say, using a cache side-channel attack [13], [16]. The attacker first fills some cache sets with carefully-structured data before the victim task executes. Then, the victim task runs a crypto algorithm with a given input, which will use some of the cache sets. Later, the attacker task reads the cache sets it filled in and measures the latencies. The attacker collects such timing information that provides hints about the victim's crypto algorithm and analyzes them to extract the crypto key.

Such attacks require the adversary to perform monitoring activities for a certain period of time to improve the accuracy of the extracted information. More importantly, the establishment of a narrow time range when the victim task can appear will significantly increase the success rate of such attacks. The attacker may be able to deduce a precise fixed-priority schedule by observing busy periods from the idle task if tasks execute in a static fashion [5]. Or the attacker may exploit the priority relations between tasks. For example, the attacker could hijack a task that has a lower priority than the victim task and carefully construct an execution scenario such that the victim task would preempt the hijacked task. This will enable the attacker to perform the side-channel attack described above. Similarly, the attacker may hijack a higher priority task that runs frequently enough so that it would

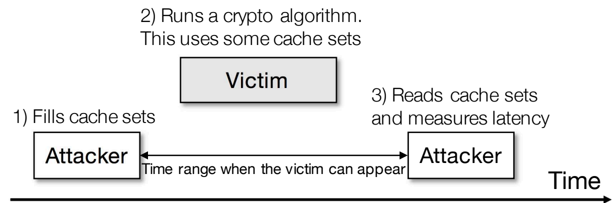

Fig. 1: A cache side-channel attack can extract a sensitive information (e.g., a crypto key) by collecting/analyzing varying cache-access latencies due to the victim's use of the cache. The accuracy increases as the attacker can precisely narrow down the time range when the victim task can appear.

execute before and after an execution of the victim task. Or the attacker may instead insert a new task that can carry out these activities (if the attacker was able to hide the new task from any other detection methods).

We consider such attempts to be effective if the adversary can predict *which* task runs *when* as precisely and as frequently as possible. Conversely, the quality of information available to the adversary degrades with increased uncertainty in the timing properties of the system. We do not make any specific assumptions on the attacker's ability to infer task schedule and to pinpoint the target task(s). The attacker may even have an ability to deduce the exact schedule, which is the worst-case scenario from the perspective of the defender. The objective of TaskShuffler is to make it difficult for the attacker to predict what task runs and when, even in the worst-case. Hence, the attacks mentioned above are likely rendered ineffective.

A covert timing-channel [22], [25] is another class of threats that can benefit from deterministic schedules; tasks can communicate indirectly by altering scheduling behaviors such as execution order as well as blocking and completion times. Even in this case, the accuracy of the information that communicates through such a channel depends heavily on precise timing coordination and ordering.

We assume that the scheduler is trustworthy. Otherwise, the adversary can perform more active attacks than what is described above – this is out of scope for this paper and we intend to analyze it in future work. There are various other threats faced by real-time systems [14], [26]–[29], [31] – most of these techniques are about intrusion detection and hence complementary to those presented here.

III. Schedule Randomization Protocol

We now discuss the core of TaskShuffler, viz., the protocol for randomizing the schedule. This mechanism reduces the *inferability* of the schedule for real-time task sets – hence, even if an observer is able to record the *exact* schedule for a period of time (say, one hyper-period), there is no guarantee that the next period will show the exact same order (and timing) of execution for the tasks. The main idea is that at each scheduling point, we *pick a random task* from the ready queue and let it execute. This is counter-intuitive to most real-time systems where the task with the highest priority would be picked next. The main reason why designers of

real-time systems do not follow such methods is that it (a) leads to priority inversions [19] that, in turn, cause (b) missed deadlines – hence, putting at risk the safety of the system. Our randomization protocol seems to allow random priority inversions at each scheduling decision point.

To solve this problem, we only allow *bounded* priority inversions – i.e., restrict how the schedule may use priority inversions while ensuring that the entire task set still meets its original deadline constraints. To this end, we calculate the amount of priority inversion that each task can endure, in the worst-case, while using the TaskShuffler protocol. If the limit is reached during execution, then we stop allowing lower priority tasks to execute ahead of the current task until its outstanding job completes execution (at which point the limit is reset). The following sections will present the details of the TaskShuffler protocol as well as the related analyses.

A. Bounding Priority Inversions in TaskShuffler

A key step in this protocol is to calculate the maximum amount of time that all the lower priority tasks, $lp(\tau_i)$, of each task, τ_i, can spend executing while τ_i *waits*. Hence, we need to calculate the *budget* allowed for lower priority tasks to execute in the priority inversion mode. We calculate this *worst-case inversion budget*, V_i, for every task τ_i.

V_i is calculated using the worst-case interference from the higher priority tasks. Note that without any priority inversions, a job of τ_i could be delayed by up to $\sum_{\tau_j \in hp(\tau_i)} \lceil wcrt_i/p_j \rceil e_j$ by the higher priority tasks. However, with arbitrary priority inversions, the job could be further delayed because of the following *chain reaction*: the lower priority tasks, $lp(\tau_i)$, delay the higher priority ones, $hp(\tau_i)$, that in turn delay τ_i. In the worst-case, illustrated in Figure 2, the higher priority jobs could be maximally delayed by $lp(\tau_i)$ in a way that the jobs that were released before τ_i's arrival execute *on or after* the release of τ_i.[2] Accordingly, τ_i could take more than $wcrt_i$ to complete because of the additional interference by the deferred executions (also known as back-to-back hit) [2], [18]. Now, without any assumptions on the execution patterns of $lp(\tau_i)$, the upper-bound interference that a job of τ_i can experience due to the higher priority tasks during d_i is

$$I_i = \sum_{\tau_j \in hp(\tau_i)} \left(\left\lceil \frac{d_i}{p_j} \right\rceil + 1 \right) e_j. \quad (2)$$

τ_i cannot experience more than I_i of interference from higher priority tasks because (i) deadline is equal to period and (ii) the task set is schedulable. Hence, no task can have two or more *outstanding* jobs at any time instant.

Note that Equation (2) may not be a tight bound because the worst-case busy interval[3] of τ_i could be much shorter than the deadline d_i and thus experience fewer delays than I_i. Also, for some elements of $hp(\tau_i)$ the additional interference may not ever show up especially if the load from the lower priority jobs is too small to defer higher priority jobs.

[2] Note that this is equivalent to $hp(\tau_i)$ having release jitters [2].

[3] A busy interval of τ_i (also known as a level-π_i busy interval) is a time window $(t_0, t]$ that 1) begins at t_0 when all jobs of priority i or higher released before t_0 have completed and 2) ends at the first time instant t when all jobs of priority i or higher released during $(t_0, t]$ are complete.

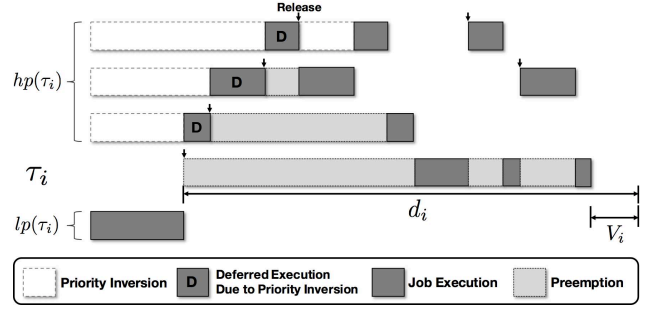

Fig. 2: Worst-case interference from higher priority tasks $hp(\tau_i)$ occurs when they are maximally 'packed' into the busy interval of τ_i due to priority inversions by $lp(\tau_i)$.

Using the worst-case interference obtained above, we now define the worst-case maximum inversion budget.

Definition 1 (Worst-case Maximum Inversion Budget)**.** *The worst-case maximum inversion budget of task i, denoted by V_i, is the maximum amount of time for which all lower priority tasks $lp(\tau_i)$ are allowed to execute while an instance of τ_i is still active (i.e., in the ready queue) while meeting its deadline even in the worst-case scenario. It is calculated by,*

$$V_i = d_i - (e_i + I_i) = d_i - \left[e_i + \sum_{\tau_j \in hp(\tau_i)} \left(\left\lceil \frac{d_i}{p_j} \right\rceil + 1 \right) e_j \right].$$

Example 1. *Consider $\Gamma = \{\tau_0, \tau_1, \tau_2\}$ with the following parameters. Then each V_i is calculated as shown in the last column of the table:*

	p_i	e_i	d_i	V_i
τ_0	5	1	5	4
τ_1	8	2	8	3
τ_2	20	3	20	4

For instance, τ_1 can let $lp(\tau_1) = \{\tau_2\}$ run for 3 time units while not missing the deadline.

Note that V_i may be pessimistic, especially for lower priority tasks, since Equation (2) is calculated using the conservative assumption that τ_i's worst-case response time is d_i. It is also worth noting that V_i can be negative. However, a negative V_i does not necessarily imply that τ_i is not schedulable but that, as we will see later, it might miss the deadline if priority inversions by $lp(\tau_i)$ are not bounded. We will explain what it means for the protocol when $V_i < 0$.

Given a task set that is schedulable, we first calculate the worst-case maximum inversion budget V_i for each task in an offline manner using Equation (2). Hence, the parameters of a task now includes V_i, i.e., (e_i, p_i, d_i, V_i).

The randomization protocol guarantees that deadlines are met by bounding priority inversions using V_i. The scheduler enforces these budgets at run-time by maintaining a per-task counter called *remaining inversion budget* v_i where $0 \leq v_i \leq$

V_i. This represents the time budget left for the lower priority tasks $lp(\tau_i)$ to execute in a priority inversion mode while τ_i has an unfinished job. The counter v_i is reset to the maximum value V_i every time τ_i releases a new job. It is decremented for each time unit that τ_i is blocked by any lower priority job in $lp(\tau_i)$. Once v_i reaches zero *no lower priority task is allowed to run until τ_i completes.*

Theorem 1. *Suppose*

1) *Γ is schedulable with the fixed-priority preemptive scheduling and*
2) *$V_i \geq 0$ for some task $\tau_i \in \Gamma$ and $lp(\tau_i)$ does not delay τ_i more than V_i during its busy interval.*

Then τ_i is still schedulable with the randomization protocol.

Proof: A busy interval of τ_i is composed of its execution, preemptions by $hp(\tau_i)$, and delay by $lp(\tau_i)$ due to priority inversions. The worst-case interference that τ_i can experience due to $hp(\tau_i)$ is bounded by I_i in Equation (2). In addition, τ_i can be delayed for up to V_i by priority inversions during its busy interval. Hence, the worst-case busy interval of τ_i is bounded by

$$e_i + I_i + V_i = e_i + I_i + \big(d_i - (e_i + I_i)\big) = d_i.$$

Therefore, τ_i is schedulable with the randomization. ■

As mentioned earlier, some task τ_x may have a negative worst-case maximum inversion budget, i.e., $V_x < 0$. In this case v_x will always be negative and no lower priority tasks in $lp(\tau_x)$ can run while τ_x has an outstanding job. Hence, the busy interval of τ_x now includes only its own execution and any higher priority jobs. However, $lp(\tau_x)$ could cause τ_x to miss its deadline by inducing the worst-case interference from the higher priority jobs, i.e., I_x. This could happen due to the chain reaction by deferred executions described earlier (and in Figure 2); some or all of $hp(\tau_x)$ released before τ_x's arrival are *packed* by $lp(\tau_x)$'s priority inversions and this can cause τ_x to experience additional delays (i.e., the '+1' term in Equation (2)) – this in turn could lead to $d_x < (e_x + I_x)$.

Therefore, to preserve the schedulability of such task sets we must prevent it from experiencing such additional delays. This is achieved when $hp(\tau_x)$ is not delayed by $lp(\tau_x)$ *even when τ_x has no job to run.*

Definition 2 (Level-τ_x exclusion policy)**.** *If $V_x < 0$ for some τ_x, no job of $lp(\tau_x)$ is allowed to run while any of $hp(\tau_x)$ has an unfinished job.*

The following theorem proves that such a task that has a negative worst-case maximum inversion budget is still schedulable with the above exclusion policy.

Theorem 2. *Suppose*

1) *Γ is schedulable with the fixed-priority preemptive scheduling,*
2) *$V_x < 0$ for some task $\tau_x \in \Gamma$ and the level-τ_x exclusion policy is applied.*

Then τ_x's worst-case response time does not change by the randomization and thus it is still schedulable.

Proof: From 1), τ_x is schedulable without any randomization and thus satisfies Equation (1). Now, by the level-τ_x exclusion policy and $V_x < 0$, no job of $lp(\tau_x)$ can delay $hp(\tau_x)$ and τ_x. Thus, from τ_x's perspective the executions of $lp(\tau_x)$ are equivalent to processor idle times. Now, τ_x's busy interval is not dependent on how $hp(\tau_x)$ are scheduled among themselves but on the total demands from $hp(\tau_x)$ because priority inversions among $hp(\tau_x)$ do not change the total demands. The worst-case delay that can be caused by $hp(\tau_x)$ occurs when they release simultaneously with τ_x [11], [12] and accordingly the worst-case response time does not change. That is, τ_x does not experience the additional delay by the deferred executions of $hp(\tau_x)$ because the chain reaction by $lp(\tau_x)$ is prevented by the level-τ_x exclusion policy. The worst-case response time of τ_x satisfies Equation (1) and thus is schedulable with the randomized scheduling. ■

We now define the following per-task property (calculated offline) so that the scheduler can use the level-τ_x exclusion policy at run-time:

Definition 3 (Minimum inversion priority)**.** *The minimum inversion priority of τ_i, denoted by M_i, is the minimum priority that can delay τ_i (i.e., priority inversion). It is defined by the highest priority among $lp(\tau_i)$ that has a negative worst-case maximum inversion budget. That is,*

$$M_i = \max(\texttt{Pri}(\tau_j) | \tau_j \in lp(\tau_i) \text{ and } V_j < 0), \qquad (3)$$

When there is no such a task, then M_i is set to an arbitrarily minimum priority.

M_i determines which jobs to *exclude* from priority inversions. That is, no job who has a lower priority than M_i can execute as long as τ_i has an unfinished job. Otherwise, again, the task whose priority is M_i (not τ_i) could miss its deadline as illustrated before.

Example 2. *Let $\Gamma = \{\tau_0, \tau_1, \tau_2, \tau_3, \tau_4\}$ with the following parameters:*

	p_i	e_i	d_i	V_i
τ_0	5	1	5	4
τ_1	8	3	8	2
τ_2	20	4	20	-1
τ_3	40	2	40	-1
τ_4	80	4	80	0

The minimum inversion priority of each task is

$$M_0 = M_1 = \texttt{Pri}(\tau_2), M_2 = \texttt{Pri}(\tau_3), \text{ and } M_3 = M_4 = \texttt{Pri}(\tau_4).$$

Hence, τ_3 and τ_4 are not allowed to be scheduled when any of τ_0, τ_1, τ_2 has an unfinished job. This is to prevent τ_2 from missing the deadline.

At run-time, we can guarantee the level-τ_x exclusion policy for all levels by examining only the minimum inversion priority of the highest-priority job in the ready queue, due to the following lemma:

Lemma 1. *Let $L_{\mathcal{R}} = (\tau_{(1)}, \tau_{(2)}, \ldots, \tau_{(|L_{\mathcal{R}}|)})$ be the ready queue of jobs sorted in decreasing order of priority and let $\mathcal{R}$ be the set of the ready jobs. If we exclude $\{\tau_j | \texttt{Pri}(\tau_j) <$*

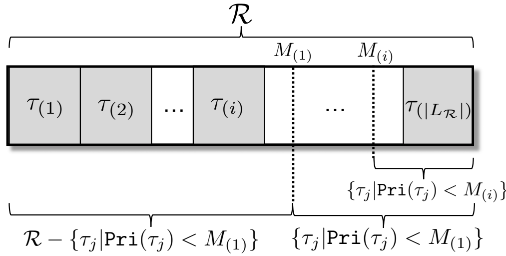

Fig. 3: $\tau_{(i)}$ and $\{\tau_j|\texttt{Pri}(\tau_j) < M_{(i)}\}$ cannot be left over in $\mathcal{R} - \{\tau_j|\texttt{Pri}(\tau_j) < M_{(1)}\}$ at the same time.

$M_{(1)}\}$ from $\mathcal{R}$, then every remaining job $\tau_{(i)}$ satisfies the level-$\tau_{(i)}$ exclusion policy. That is, as long as $\tau_{(i)}$ has an unfinished job, no task whose priority is lower than $M_{(i)}$ can run.

Proof: By the definition of M_i in Equation (3),

$$M_{(1)} \geq M_{(2)} \geq \cdots \geq M_{(|L_\mathcal{R}|)},$$

and thus, as shown in Figure 3,

$$\{\tau_j|\texttt{Pri}(\tau_j) < M_{(1)}\} \supseteq \cdots \supseteq \{\tau_j|\texttt{Pri}(\tau_j) < M_{(|L_\mathcal{R}|)}\}$$
$$\mathcal{R} - \{\tau_j|\texttt{Pri}(\tau_j) < M_{(1)}\} \subseteq \cdots \subseteq \mathcal{R} - \{\tau_j|\texttt{Pri}(\tau_j) < M_{(|L_\mathcal{R}|)}\}.$$

Hence, for any i,

$$\big(\mathcal{R} - \{\tau_j|\texttt{Pri}(\tau_j) < M_{(1)}\}\big) - \big(\mathcal{R} - \{\tau_j|\texttt{Pri}(\tau_j) < M_{(i)}\}\big) = \emptyset.$$

Using a series of set operation $A - B = A \cap B^c$,

$$\big(\mathcal{R} - \{\tau_j|\texttt{Pri}(\tau_j) < M_{(1)}\}\big) \cap \{\tau_j|\texttt{Pri}(\tau_j) < M_{(i)}\} = \emptyset.$$

Therefore, $\tau_{(i)}$ and any job whose priority is lower than $M_{(i)}$ cannot be left over in $\mathcal{R} - \{\tau_j|\texttt{Pri}(\tau_j) < M_{(1)}\}$ at the same time. Thus, no job of $\{\tau_j|\texttt{Pri}(\tau_j) < M_{(i)}\}$ can be selected when $\tau_{(i)}$ has an unfinished job. ■

Hence, at each scheduling decision, we exclude all jobs who have lower priority than $M_{(1)}$ from the selection. Then, by Lemma 1, it is guaranteed that no task in $lp(\tau_x)$ for any τ_x whose $V_x < 0$ can be selected as long as $hp(\tau_x)$ or τ_x has unfinished jobs. Such τ_x (e.g., τ_2 in Example 2) is guaranteed to be schedulable by Theorem 2.

B. The Schedule Randomization Protocol

We now present the randomization protocol that forms the crux of TaskShuffler. The protocol selects a new job using the sequence of steps presented in this section (see Algorithm 1 for a formal description) whenever a scheduling decision is to be made. Let $L_\mathcal{R} = (\tau_{(1)}, \tau_{(2)}, \ldots, \tau_{(|L_\mathcal{R}|)})$ be the current ready queue in which the ready jobs are sorted in decreasing order of priority and $\mathcal{R}$ be the set of all the ready jobs.

Step 1: Finding candidates – Add the highest-priority job $\tau_{(1)}$ to the candidate list, $L_\mathcal{C}$. If its remaining inversion budget is zero or smaller (i.e., $v_{(1)} \leq 0$), then jump to Step 2. Otherwise, iterate over from $\tau_{(2)}$ to $\tau_{(|L_\mathcal{R}|)}$. Let $\tau_{(i)}$ denote the job in the current iteration.

[1-a] Add $\tau_{(i)}$ to the candidate list $L_\mathcal{C}$ if its priority is higher than or equal to $M_{(1)}$, i.e., the minimum inversion priority of the highest-priority job at present.

[1-b] Then, if $\tau_{(i)}$ has zero or smaller remaining inversion budget (i.e., $v_{(i)} \leq 0$), stop the iteration. Otherwise, move to the next priority job.

Step 2: Random selection – *Randomly pick a job from the candidate list $L_\mathcal{C}$.* Suppose it is a job of task τ_s and the current time is t. It will run until the next scheduling decision.

[2-a] If τ_s is the highest-priority in the ready queue (i.e., $\tau_s = \tau_{(1)}$), the next decision will be made either when the job finishes or a new job of another task arrives.

[2-b] If τ_s is *not* the highest-priority in the ready queue (i.e., $\tau_s \neq \tau_{(1)}$), the next decision will be made at time

$$t' = t + \min\big(v_j|\tau_j \in hp(\tau_s) \cap \mathcal{R}\big), \tag{4}$$

unless a new job arrives or τ_s finishes before time t'. The remaining inversion budgets of the jobs in $hp(\tau_s) \cap \mathcal{R}$ (i.e., high priority jobs blocked by τ_s) will be decreased by the amount of time τ_s executes until the next decision.

Note that $\min\big(v_j|\tau_j \in hp(\tau_s) \cap \mathcal{R}\big)$ in Equation (4) is always positive. That is, every job that has a higher priority than the selected job has some remaining inversion budget. Otherwise, τ_s would not have been added to $L_\mathcal{C}$ at Step [1-b].

Theorem 3. *If Γ is schedulable with the fixed-priority preemptive scheduling, then it is still schedulable with the schedule randomization protocol described above.*

Proof: The theorem is an immediate application of Theorems 1 and 2 along with Lemma 1. First, since Γ is schedulable with the fixed-priority preemptive scheduling, condition 1) of both Theorem 1 and 2 is satisfied. Now, any task in Γ falls into either of the following cases:

- For any τ_i whose $V_i \geq 0$, Step [1-b] and Equation (4) guarantee that τ_i cannot be delayed more than V_i by lower priority jobs. This satisfies conditions 2) of Theorem 1. Hence, τ_i is schedulable.
- For any τ_i whose $V_i < 0$, Step [1-a] guarantees the exclusion of $lp(\tau_i)$ when any of $hp(\tau_i)$ is in the ready queue due to Lemma 1. This satisfies condition 2) of Theorem 2. Hence, τ_i is schedulable.

Therefore, any task τ_i in Γ is schedulable with the randomization protocol. ■

Example 3. *Figure 4 shows an example schedule (derived from Example 2) after application of the randomization protocol.[4] It also shows how the v_i values change as tasks execute.*

- *At $t = 0$, the highest-priority job is τ_0 and its minimum inversion priority, $M_{(1)}$, is the priority of τ_2. Hence, due to the level-τ_2 exclusion policy, $lp(\tau_2) = \{\tau_3, \tau_4\}$ are excluded from the selection. The candidate list is $L_\mathcal{C} = \{\tau_0, \tau_1, \tau_2\}$. Among them, τ_2 is randomly selected. The next scheduling decision will be made after 2 time units because of Equation (4) where $\min(v_0, v_1) = 2$.*
- *At $t = 2$, v_1 becomes 0 so $lp(\tau_1)$ cannot run until τ_1 ends. At this point, $L_\mathcal{C} = \{\tau_0, \tau_1\}$ and τ_1 is randomly*

[4] Assuming that all tasks are initially released at time 0.

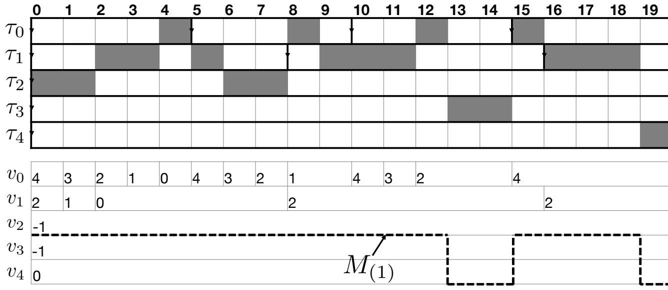

Fig. 4: A randomized schedule of Γ from Example 2.

selected to run. The next decision will be made after a further $\min(v_0) = 2$ *time units.*

- *Going further along, at* $t = 10$*, a new job of* τ_0 *arrives and thus a new scheduling decision is made. In this case,* τ_1 *is selected randomly.*
- *At* $t = 13$*, the highest-priority job is* τ_3 *and so* $M_{(1)}$ *is lowered to the minimum priority. However,* $v_3 < 0$ *and thus* $L_{\mathcal{C}} = \{\tau_3\}$*. Hence, it is selected to run.*
- *At* $t = 15$*, a new job of* τ_0 *arrives and thus* $M_{(1)}$ *is raised back. Hence* τ_3 *and* τ_4 *are excluded from* $L_{\mathcal{C}}$*.*

C. Idle Time Scheduling

Figure 5(a) shows the schedule of the task set from Example 1 for the duration of 15 hyper-periods (each row is one hyper-period). The schedules in the figure are generated by the fixed-priority preemptive scheduling. The number i in each slot depicts the fact that task i executed in that slot; slots with $i = 3$ represent idle times. Figure 5(b) is the schedule generated by the randomization protocol and shows more variations than (a). However, the randomization is limited in that the task executions appear at similar places over multiple hyper-periods, i.e., sandwiched between the deterministic idle time slots. This is because we randomized only the *task schedules*. Hence, there exist separations between task executions and idle times because of the work-conserving nature of the scheduling algorithm. While this increases the schedule randomness, it is still fairly predictable and can be susceptible to timing inference attacks.

To increase the randomness of the schedules, *the range within which a task can appear should be as wide as possible*. This can be done by making the worst-case busy interval of each task as long as possible. Given a fixed workload the only way to do so is intentionally *idling* the processor at seemingly random times. We achieve this by *treating idle times as instances of an additional task in the system*, viz., the *idle task*, τ_I, and applying the aforementioned randomization protocol to the *augmented task set*, $\Gamma' = \Gamma \cup \{\tau_I\}$.[5] The idle task has (a) the lowest-priority, (b) infinite period and deadline and (c) infinite execution time. Hence, the idle task can force all other tasks in the original set Γ to maximally consume their inversion budgets. This effectively makes tasks appear across a wider range of times. Figure 5(c) shows the randomized schedule of the task set that now includes the idle task as part of the schedule randomization protocol. Compared to both (a) and (b), the idle times and task executions spread randomly over wider ranges. As will be seen in Section V, this *idle time scheduling* can significantly increase the randomness in the schedule, especially for low to medium system loads.

[5]In fact, in most real-time operating systems, a special task called *idle task* runs when there is no normal task to run. Hence we can easily incorporate the idle task into TaskShuffler and it is treated no differently than other, normal, tasks from the perspective of the scheduler.

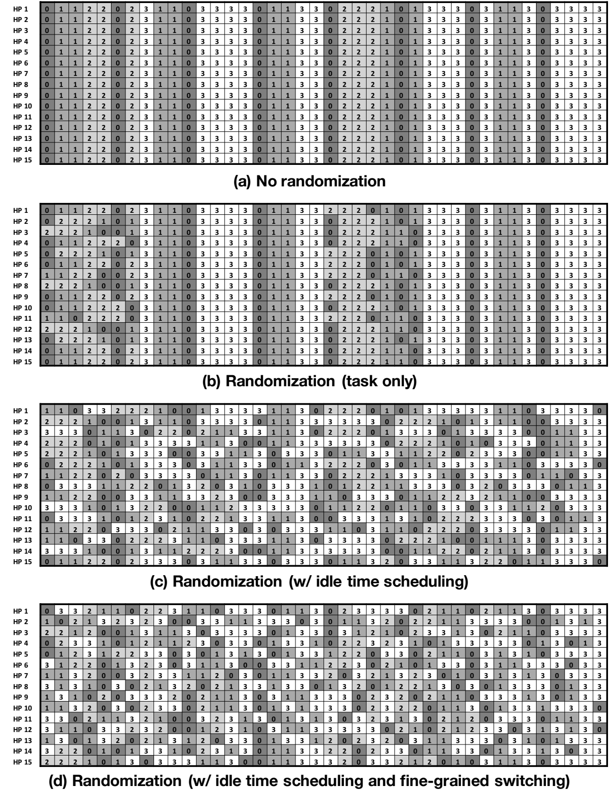

Fig. 5: Schedules of Γ in Example 1 with or without the randomization for the duration of 15 hyper-periods.

D. Fine-grained Switching

Scheduling decisions are made at various points in our system, viz., (a) a new job arrives, (b) the current job finishes or (c) some job(s) can no longer allow priority inversion (see Step [2-b] from Section III-B). When V_i values are large enough for most tasks, jobs will tend to run to completion once scheduled. Hence, to further decrease the inferability of the schedule, we allow a job to randomly *yield* to others in the middle of their execution even if the job's continued execution would not cause missed deadlines in other jobs. This is done by modifying Equation (4) to $t' = t + \Delta t$, where Δt is selected uniformly from the range

$$\left[1, \min\left(v_j | \tau_j \in hp(\tau_s) \cap \mathcal{R}\right)\right].$$

Figure 5(d) shows the schedule of the task set randomized with both, idle time scheduling as well as fine-grained switching.

Algorithm 1 TASKSHUFFLER(t, Γ', $L_\mathcal{R}$)

t: the current time
Γ': the augmented task set $\Gamma \cup \{\tau_I\}$
$L_\mathcal{R}$: the sorted ready queue. $L_\mathcal{R} = (\tau_{(1)}, \tau_{(2)}, \dots, \tau_{(|L_\mathcal{R}|)})$

1: $L_\mathcal{C} \leftarrow \tau_{(1)}$ ▷ i.e., the highest-priority job
2: **if** $v_{(1)} \leq 0$ **then**
3: **return** $\tau_{(1)}$
4: **end if**
5: **for** $\tau_{(i)} = \tau_{(2)}, \dots, \tau_{(|L_\mathcal{R}|)}$ **do**
6: **if** $\texttt{Pri}(\tau_{(i)}) \geq M_{(1)}$ **then**
7: $L_\mathcal{C} \leftarrow L_\mathcal{C} \cdot \tau_{(i)}$ ▷ i.e., add $\tau_{(i)}$ to $L_\mathcal{C}$
8: **end if**
9: **if** $v_{(i)} \leq 0$ **then**
10: Stop the iteration
11: **end if**
12: **end for**
13: $idx \sim [1, |L_\mathcal{C}|]$ ▷ i.e., random selection
14: $\tau_s \leftarrow L_\mathcal{C}[idx]$
15: **if** $idx > 1$ **then** ▷ i.e., priority inversion
16: $\Delta t \sim \Big[1, \min\big(v_j | \tau_j \in hp(\tau_s) \cap \mathcal{R}\big)\Big]$
17: Schedule next decision at $t' = t + \Delta t$
18: **end if**
19: **return** τ_s

E. Summary of the TaskShuffler *Protocol*

Algorithm 1 summarizes the randomization protocol that has been described so far. The procedure TASKSHUFFLER(t, Γ', $L_\mathcal{R}$) is called whenever a scheduling decision is to be made and it returns a randomly selected task (that includes the idle task) from the current ready queue $L_\mathcal{R}$. LINE 1–12 is the part that finds the candidate jobs. The list includes the highest-priority job and any jobs that can afford to wait for a lower priority task after application of the level-τ_x exclusion policy. In LINE 13–18, a random selection is made from the candidate list. If a priority inversion is to occur then the next decision event is scheduled at time t', unless a new job of some task arrives or the selected job τ_s completes before t'.

The protocol iterates over the jobs in the current ready queue once and makes a single draw from the candidates. Hence, the complexity of our randomization algorithm is $\mathcal{O}(n)$, where n is the number of tasks, assuming a single draw from a uniform distribution (LINE 16 and LINE 20) takes no more than $\mathcal{O}(n)$.

IV. SCHEDULE ENTROPY

Given a randomization algorithm such as the one presented in Section III we need to *measure the amount of uncertainty* that has been introduced into the schedule since the main objective was to obfuscate the inherent determinism in the scheduling information. We also need to *quantify* the randomness between different schedules. To tackle these issues, we introduce a new metric called *schedule entropy*. The concept of entropy has been used in the security field [3], [8] to quantify the uncertainty in the amount of information available in for example, ciphertext. We use it as a metric to measure the randomness (or unpredictability) in the real-time schedule.

Let an L-dimensional random vector $\mathcal{S} = (S_0, \dots, S_{L-1})$ represent a hyper-period schedule whose length is L. Each element S_t is a discrete random variable with possible outcomes $\{1, 2, \dots, N\}$ where N is the total number of tasks in Γ. If $S_t = i$, then task i appears at slot t.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\Pr(s_t = 0)$	0.25	0.19	0.14	0.15	0.27	0.43	0.20	0.15	0.07	0.15	0.34	0.26	0.20	0.10	0.10	0.50
$\Pr(s_t = 1)$	0.25	0.27	0.26	0.28	0.34	0.30	0.30	0.00	0.39	0.37	0.27	0.30	0.30	0.30	0.07	0.00
$\Pr(s_t = 2)$	0.25	0.27	0.30	0.29	0.20	0.13	0.24	0.36	0.21	0.17	0.14	0.14	0.14	0.15	0.00	0.00
$\Pr(s_t = 3)$	0.25	0.27	0.29	0.29	0.20	0.13	0.26	0.50	0.33	0.30	0.25	0.30	0.35	0.45	0.84	0.50
$H_\Gamma(S_t)$	**2.00**	**1.99**	**1.95**	**1.95**	**1.96**	**1.82**	**1.98**	**1.44**	**1.80**	**1.90**	**1.94**	**1.95**	**1.92**	**1.78**	**0.80**	**1.00**

Fig. 6: Empirical $\Pr(s_t)$ values for the task set (including the idle task) from Example 1 and slot entropies for $t = 0, \dots, 15$.

We now define the schedule entropy of a task set as a measure of the amount of uncertainty in predicting a particular hyper-period schedule, using the Shannon entropy [20]:

Definition 4 (Schedule entropy)**.** *The schedule entropy of a task set* Γ *is the Shannon entropy of the distribution of the hyper-period schedules. Hence,*

$$H_\Gamma(\mathcal{S}) = -\sum_{s_0=1}^{N} \cdots \sum_{s_{L-1}=1}^{N} \Pr(s_0, \dots, s_{L-1}) \log_2 \Pr(s_0, \dots, s_{L-1}),$$

where $\Pr(s_0, \dots, s_{L-1})$ *is the probability mass function of a hyper-period schedule. The summand is* 0 *if* $\Pr(s_0, \dots, s_{L-1}) = 0$.

In practice, it is infeasible to find the joint probability mass function of $\mathcal{S}$, i.e., $\Pr(s_0, \dots, s_{L-1})$, especially if the length of the hyper-period, L, is large since this requires an enumeration of all possible schedules of length L. This requires an asymptotic complexity of $\mathcal{O}(N^L)$ as there can be at most N choices each time. Hence, we propose the use of an approximation of the true schedule entropy instead. We first define *slot entropy*:

$$H_\Gamma(S_t) = -\sum_{s_t=1}^{N} \Pr(s_t) \log_2 \Pr(s_t), \tag{5}$$

where $\Pr(s_t)$ is the probability mass function of a task appearing at slot t. It measures the uncertainty of job executions at each time slot t. Figure 6 shows empirical $\Pr(s_t)$ values for the task set (including the idle task) from Example 1 for $t = 0, \dots, 15$. The figure shows the situation when the tasks have executed for $10,000$ hyper-periods (they release at the same slot in every hyper-period). The figure shows that certain tasks are more likely to appear than others in particular slots. For instance, the probability of seeing τ_0 at $t = 1$ is smaller than that for other tasks although $\Pr(s_0)$ were uniform. This is because τ_0 completes if it was selected at time $t = 0$. This results in the lower $\Pr(s_1 = 0)$ value.

The upper-bound of slot entropy is $\log_2 n_t$ where n_t is the number of distinct tasks that can appear at slot t. This maximum value is achieved when $\Pr(s_t)$ is uniform across those n_t tasks. Hence, slot entropy increases if more tasks can appear at a slot with similar probabilities.

The difficulty in deriving the true schedule entropy stems from the *dependencies* between slot entropies where a schedul-

ing decision made at slot t affects what happens in slot $t+1$ and onward. Therefore, an enumeration of all possible schedules becomes a combinatorial problem. Hence, we compute an approximation of the schedule entropy by assuming *independence* between the slot entropies:

Definition 5 (Upper-approximated schedule entropy)**.** *An upper-approximation of the schedule entropy of a task set* Γ *is defined as*

$$\widetilde{H}_\Gamma(\mathcal{S}) = \sum_{t=0}^{L-1} H_\Gamma(S_t). \tag{6}$$

I.e., it is the sum *of all slot entropies over a hyper-period.*

This is due to that the joint entropy is less than or equal to the sum of the individuals [20]:

$$H_\Gamma(S_0, \ldots, S_{L-1}) \leq H_\Gamma(S_0) + \cdots + H_\Gamma(S_{L-1}). \tag{7}$$

The equality holds if and only if the scheduling decisions are independent. Since the assumption does not hold in general, the true schedule entropy is always bounded by $\widetilde{H}_\Gamma(\mathcal{S})$.

Despite the approximation, $\widetilde{H}_\Gamma$ is a strong enough criterion to compare the randomness across different, randomized schedules. That is, if $H_\Gamma(\mathcal{S}_1) < H_\Gamma(\mathcal{S}_2)$, then it is highly likely that $\widetilde{H}_\Gamma(\mathcal{S}_1) < \widetilde{H}_\Gamma(\mathcal{S}_2)$ for two different schedules $\mathcal{S}_1$ and $\mathcal{S}_2$. To assess the correlation between the two, we measured the true and the approximated schedule entropies for 900 randomly-generated *small* task sets by simulating each one for 1 million hyper-periods with our randomization protocol. We limited the hyper-period length, L, to 20, because otherwise the simulation cannot cover most of the possible schedules and thus we cannot obtain the probability mass function of hyper-period schedules which is used to calculate the true schedule entropy.[6] Figure 7 shows that there is a *strong correlation* between the true and the approximated schedule entropies. The true schedule entropy increases when more unique hyper-period schedules can be generated. Given a fixed task set, we can generate more, unique, schedules by increasing variations at slots (i.e., more tasks appear at each slot) – this leads to higher entropies in such slots. Also, even if the number of unique schedules is fixed, the true entropy is higher if the probability distribution of schedules is less skewed; i.e., there is a similar probability of generating each of the schedules. In such situations, there is increased uncertainty in observing particular tasks in some slots and, thus, their slot entropies increase. Hence, a higher $H_\Gamma(\mathcal{S})$ would likely result in a higher $\widetilde{H}_\Gamma(\mathcal{S})$. Note that these relations between the true and the approximated entropies do not depend on the length of the hyper-period. Instead, it affects the approximation *error* as can be seen from Figure 7. This is because the error due to the assumption of independence between slots (see the inequality in (7)) accumulate with each slot. Hence, the upper-approximated schedule entropy, $\widetilde{H}_\Gamma(\mathcal{S})$,

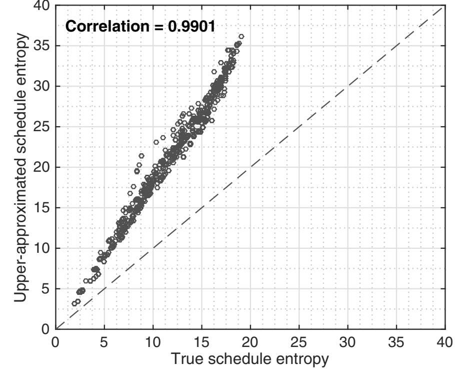

Fig. 7: Correlation between the true, H_Γ, and the upper-approximated, $\widetilde{H}_\Gamma$, schedule entropies.

should be used to compare the *relative* randomness of two schedules. From observing $\widetilde{H}_\Gamma(\mathcal{S}_1) < \widetilde{H}_\Gamma(\mathcal{S}_2)$, one can say $H_\Gamma(\mathcal{S}_1) < H_\Gamma(\mathcal{S}_2)$, with a confidence proportional to the difference.

V. Evaluation

We carry out an extensive evaluation of the TaskShuffler protocol using randomly generated synthetic task sets to gain a better understanding of the effect of the various parameters on the schedule randomization.

A. Evaluation Setup

We generated $6,000$ random, synthetic, task sets evenly from ten base utilization groups, $[0.02+0.1\cdot i, 0.08+0.1\cdot i]$ for $i = 0, ..., 9$, i.e., 600 instances per group. The base utilization of a set is defined as the total sum of the task utilizations. Each group has six sub-groups, each of which has a fixed number of tasks – 5, 7, 9, 11, 13 and 15. This was to generate task sets with an even distribution of tasks. Each task period is a divisor of 3000 (but not smaller than 10). This was to set *a common hyper-period* (i.e., 3000) over all the task sets. The task execution times are randomly drawn from $[1, 50]$. The deadline for each task is the same as its period and priorities are assigned according to the Rate Monotonic algorithm [12].

All of the $6,000$ random sets are guaranteed to be *schedulable* by the fixed-priority preemptive scheduling with the release jitters up to 30% of the period for each task.[7] Note that with *no* release jitters, the schedule entropy by the fixed-priority scheduling (i.e., without our randomization protocol) is always 0 as the same schedule repeats every hyper-period (see Figure 5(a)). This would prevent us from evaluating how much randomness can be introduced by the TaskShuffler protocol compared to the base case. Hence, we allow the simulator to randomly introduce release jitters up to 10%. At

[6]We used the scheduling simulator used in our evaluation in Section V. The parameters are $N \in \{3, 4, 5\}$, $p_i \in \{4, 5, 10, 20\}$, $e_i \in [1, \ldots, 10]$. The 900 sets were equally generated from three utilization groups – $[0.3, 0.4]$, $[0.6, 0.7]$, and $[0.9, 1.0]$.

[7]Note that having release jitters requires a change only in the worst-case maximum inversion budget in Definition 1; it becomes $V_i = d_i - J_i - (e_i + I_i)$, where J_i is the maximum release jitter of task i. Also, I_i does not change as it already includes the jitter effect [2]. It is straightforward to show that the analysis in Section III still holds in the presence of release jitters.

the end of Section V-B, we discuss the impact of different jitter values on the schedule randomization.

Our simulator schedules each task set using the fixed-priority preemptive scheduling policy. It can simulate both situations – with or without the TaskShuffler protocol.

We use $\widetilde{H}_{\Gamma,j}$, the approximated schedule entropy calculated by Equations (5) and (6), for schedules generated until the jth hyper-period. We declare the entropy as having *converged* at the kth hyper-period if

$$\Delta(\widetilde{H}_{\Gamma,j-1}, \widetilde{H}_{\Gamma,j}) = \left| \frac{\widetilde{H}_{\Gamma,j} - \widetilde{H}_{\Gamma,j-1}}{\widetilde{H}_{\Gamma,j-1}} \right| < 0.0001 \text{ (i.e., 0.01\%)}$$

for $j = k - 999, \ldots, k-1, k$, consecutively. The simulation terminates if it reaches a pre-defined time limit (e.g., $10,000$ hyper-periods). We confirmed that the entropy always converged prior to the $10,000^{th}$ hyper-period in all task sets.

B. Results

We first evaluate how much our TaskShuffler protocol can increase the unpredictability of schedules. To measure this, we ran the simulation with and without the randomization algorithm and measured the schedule entropy at the end of the simulations (at which point the entropies have converged). Figure 9 shows the average schedule entropy for each utilization group for the following schemes: (a) the regular fixed-priority schedule, i.e., no randomization, (b) the randomization applied only to tasks in the schedule, (c) randomization applied to tasks *and* the idle times and finally, (d) randomization+idle time scheduling and the fine-grained switching. Figure 8 breaks it down into the per-task set entropies. The results clearly show that the randomization protocol can significantly increase the schedule entropies. The idle time scheduling shows a significant increase in the overall randomness in the system and fine-grained switching improves it further. However, the effects of idle time scheduling is less for higher utilizations task sets (i.e., $U > 0.8$) simply because there are not enough idle times that can be slotted in and randomized among the other tasks. Figure 8 also shows the trend that the entropy increases for larger task sets if the utilization is similar. This is because the slot entropies increase if one can observe more tasks per slot.

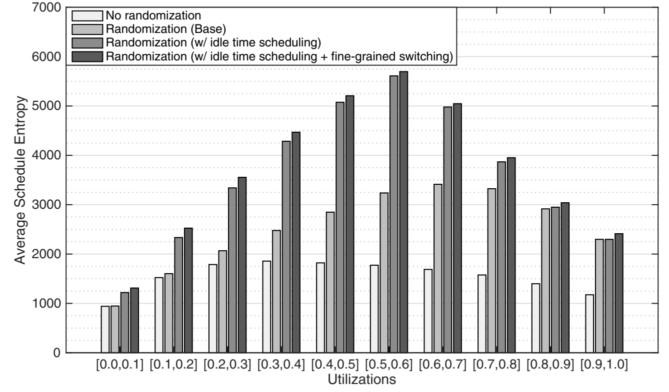

Fig. 9: The average schedule entropies gained by the schedule randomization protocol.

It is interesting to note from the results in both Figures 8 and 9 that our randomization protocol is most effective when

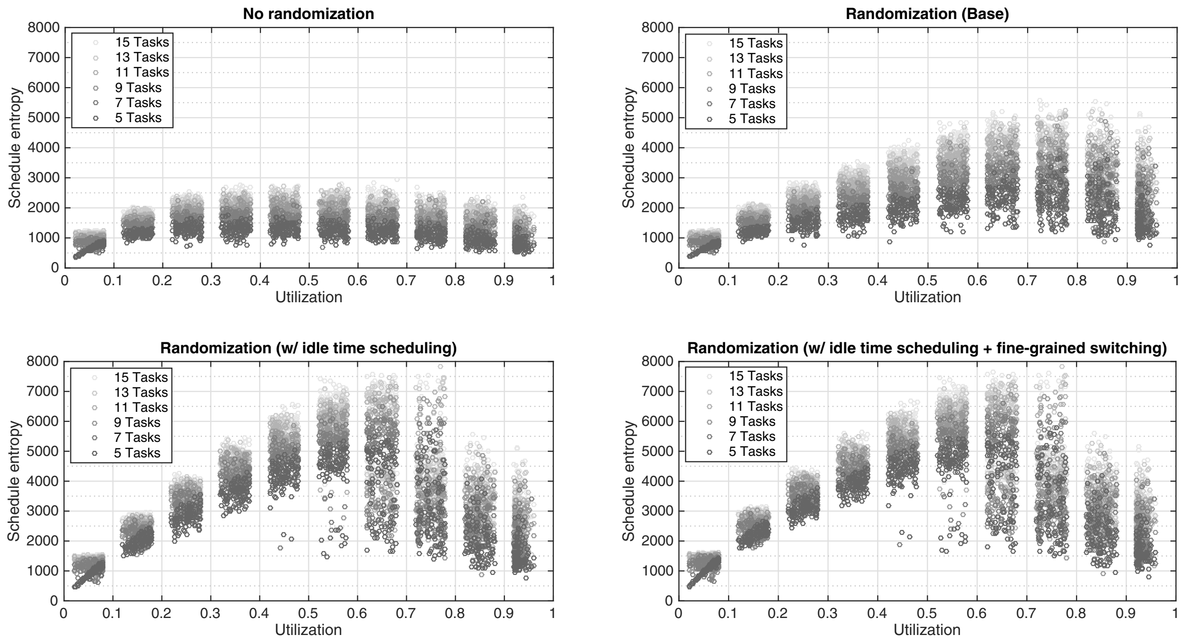

Fig. 8: The per-task set schedule entropy for varying number of tasks and utilizations.

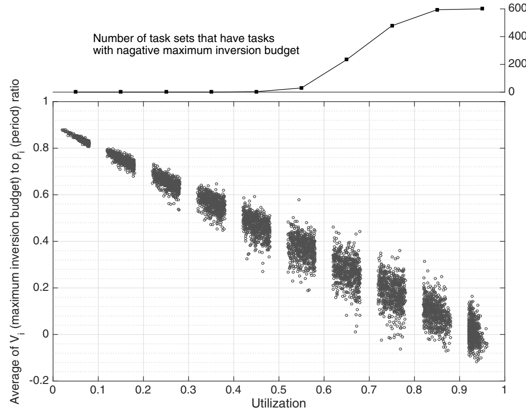

Fig. 10: The average of $\frac{V_i}{p_i}$ in each task set (scatter plot) and the number of task sets that have any $V_i < 0$ (line plot).

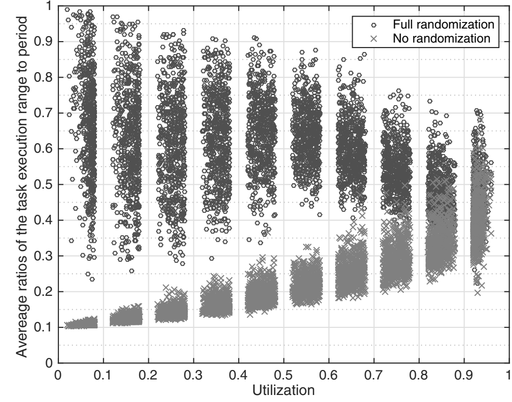

Fig. 11: The average ratio of the task execution range to period.

the system has a *medium* load. When the system load is low, the processor is mostly occupied by idle times and the amount of time spent in normal task executions is too small to show enough variations. In such a case, the candidate job list consists mostly of a few tasks. Hence, the improvement due to the basic randomization (i.e., task-only randomization) compared to the scheme with no randomization is less when the utilization is low. Now, as the utilization increases, there are a larger number of candidates for randomization. However, if the utilization further increases, fewer tasks are available for execution in the *priority inversion mode* since they have much smaller maximum inversion budgets (V_i) even though more candidate jobs are available – they experience greater interference from higher priority tasks. Figure 10 demonstrates this effect – the figure shows the average of maximum inversion budgets to period ratio in each set, i.e., $\sum_{\tau_i \in \Gamma} \frac{V_i}{p_i}/N$. Smaller ratios indicate fewer priority inversions are available. The impact is bigger if some tasks have negative maximum inversion budgets, $V_x < 0$, because of the level-τ_x exclusion policy from Section III-B (see Definition 2); $lp(\tau_x)$ cannot run while $hp(\tau_x)$ have unfinished jobs. Hence, the number of jobs in the candidate list (and thus slot entropies) is likely low when there are many such tasks. As the plot in Figure 10 shows, a higher utilization indicates that more tasks have negative maximum inversion budgets. Hence, the schedule entropies are limited as we saw previously in Figures 8 and 9.

One other way to evaluate the schedule randomization is by widening the range within which each task can appear. That is, the wider the range, the harder it is to predict when tasks would execute. Hence, we measured the first and the last time slots where each task appears and used the difference between them as the execution range for the task. Figure 11 shows the geometric mean of the task execution range to period ratios in each task set. Without the randomization, the ratio is small, i.e., tasks appear within narrow ranges because of the work-conserving nature – hence the schedules are more predictable. As the utilization grows, the ranges become wider simply because the worst-case response times of tasks (especially for lower priority tasks) increase due to the higher loads. Now, with the TaskShuffler protocol turned on, tasks appear in wider ranges as indicated by the higher means in the figure – priority inversions and idle time randomization increase task response times (especially for higher priority tasks). The priority inversions can also draw lower priority jobs close to their release times. The result also shows that the ranges get narrower for higher utilization task sets. This can be explained by the results in Figure 10 – tasks have less space to yield to lower priority ones as the utilization grows and the ranges that the latter can execute can be further limited by the level-τ_x exclusion policy.

As mentioned before, release jitters can introduce some randomizations. In order to assess the impact of release jitters on the schedule randomization, we performed additional experiments with (a) no release jitters and (b) the maximum jitter of 30% of the period for each task. Figure 12 compares the average schedule entropies with and without randomization for different values of maximum release jitter. First of all, without the randomization, the entropy is zero when there is no release jitter. As tasks experience more jitters, the entropies increase because of varying job-release points, as can be seen from the base case (i.e., no randomization). We can also see this trend from the randomized schedules when utilization is lower

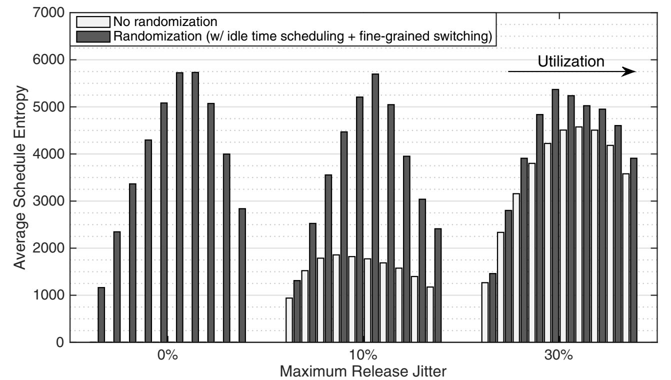

Fig. 12: The impact of release jitters on schedule entropies. In each group, the utilization grows from left to right.

than 0.5. However, release jitters could negatively affect the randomization because the maximum inversion budget reduces as the maximum release jitter increases. Hence, as we can see from the higher utilization groups, the entropies decrease when the maximum jitter becomes 10%. Now, with higher jitters (i.e., 30%), the entropies of the task sets with high utilizations turn to increase. This is because the randomization effect due to varying job-release points outweigh the reduction due to smaller inversion budget. That is, the release jitters have a high impact on the schedule randomness in such environments. This is also why the differences between the entropies of the base case and the randomization are not significant compared to the case with smaller jitters. Our method is thus more effective when tasks experience smaller release jitters, which is a harder situation from the defender's perspective as that increases the chance that the attacker can narrow down the time range within which the victim task can appear.

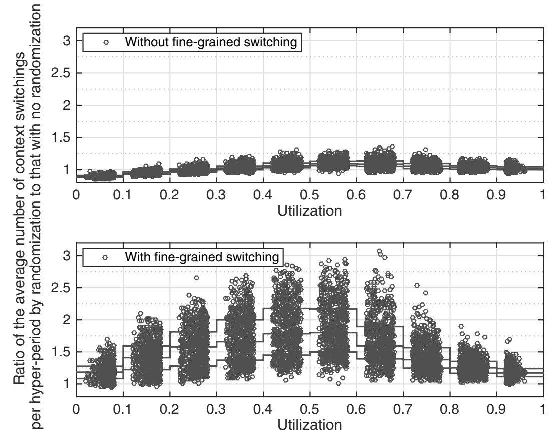

Fig. 13: The increase in the number of context switches by the schedule randomization with (bottom) and without (top) the fine-grained switching.

VI. Limitations and Discussion

While TaskShuffler does increase the difficulty for adversaries to launch timing inference attacks, there are still some areas for improvement for the mechanisms presented in this paper. For instance, we did not attempt to derive an analytic upper-bound on the schedule entropy. An interesting question would be how to compute a *tight* bound and also the *expected entropy* given a task set and a particular randomization algorithm. These can answer questions such as (a) which algorithm can introduce more uncertainty in schedules or (b) how unpredictable (therefore how resilient to timing inference attacks) a task set will be – all without running extensive simulations. In addition, such analyses can enable additional optimizations in the randomization protocol for increased security. We intend to address these topics in future work.

Finally, there is *cost* to be assessed for the randomization protocol. Due to the increased variation in the generated schedules, one might expect an increase in the number of context switches. Hence, we measured the average number of context switches per hyper-period. Then, for each of the $6,000$ task sets, we calculated the ratio between configurations with and without randomization. Figure 13 shows the results for two configurations – randomization with idle scheduling (top) and that with both idle scheduling and fine-grained switching (bottom). The horizontal/step-like lines represent the first, second and third quartiles of the ratio values in each utilization group. First of all, the trends are similar to those in Figure 9 – higher entropy results in more context switches and vice versa. Second, the priority inversion itself does not increase the number of context switches much – this can be seen by comparing the two plots. Without the fine-grained switching, the extra context switches number 10% in average for the utilization group of $[0.5, 0.6]$ (which achieved the highest average entropy, in Figure 9). We can see from the bottom plot that the fine-grained switching is the dominant source of the increased number of context switches. The additional switches number about 80% in average for the utilization group of $[0.5, 0.6]$. Hence, it is advisable to restrict the fine-grained switching or to limit the minimum switching interval for such systems that are sensitive to context switch overheads. As Figure 9 has shown, the schedule entropy can be significantly increased even without the fine-grained switching. While these costs will decrease the overall performance of such systems, we believe that TaskShuffler provides *increased security* in a *measurable* way. Hence, system designers can know the cost for preventing timing inference attacks and account for it in their designs.

VII. Related Work

Researchers have demonstrated that real-time scheduling can be a source for covert channels [22], [24], [25]. Son *et al.* [22] showed that RM scheduling can be the victim of covert timing channels. Völp *et al.* [25] proposed modifications to the scheduler that alters thread blocks that may leak information to the idle thread – it aims to avoid the exploitation of timing channels while achieving real-time guarantees. They also examined covert channels in shared resources used in real-time systems and developed transformed locking protocols [24]. Mohan *et al.* [15] considered information leakage through storage timing channels using architectural resources (e.g., caches) shared between real-time tasks with different security levels. The authors introduced a modification to the fixed-priority scheduling algorithm and a state cleanup mechanism to mitigate information leakage through shared resources. This work was further extended to a more generalized task model [17]. Nonetheless, such countermeasures may not be completely effective against timing inference attacks that focus on the deterministic scheduling behaviors. TaskShuffler works to break this very predictability (inherent in real-time scheduling) by introducing randomness.

The notion of randomization has been used in many studies to harden security mechanisms. Zhang *et al.* [30] introduced cache random-eviction and cache random permutation for removing cache side-channels and mitigating the information leakage. Chan *et al.* [4] adopted random key distribution and

presented random-pairwise keys scheme to reduce the likelihood of communication channels being compromised in sensor networks. Davi *et al.* [7] addressed code-reuse attacks by proposing a software diversification tool that applies Address Space Layout Randomization (ASLR) to randomize the code throughout all sections, on-the-fly, for each execution instance. Crane [6] further raised the security of the code randomization to another level by enabling execute-only memory in modern CPUs to eliminate code leakage. Nevertheless, our work differs from the above since we are reducing the *inferability* of real-time system schedules. Also, these, existing techniques did not have to contend with real-time constraints.

VIII. Conclusion

Timing inference attacks can be quite insidious, especially in real-time systems. Adversaries are able to take advantage of the precise properties of such systems that make them safer – their predictable execution patterns. By using protocols such as the ones presented by TaskShuffler, designers of real-time systems are now able to improve their security guarantees, thus ensuring *increased safety* – which is the main goal for such systems. Future real-time systems can now include, as part of their design, the resources necessary for implementing such techniques. We are also able to provide a glimpse into the means necessary to *measure* the security of a system – albeit for those with real-time constraints – but this is a start towards developing metrics for the field of systems security in general.

Acknowledgment

The authors would like to thank the anonymous reviewers and the shepherd for their valuable comments and suggestions. This work is supported in part by grants from NSF CNS 13-02563, NSF CNS 12-19064, NSF CNS 14-23334, ONR/Navy N00014-12-1-0046, and ONR/Navy N00014-13-1-0707. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of sponsors.

References

[1] Jeep Hacking 101. *IEEE Spectrum*, Aug 2015. http://spectrum.ieee.org/cars-that-think/transportation/systems/jeep-hacking-101.

[2] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8:284–292, 1993.

[3] C. Cachin. *Entropy Measures and Unconditional Security in Cryptography*. PhD thesis, ETH Zurich, 1997. Hartung-Gorre Verlag, Konstanz.

[4] H. Chan, A. Perrig, and D. Song. Random key predistribution schemes for sensor networks. In *Proc. of the IEEE Symposium on Security and Privacy*, 2003.

[5] C.-Y. Chen, S. Mohan, and R. Bobba. Schedule-based side-channel attack in fixed-priority real-time systems. Technical report, University of Illinois at Urbana Champaign, 2015. http://hdl.handle.net/2142/88344.

[6] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *Proc. of the IEEE Symposium on Security and Privacy*, 2015.

[7] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi. Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and arm. In *Proc. of the ACM SIGSAC symposium on Information, computer and communications security*, 2013.

[8] Y. Dodis and A. Smith. Entropic security and the encryption of high entropy messages. In *Proc. of the International Conference on Theory of Cryptography*, 2005.

[9] P. C. Kocher. Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems. In *Proc. of the Annual International Cryptology Conference*, volume 1109 of *Lecture Notes in Computer Science*. Springer, 1996.

[10] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental security analysis of a modern automobile. In *Proc. of the IEEE Symposium on Security and Privacy*, 2010.

[11] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proc. of the IEEE Real-Time Systems Symposium*, 1990.

[12] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[13] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. Lee. Last-level cache side-channel attacks are practical. In *Proc. of the IEEE Symposium on Security and Privacy*, 2015.

[14] S. Mohan, S. Bak, E. Betti, H. Yun, L. Sha, and M. Caccamo. S3A: Secure system simplex architecture for enhanced security and robustness of cyber-physical systems. In *Proc. of the ACM Conference on High Confidence Networked Systems*, 2013.

[15] S. Mohan, M.-K. Yoon, R. Pellizzoni, and R. Bobba. Real-time systems security through scheduler constraints. In *Proc. of the Euromicro Conference on Real-Time Systems*, 2014.

[16] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of aes. In *Proc. of the Cryptographers' Track at the RSA Conference on Topics in Cryptology*, 2006.

[17] R. Pellizzoni, N. Paryab, M.-K. Yoon, S. Bak, S. Mohan, and R. Bobba. A generalized model for preventing information leakage in hard real-time systems. In *Proc. of the IEEE Real-Time Embedded Technology and Applications Symposium*, 2015.

[18] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proc. of the IEEE Real-Time Systems Symposium*, 1988.

[19] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, Sept. 1990.

[20] C. Shannon. A mathematical theory of communication. *Bell System Technical Journal, The*, 27(3):379–423, July 1948.

[21] D. Shepard, J. Bhatti, and T. Humphreys. Drone hack: Spoofing attack demonstration on a civilian unmanned aerial vehicle. *GPS World*, August 2012.

[22] J. Son and J. Alves-Foss. Covert timing channel analysis of rate monotonic real-time scheduling algorithm in mls systems. In *Proc. of the IEEE Information Assurance Workshop*, 2006.

[23] H. Teso. Aicraft hacking. In *Proc. of the Fourth Annual HITB Security Conference in Europe*, 2013.

[24] M. Völp, B. Engel, C.-J. Hamann, and H. Härtig. On confidentiality preserving real-time locking protocols. In *Proc. of the IEEE Real-Time Embedded Technology and Applications Symposium*, 2013.

[25] M. Völp, C.-J. Hamann, and H. Härtig. Avoiding timing channels in fixed-priority schedulers. In *Proc. of the ACM Symposium on Information, Computer and Communication Security*, 2008.

[26] M.-K. Yoon, S. Mohan, J. Choi, M. Christodorescu, and L. Sha. Learning execution contexts from system call distribution for intrusion detection in embedded system. 2015. http://arxiv.org/abs/1501.05963.

[27] M.-K. Yoon, S. Mohan, J. Choi, J.-E. Kim, and L. Sha. SecureCore: A multicore-based intrusion detection architecture for real-time embedded systems. In *Proc. of the IEEE Real-Time Embedded Technology and Applications Symposium*, 2013.

[28] M.-K. Yoon, S. Mohan, J. Choi, and L. Sha. Memory Heat Map: Anomaly detection in real-time embedded systems using memory behavior. In *Proc. of the ACM/EDAC/IEEE Design Automation Conference*, 2015.

[29] M. M. Z. Zadeh, M. Salem, N. Kumar, G. Cutulenco, and S. Fischmeister. SiPTA: Signal processing for trace-based anomaly detection. In *Proc. of the International Conference on Embedded Software*, 2014.

[30] T. Zhang and R. B. Lee. New models of cache architectures characterizing information leakage from cache side channels. In *Proc. of the Annual Computer Security Applications Conference*, 2014.

[31] C. Zimmer, B. Bhatt, F. Mueller, and S. Mohan. Time-based intrusion detection in cyber-physical systems. In *Proc. of the International Conference on Cyber-Physical Systems*, 2010.