Work-in-Progress: RWS - A Roulette Wheel Scheduler For Preventing Execution Pattern Leakage

Ying Zhang  Lingxiang Wang  Wei Jiang  Zhishan Guo
Department of Computer Science, Missouri University of Science and Technology
{yzhg4, wjiang, guozh}@mst.edu

Abstract—Many real-time systems are safety-critical, where reliability is crucial. Under traditional scheduling mechanism, the execution patterns of the tasks on such system can be easily derived from side-channel attacks, such that attackers can launch short high-priority tasks at critical instants which may cause deadline miss for high-critical tasks. In order to protect the system from such kind of attacks, this paper proposes the roulette wheel scheduler (RWS) to randomize the task execution pattern. Under RWS, probabilities will be assigned to each task at predefined scheduling points, and the choice for execution is randomized, such that the execution pattern is no longer fixed. We formalize the concept of schedule entropy the additional safety provided by any randomized scheduler. It is used to measure the amount of uncertainty introduced by the new scheduler.

Index Terms—randomized scheduler, execution pattern, roulette wheel, schedule entropy.

I. INTRODUCTION

Real-time computing paradigm is widely used to support various critical computations and control systems, such as navigation of automobiles and avionics [1] [2] [3]. These systems are usually designed in a multi-task programming pattern with a sequence of sub-tasks for execution. To ensure the real-time performance, the completion of many sub-tasks is constrained by predefined deadlines. As a result, cyber attackers will try to compromise the temporal correctness [4] [5] and damage the reliability of the real-time systems.

Predictability is a significant property of real-time systems [6]. Given a real-time periodic task set running under a traditional scheduling algorithm (e.g., earliest deadline first (EDF) or deadline monotonic), the tasks with regular released time and predefined execution time are scheduled periodically, so that the execution sequence of the task set can be the same in any two hyper-periods. Note that, we define the duplicated execution sequences as the execution pattern of the system. Under such circumstance, if we obtain the execution information of the task set in a long period, the execution sequence of the tasks can be gauged/determined in one hyper-period, where the execution pattern is able to be conjectured/inferred in the following hyper-period. Thus, such predictability property of real-time systems can be taken advantage by attackers: it is possible to launch an attack if the attacker can collect enough information of the system and predict the execution patterns.

Based on the aforementioned predictability property of the real-time systems, one possible way to launch an attack is through side channels [7] [8]. Adversaries exploit the different memory addresses access time stamps of tasks during execution [9], since the tasks access the memory with a regularity (the execution sequences are the same in different hyper-period), the execution pattern can be derived because of information leakage. The leakage of execution pattern causes the real-time systems vulnerable to adversaries. They can insert a redundant computation task of higher priority to compete for computing resource with the task approaching the deadline. As a result, critical tasks might break the temporal constraint; then the system can be compromised under the attack, which lead to catastrophic accidents.

Related work. Therefore, to defend such attacks, researchers have proposed two sets of approaches: randomized scheduling and cache flushing [7].

Randomized scheduling approaches aim at reducing the regularity of the execution sequence, to prevent attackers from predicting task execution pattern even if they can trace the task execution information. Yoon et al. [10] proposed a novel randomization scheduling protocol: TaskShuffler. This protocol randomizes the execution sequence of tasks and increases the non-regularity of the execution pattern. At each scheduling point, it first forms a candidate task set based on criticality levels and the deadlines, and then randomly selects a task from the candidates to decrease the regularity in the execution sequence. However, the limited size of candidate task set restricts the resource dedicated to specific tasks at each scheduling point. Moreover, even they introduce the schedule entropy to measure the non-regularity in Taskshuffler protocol, they did not derive an analytic upper-bound on the schedule entropy, so the level of security cannot be measured accurately.
Compared with randomized scheduling, cache flushing technique clears up the items or invalidates the data from the cache \([11],[12]\) when a task finished execution. Without the data leakage, attackers cannot further exploit the priority order and predict the execution pattern. In 2014, Mohan et al. \([1]\) provided a cache flushing method for defending the cache-channel attack at the design phase. They mainly focused on fixed priority (FP) scheduling algorithms. Based on this work, Pellizzoni et al. \([13]\) proposed a generalized model for defending the potential information leakage in all possible shared resources. However, these techniques suffer the significant performance overhead compared with randomized scheduling.

**Our contribution.** Therefore, in our paper, we apply the randomized scheduling technique to decrease the regularity of the execution sequence. We propose roulette wheel scheduler (RWS) for preventing execution pattern leakage. Roulette wheel selection strategies have been widely applied in industry \([14],[15]\); it is easy to apply to practical systems to deal with the scheduling problem as well \([16],[17]\). For example, Hou \([16]\) et al. utilized this strategy to solve the multiprocessor scheduling problem; Ishibuchi et al. \([18]\) combined the roulette wheel selection with genetic search for clarifying multi-objective permutation flowshop scheduling. Hence, the proposed RWS could be applied to real-time systems without modification of the system architecture.

During run time, RWS generates the probability assignment rule at each scheduling point for the activate tasks in this system, while guaranteeing the schedulability of the system. The system randomly picks up a task to execute based on the probability distribution, such that the regular execution sequence becomes randomized.

Our main advantages are summarized as follows:

- RWS can be applied to both fixed and dynamic priority scheduling algorithm. Moreover, it could handle the settings with sporadic tasks, where release patterns can be relaxed.
- RWS considers all activated tasks as candidates in each scheduling point to enhance the anonymity of task execution. In other words, every candidate task can be selected to execute between two scheduling points.
- The concept of schedule entropy is formalized to measure the non-regularity for a given schedule within a given period. The upper-bound of schedule entropy could be provided according to the probability distribution provided by RWS.

**Organization.** The remainder of this paper is organized as follows: Section II presents the workload model, while Section III defines this problem and presents an example for the probability based scheduling. Section IV proposes Roulette Wheel Scheduler (RWS) to maximize the non-regularity of the execution sequence and briefly introduces the future research directions.

**II. System Model and Terminology**

In this section, we first describe the workload model being considered. Then we introduce the terminologies to be used in rest of the paper.

**Workload.** We consider a predefined workload to be run on a fully preemptive uni-core system. The workload can be characterized by a set of sporadic tasks \(\tau = \{\tau_1, \tau_2, \cdots, \tau_n\}\). All tasks have implicit deadlines, so each task can be denoted as \((C_i,T_i)\), where \(C_i\) is the worst-case execution time (WCET), and \(T_i\) is the minimum separation of the release time. Task priority is assigned by the task’s deadline: a task with an earlier deadline has a higher priority.

A sporadic task \(\tau_i\) may generate infinite number of jobs \(\{\tau_{i,1}, \tau_{i,2}, \cdots\}\), where consecutive releases must be at least \(T_i\) time units apart. As an instance of a task, each job \(\tau_{i,j}\) can be characterized by 3-tuple \((a_{i,j}, c_{i,j}, d_{i,j})\):

- \(a_{i,j} \geq 0\) denotes its release time (the first moment that the job can start to execute);
- \(c_{i,j} = C_i\) is the worst case execution time (WCET);
- \(d_{i,j} \geq a_{i,j}\) is the absolute deadline, which occurs \(T_i\) time units after its release time.

**Remark 1.** In this paper, we assume the predefined set task is feasible, i.e., schedulable under EDF scheduling algorithm (since we are considering uniprocessor), which satisfies:

\[
\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1
\]  

(1)

When conducting the randomized scheduling, it is possible that the lower priority task will be executed before the higher priority one, which is also known as priority inversion \([19]\). In order to guarantee the schedulability, we introduce the following concept.

**Definition II.1.** (Semi-inversion time budget). At a scheduling point \(t\), for a released job \(\tau_{i,j}\) in the ready queue \(J\) (which contains all jobs waiting to be executed), the semi-inversion time budget can be defined as

\[
B_{i,t} = d_{i,j} - t - \sum_{k \in h p (i), \tau_{k,q} \in J} \left( \frac{d_{i,j} - d_{k,q}}{T_k} \times C_k + r_{k,t} \right)
\]  

(2)

where \(h p (i)\) indicates the jobs with higher priority than \(\tau_{i,}\) and \(r_{k,t}\) represents the remaining workload for a job \(\tau_{k,q}\) at time instant \(t\).

\(B_{i,t}\) represents the total amount of allowed time (from \(\tau_{i,j}\)’s perspective) for lower priority jobs (including job \(\tau_{i,j}\)) to execute before the deadline of \(\tau_{i,j}\).

For the task execution sequence under a given scheduling algorithm such as EDF, the execution sequence/pattern is fixed over different hyper-periods, such regularity may cause scheduling information leakage.
After we apply the randomized scheduler, we want to measure the non-regularity or randomness such that the security level of scheduling information can be well evaluated.

**Definition II.2.** (Slot entropy).

\[
H(t) = - \sum_{\tau_{i,j} \in J} p_{i,t} \log p_{i,t} \tag{3}
\]

where \(p_{i,t}\) is the probability assigned for job \(\tau_{i,j}\) at the scheduling point \(t\). Note that at any time, there will be at most one job belonging to a task in the ready queue. So we use \(i\) to represent the job’s releaser.

Slot entropy \(H(t)\) represents the randomness metric of the job set \(J\) at a scheduling point \(t\).

In order to evaluate the security level of the giving randomized scheduling, we should further provide the schedule entropy in any hyper-period [10].

**Definition II.3.** (Schedule entropy). Schedule entropy \(S^*\) is a measure of the non-regularity associated with a given schedule in a hyper-period of length \(L\).

\[
H = \sum_{t \in L} H(t) \tag{4}
\]

### III. Problem Definition

In this section, we first define the variables to be used, and then provide the formal definition of the problem to be solved. An illustrated example is given in the last subsection.

We want to generate a new scheduling policy to randomize the task execution without violating the time constraints. Under our new scheduling policy, it is harder for attackers to deduct the task execution sequence. The information leakage will be measured by the schedule entropy.

**A. Definition**

Before we describe the problem, we must first define the variables to be used later:

- \(m_{i,t}\) indicates the length of the executed part of a job \(\tau_{i,j}\) before the time instant \(t\).
- \(B_{i,t}\) is the semi-inversion time budget for job \(\tau_{i,j}\), which refers to the time slots left for executing the job \(\tau_{i,j}\).
- \(p_{i,t}\) is the probability assigned to the job \(\tau_{i,j}\) at time \(t\). Note that, the probability may vary at different time instants.
- \(C_{i,t}(p)\) is the cumulative distribution function (CDF) of the probability for \(\tau_{i,j}\) finishing its execution before the time instant \(t\).

The problem can be defined as follows: Given a job set \(J = \{\tau_{1,j}, \tau_{2,j}, \cdots, \tau_{n,j}\}\) in the ready queue at a time instant \(t\), all these jobs are ordered by their deadline in an ascending order. The following two conditions must hold at time instant \(t\):

\[
\tau_{i,t} = c_{i,j} - m_{i,t} \tag{5}
\]

\[
B_{i,t} = d_{i,j} - t - \sum_{k \in hP(i), \tau_{k,q} \in J} \left( \frac{d_{i,j} - d_{k,q}}{T_k} \times C_k + r_{k,t} \right) \tag{6}
\]

The assigned probabilities \(p_{i,t}\) to each job must guarantee that every job completes its execution on or before the deadline, while the total probability of all jobs at each instant cannot exceed 1:

\[
\forall t, C_{i,t}(p) \leq 1 \tag{7}
\]

\[
\forall t, \sum_{i=1}^{n} p_{i,t} = 1 \tag{8}
\]

Our goal is to maximize the schedule entropy in a hyper-period under the constraints (5)–(8):

\[
\max_{p_{i,t}} H
\]

**Example III.1.** A task set \(\tau = \{\tau_1, \tau_2\}\) are shown in Table 1. Assume all these tasks are released at time 0; \(T_i\) denotes the minimum inter-arrival time between successive releases; the relative deadline is equal to \(T_i\). For example, the parameters for \(\tau_{1,1}\) is \((0, 1, 3)\); the parameters for \(\tau_{1,2}\) is \((3, 1, 6)\), and so on.

<table>
<thead>
<tr>
<th>Task</th>
<th>WCET ((\tau_i))</th>
<th>Period ((\tau_i))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\tau_1)</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>(\tau_2)</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

The jobs’ execution sequence under earliest deadline first (EDF) scheduling algorithm is shown in Fig. 1 (Proc), where the execution sequence in \([0,6]\) is the same as the execution sequence in \([6,12]\). It is possible that the execution sequence can be derived with the execution information in one hyper-period. We want to generate a new scheduling algorithm which could randomize the task’s execution pattern (e.g., Proc_R in Figure 1). The line Proc_R shows one possible scenario under randomized scheduling, where at each time instant all active jobs are selected based on an online calculated probability. It is hard to identify execution patterns over various hyper-periods.

### IV. Roulette Wheel Scheduler

To solve the scheduling problem mentioned above, we propose a randomized scheduler called RWS to randomize the execution pattern. We introduce the idea of RWS and discuss potential improvement directions of this approach.

RWS works under a quantum/slice setting; i.e., timeline is sliced into mini-slots (slices) of length \(\Delta\). During
A straightforward approach is to assign sufficient slices to every job. The process of choosing sufficient slices for jobs can be treated as a combination problem. Assume $\Delta$ can be divisible by the $B_{i,t}$ and $r_{i,t}$ of every job $\tau_{i,j}$. For each job $\tau_{i,j}$ at time instant $t$, we randomly pick $r_{i,t}/\Delta$ time slices out of $B_{i,t}/\Delta$ time slices for executing job $\tau_{i,j}$, so the total number of combinations $S$ are:

$$S = \prod_{t=1}^{n} \binom{B_{i,t}/\Delta}{r_{i,t}/\Delta}$$

Since the scheduler will randomly pick one of the $S$ combinations (which can be exponentially huge), it is unlikely that the execution patterns from two hyper-periods are identical.

However, this method cannot guarantee the schedulability for aperiodic task set. For example, a job set $J = \{\tau_{1,1}, \tau_{2,1}, \tau_{3,1}\}$, where the parameters of $\{\tau_{1,1}, \tau_{2,1}, \tau_{3,1}\}$ are $(0, 1, 5, 3), (0, 2, 6), (3, 1, 5, 6)$ respectively, and $\Delta = 0.5$. One possible execution combination is: in $t = [1, 2, 5]$, $\tau_{1,1}$ is executing; in $t = [4, 6]$, $\tau_{2,1}$ is executing. There is not enough time slots for the aperiodic job $\tau_{3,1}$ to executing.

**Proposed future work.** The proposed RWS still has some weakness. In order to tackle these drawbacks, we will improve the current RWS to handle more kinds of task sets. Besides, we will provide an algorithm to calculate the possibilities for every task. Moreover, the experimental measurement and theoretical analysis will be conducted to quantify the non-regularity of the randomized scheduling pattern.

---

**References**


