Schedule-Based Side-Channel Attack in Fixed-Priority Real-time Systems

***Abstract*—Security failures in real-time embedded systems can have catastrophic effects and can lead to injury to (or even loss of life for) humans, damage to the system and also environmental fallouts. Until recently security was an afterthought in the design of such systems. Even less understood are *attack mechanisms* that target real-time systems.**

In this paper we present a novel attack model and algorithm to extract the exact schedules of real-time systems designed using fixed priority algorithms. The attack is demonstrated on a real hardware platform and shows a high success rate. The leaked schedules are then used to launch a side-channel attack against a specific victim task. Our algorithm is robust in the presence of some schedule randomization defenses as well as jitters.

I. Introduction

Real-time embedded systems are all around us. They find use in a variety of domains such as aircraft, automobiles, medical devices, space vehicles, industrial control systems and nuclear power plants to name just a few. Many of these systems also have *safety-critical* requirements so any problems that deter from the normal operation of such systems could result in damage to the system, the environment and even pose a threat to human safety. Traditionally, security has often been an afterthought in the design of real-time systems since they used custom hardware, software and protocols and were often not connected to the external world. These assumptions are increasingly being challenged due to the rise in the use of common-off-the-shelf (COTS) components in new real-time systems and the drive towards remote monitoring and control facilitated by the growth of the Internet. Due to the critical nature of such systems successful attacks could lead to problems more serious than just loss of data or availability. The explosion in growth of embedded devices, *e.g.,* the internet-of-things (IoT), smart meters, *etc.* just exacerbates the problem. While there exists some recent work on security for real-time systems [20]–[22], [30]–[32], there is not much focus on *attack mechanisms* for such systems. In this paper, we focus on this latter, important, aspect of security for real-time systems.

Due to the safety issues mentioned above, real-time systems are also designed with great care and significant engineering effort to operate in a *predictable* manner. For instance, *(a)* designers take great care to ensure that the constituent tasks in such systems execute at well determined points in time [17], *(b)* their interrupts are carefully managed [34], *(c)* the memory management is deterministic[1] [16], *(d)* the execution of code on the processor is also analyzed to great degree (both, at compile time as well as run-time) [28], *etc.*

This deterministic property can be a double edged sword. On the one hand, the predictable behavior often makes it easier for adversaries to gauge the behavior of the system with high precision. This can increase the success rates for certain attacks, *e.g.,* side-channel [14] or even covert-channel attacks [26]. Consider the system presented in Figure 1. This high-level UAV design includes a camera that can be used for surveillance. Ideally an attacker would like to leak the camera images that are captured as a result. If this is difficult (say, due to the use of encryption) then the next best thing would be to identify locations of high-interest targets. This can be achieved by observing the *cache usage* of the camera task (or even image encoding task). If the cache usage is high, then a higher resolution image has been captured. If low, it is an indication that the camera (the victim in this case) is operating in a lower resolution mode. This information (camera in higher resolution mode) coupled with location information (*e.g.,* GPS data) can be really valuable to an adversary[2]. Adversaries could also use the knowledge of the exact behavior to launch targeted attacks, *e.g.,* denial-of-service where low level system resources (*e.g.,* caches, system buses, *etc.*) are made unavailable by overloading them at critical points in time (say, when a high priority task is about to execute).

On the other hand, any deviations from the expected behavior is suspicious and easier to detect (when compared to general purpose systems) so attackers must operate within narrow operational parameters (*e.g.,* stringent timing and resource constraints) if they are to avoid detection. This is particularly true in the case of side-channel and covert-channel attacks. Often such attacks require *precise knowledge* of when the victim task is about to execute [3], [14], [18] so that the attacker has the highest chances for success. To this end, adversaries *probe* the system in a repeated manner. An increased probing frequency will also increase the chance of identifying the

[1]For instance, many such systems have no virtual memory management and may even turn off their data caches.

[2]We discuss this in more detail in Section II.

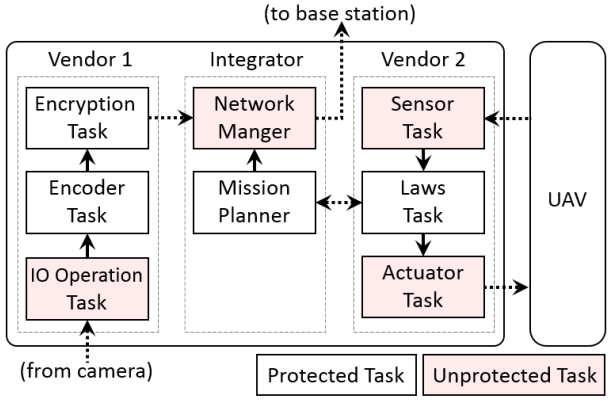
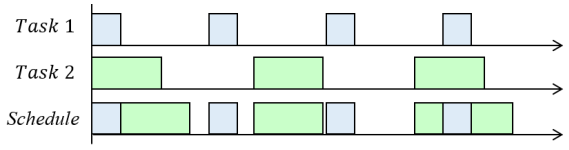
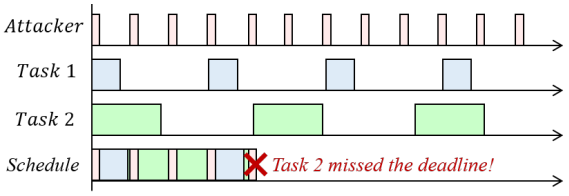


Figure 1: A high-level design of an unmanned aerial vehicle platform(UAV).

victim's execution points. In a real-time system, this probing mechanism might be easily detected. If attackers perturb the system too much then *(i)* they will be detected right away since other tasks will miss their constraints (often represented as timing constraints called *deadlines*) and *(ii)* the attacks will not succeed since the system's operation has been impeded[3](for instance, in the camera example, the adversary might want to track *all* the points where the camera was operated in a higher resolution mode). Figure 2a shows the normal execution of two real-time tasks that meet their constraints (deadlines). Figure 2b depicts the situation when a "probing task" (labeled as "Attacker" in the Figure) is introduced into the system. The increase in system utilization (*i.e.,* the extra execution) due to the probe results in a regular real-time task missing its deadline, thus leading to detection and also putting the operational safety of the entire system at risk.



(a) Without attacker's task.



(b) With attacker's task involved.

Figure 2: An example of schedules with or without attacker's task: (a) shows the normal schedule when no attacker is involved and (b) demonstrates that a thoughtlessly planned attack may lead to malfunction of the existing tasks.

Hence, a successful attack on real-time systems necessitates the following *(a)* the ability to *accurately reconstruct* the behavior of the system which, in this context, should be the points in time when one or more victim tasks execute and *(b) avoid detection* during the attack process by not changing the system behavior. We demonstrate methods that achieve both of the above. We present algorithms to *reconstruct (and hence,leak) the precise schedule* of the system (Section III) and avoiding detection by *hijacking the idle task* (Sections II and V)[4]. Specifically, we focus on hard real-time systems designed around *fixed priority algorithms* [7], [17], [19] since these are the most common class of real-time scheduling algorithms found in practice today [19]. Also, this class of scheduling algorithms is most vulnerable to such attacks since they have the most stringent constraints (hence they are very predictable).

We evaluate our approach using an *actual implementation* (Section V) of the attack on a hardware board. The board (an ARM-based Zedboard) implements the aforementioned UAV model [22] on a real-time operating system that also presents the vendor-based development model (Section II) that, in fact, can increase the chances for attackers to insert code due to the chaos inherent in such processes. We also carry out exhaustive simulations to understand the design space for such algorithms.

In summary, the main contributions of this paper are,

1) an attack scheme aimed at *leaking the schedule* of fixed priority hard real-time systems – this involves development of algorithms capable of *reconstructing* the schedule of the system;
2) implement and evaluate the attack scheme (and necessary algorithms) using a realistic embedded platform and exhaustive simulations and
3) demonstrate one use case where a leaked schedule can be used to launch other attacks (*e.g.,* a cache-based side-channel attack) (Section IV).

II. System and Adversary Model

A. System Model

In this paper, a hard real-time system with fixed-priority scheduling (e.g., Rate Monotonic (RM) algorithm [17]) is considered. Such system contains a task set $\Gamma = \{\tau_1, ..., \tau_n\}$ consisting of n periodic hard real-time tasks. Each task τ_i, $1 \leq i \leq n$, is characterized by p_i, c_i, d_i and $pri(\tau_i)$, as shown in Figure 3, in which p_i is the period, c_i specifies the worst execution time, d_i shows the deadline and $pri(\tau_i)$ indicates the priority of the task τ_i. Note that all tasks in Γ are indexed by descending priorities and priorities are distinct, i.e., $pri(\tau_i) > pri(\tau_{i+1})$, and every task has a diverse period.

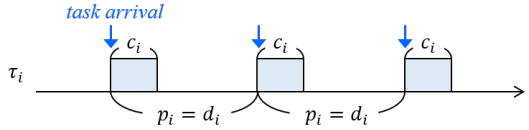


Figure 3: A real-time task τ_i is characterized by p_i, c_i, d_i and $pri(\tau_i)$ which represent the period, execution time, deadline and priority of task τ_i.

Additionally, let $HP(\tau_i)$, $HP(\tau_i) \subset \Gamma$, denote the task set that contains tasks with priorities higher than τ_i. Similarly, $LP(\tau_i)$, $LP(\tau_i) \subset \Gamma$, represents the task set that has tasks with lower priorities than τ_i. Besides, we define the notation $\Gamma^{(i,n)} = \{\tau_i, \tau_{i+1}, ..., \tau_n\}$ to be a subset of the task set Γ. Thus, according to this definition, $\Gamma^{(1,n)}$ has equivalent task set as Γ. For ease of illustration of the proposed analysis algorithm in later sections, the schedule of the specified hard real-time system is modeled by a set of intervals denoted by V. Furthermore, V consists of two subsets: *(i)* a subset of busy intervals [6], defined as $W = \{\omega_1, ..., \omega_m\}$ which contains m busy intervals, and *(ii)* a subset of idle intervals, $ID = V - W$ as shown in Figure 4. Each busy interval ω_k, $1 \leq k \leq m$, comprises zero or more jobs of τ_i, $1 \leq i \leq n$, that are scheduled, execute and complete in this interval. Let $N_k(\tau_i)$ (≥ 0), denote the number of jobs of τ_i that are enclosed in ω_k, thus the duration of the busy interval can be computed as:

$$C(\omega_k) = \sum_{i=1}^{n} (N_k(\tau_i) \cdot c_i) \quad (1)$$

To depict the schedule of a busy interval, let S_k be an array set and $S_k(\tau_i)$ represent an array storing the start times for each job of τ_i in ω_k. For example, if τ_i has $N_k(\tau_i) = 2$

[3]Assuming, of course, that the attacker's motivation was to steal information undetected.

[4]We *do not* change the system utilization by hijacking the idle task – we only consume the slack that was already present in the system. We discuss this in detail in the following sections.

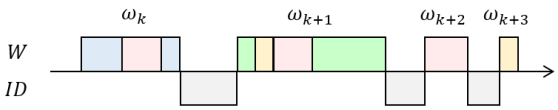


Figure 4: An example of a schedule which is modeled by busy intervals W and idle intervals ID. In this example, there are four distinct busy intervals $\{\omega_k, ..., \omega_{k+3}\}$. In each busy interval, a color block represents an instance of a task.

jobs in ω_k, then $S_k(\tau_i)$ contains two start times corresponding to the two jobs. Let $T_s(\omega_k)$ be the start time of ω_k. Then it is sufficient to depict ω_k with the 3-tuple $\{T_s(\omega_k), C(\omega_k), S_k\}$ where $T_s(\omega_k)$ is the start time of ω_k, $C(\omega_k)$ is it's length and S_k specifies the start times of all enclosed jobs.

Note that the start time of a job is used to capture the schedule in a busy interval here instead of its arrival time because our focus is more on the output of the scheduler in this paper. The arrival time pinpoints the instant when a job is ready to be scheduled and being put in the ready queue but not the instant when a job is truly being executed. Simulation of the scheduling algorithm is needed to obtain the latter (start time) from the former (arrival time). However, it's worth mentioning that, in our analysis algorithm, it is necessary to infer arrival times before the start times can be computed. Therefore, we define $A_k(\tau_i)$ as an array that indicates the arrival time for each job of τ_i in ω_k. Figure 5 shows an illustration of the introduced real-time system model, while the notation used is summarized in Table I.

Table I: Glossary of real-time system notations.

Symbol	Definition
Γ	a set of n real-time tasks $\{\tau_1, ..., \tau_n\}$
τ_i	a real-time task
p_i	period of τ_i
d_i	deadline of τ_i
c_i	execution time (computation time) of τ_i
$pri(\tau_i)$	priority of τ_i, smaller value represents higher priority
V	an interval set representing schedules, $V = ID + W$
ID	idle interval set
W	busy interval set $\{\omega_1, ..., \omega_m\}$
ω_k	a busy interval
$T_s(\omega_k)$	start time of ω_k
$C(\omega_k)$	duration of ω_k
$N_k(\tau_i)$	the number of jobs of τ_i enclosed in ω_k
$S_k(\tau_i)$	an array storing the start times for each job of τ_i in ω_k
$A_k(\tau_i)$	an array storing arrival times for each job of τ_i in ω_k

1) Example System: Avionics Demonstrator: In order to motivate and evaluate the presented research, we use the example of the Electronic Control Unit (ECU) for an avionics system depicted in Figure 1. This demonstrative system, runs many of the same types of tasks which could be expected to run on an Unmanned Aerial Vehicle (UAV) surveillance system. The ECU communicates locally with the inertial sensors, GPS localization system and actuators ("UAV" in the figure), as well as a camera subsystem. The ECU also uses off-board communication to exchange information with a base station. We assume that three parties are involved in building the ECU system, Vendor 1, Vendor 2, and the Integrator. Each party is responsible for a different ECU subsystem, which in turn comprises different real-time tasks.

Vendor 1 is responsible for the *image subsystem*. The I/O Operation Task mimics the behavior of a camera driver. The Encoder Task is realized as a JPEG compressor. Encryption

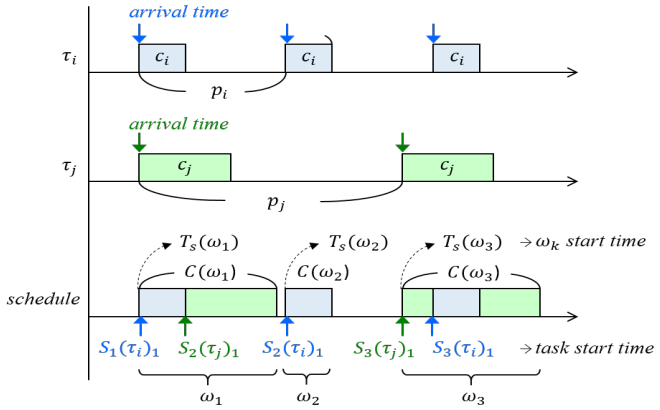


Figure 5: An illustration of the real-time system model used in this paper. Downward arrows represent arrival times of periodic jobs and upward arrows indicate the start times of corresponding jobs. Note that $pri(\tau_i) > pri(\tau_j)$ in this case.

Task uses the AES cipher using a protected, secret key. The encrypted image is then passed to the network manager in the Integrator subsystem. *Vendor 2* is responsible for the *control subsystem*. The Sensor Task receives and parses incoming sensor data for the other tasks. The Laws Task computes the control output to move the UAV towards a way-point determined by the Mission Planner in the Integrator subsystem. Finally, the Actuator Task prepares the actual output commands and send them to physical actuators. Finally, the *Integrator* is responsible for connecting the two previous subsystems together and performing mission control. The Mission Planner Task communicates with the Laws Task to determine the current position of the UAV and move it between a set of fixed way-points. Network Manager sends encrypted data from Mission Planner and Encryption Task to the base station.

B. Adversary Model

Reconnaissance is often the first step in attacking a system and it is in the interest of the attacker to stay undetected during this time to, *(i)* both collect necessary information to enable his attacks and *(ii)* to not alert system operators whose defensive actions might make attacking the system more difficult. In real-time systems having knowledge about system operation is even more important than in enterprise settings as unsuccessful attacks can more easily be detected due to the deterministic and predictable nature of the system. In the now famous Stuxnet attack [8], it is believed that the malware remained undetected for months before its eventual discovery. Similarly, the goal of the adversary in our model is to steal information undetected rather than to disable or disrupt the real-time system.

Further, we also assume that the adversary has a foothold on the real-time system. That is, he is able to run one or more tasks in the system. This could be achieved in many different ways. For example, taking into account the multi-vendor development model used for many complex real-time systems, an adversary could gain access through a compromised software supply-chain of one of the vendors. Since it is difficult for the system integrator to inspect every detail of the written code from participating vendors, it leaves room for unknown flaws in vendor's side for adversaries to exploit.

We also assume that the adversary has knowledge of the

number of tasks, their periods and execution times. The adversary may obtain this information through social engineering or in real-time from the system as this information is available to the task scheduler. The attacker's goal then is to leverage this information to steal information from or about the system's operation without being detected. Specifically, our focus will be on reconstructing the system's task schedule so targeted stealthy attacks can be launched on tasks of interest.

C. Attack Model

Task Priority Correlations: In a real-time system that uses Rate-Monotonic scheduling algorithm, each task is prioritized based on it's period instead of its purpose or other attributes such as security level. Thus, it is possible that a low security task with a short period is assigned a relatively higher priority. The preempting capability of a high priority task gives an attacker opportunities to launch a variety of attacks, *e.g.,* cache-timing based side-channel attack, against tasks being preempted. On the other hand, although tasks with lower priorities are seemingly valueless, the nature of being preempted makes them exceptionally useful in monitoring the system behavior, for example, measuring the execution time of the task that preempts it. Hence, each priority level is actually beneficial to attackers in different ways. Intuitively, compromising the highest and lowest priority tasks will provide the attacker opportunities to *(i)* launch attacks on any other task, and *(ii)* to monitor and learn the execution times of all other tasks.

However, when there are multiple tasks that have priorities lower than the compromised task, it is hard for the adversary to identify which task it is preempting as well as to isolate valuable information regarding a specific task from the captured data. Similarly, when there are multiple tasks with higher priorities than the compromised task, since more than one task can be scheduled simultaneously and preempt it, information that the compromised task gains can actually be a composition of information due to multiple tasks making it difficult to directly use it without further analysis.

Idle Task Manipulation: In order to monitor all tasks at all times, we propose to hijack the idle task instead of the lowest priority task. Idle task, by definition, is the function that the scheduler calls when there is no active task or scheduled task waiting in ready queue. The way that the idle task is implemented in practice varies from system to system. Different from generic computers where systems either enter sleep mode or lower the CPU frequency when idling, real-time systems that require predictability and timeliness tend to keep CPUs up and consume unused slack time. Take FreeRTOS as an example, it maintains an idle task which contains nothing but an infinite while loop to exhaust the idle time.

Therefore, one can say that the idle task is logically the lowest priority task in a real-time system. It inherits all potentials from the legitimate lowest priority task h_1, yet it has no limit in period, i.e., idle task has an infinite deadline. Thus the idle task can actually monitor the execution state of a system rather than any specific task. What the idle task can capture are busy intervals, W as defined earlier in SectionII-A, that are composed of arbitrary subsets of tasks in Γ. Since RM scheduling is static, the compositions of busy intervals as well as the schedule should be predictable and deterministic.

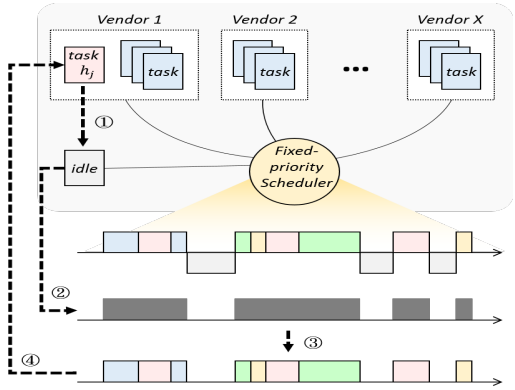


Figure 6: Attack route of the schedule-based attack scheme.

In this work we propose a schedule reconstruction algorithm that can analyze captured busy intervals along with the system profile (the number of tasks, periods and execution times) that is available to the adversary.

Attack Scheme: The goal of the adversary is to locate the execution (job) of a selected task and launch information stealing attacks on it without being detected. Our proposed attack utilizes side channels to collect system data (i.e., busy intervals) and reconstruct the system schedule which can be used to predict the next start time of a selected task.

As shown in Figure 6, everything starts with the compromised task h_j which the adversary owns initially. The task h_j maintains a state machine that injects a malicious function into idle task in run time. Once the injection is complete, task h_j switches back to normal state and continues it's original and legitimate function. Hijacked idle task implements another state machine that has three stages: *(i)* capturing busy intervals, *(ii)* analyzing busy intervals (or inferring context of busy intervals) and *(iii)* triggering attacks.

In the next section, we will elaborate on the algorithm implemented in the hijacked idle task to capture busy intervals and reconstruct the schedule.

III. Reconstruction of Schedules

As mentioned earlier, our friendly neighborhood attacker's main objective is to capture (or reconstruct if the observations are not of sufficient quality) *the exact schedule* of real-time tasks that execute in the system. The final goal could be to launch an attack on one more victim tasks to gather information (via side-channel attacks as seen in Section V-B) or perhaps reduce the availability of low level resources during the execution of critical tasks (or any number of other objectives such as setting up a covert channel). In any case, reconstructing (and thus, leaking) the schedule of the system is a first, crucial, step. This section introduces *a novel algorithm that uses the predictable, periodic, nature of hard real-time systems to carry out such a reconstruction*. The output of the algorithm is the list of start times of the tasks in the system (that can help locate the precise execution intervals of victim tasks at run-time).

We now present the reconstruction algorithm that we name "ScheduLeak"[5]. The sequence of steps is summarized in Figure 7. The high-level components of ScheduLeak are:

[5]Since its main objective is to "leak" the schedule of the system).

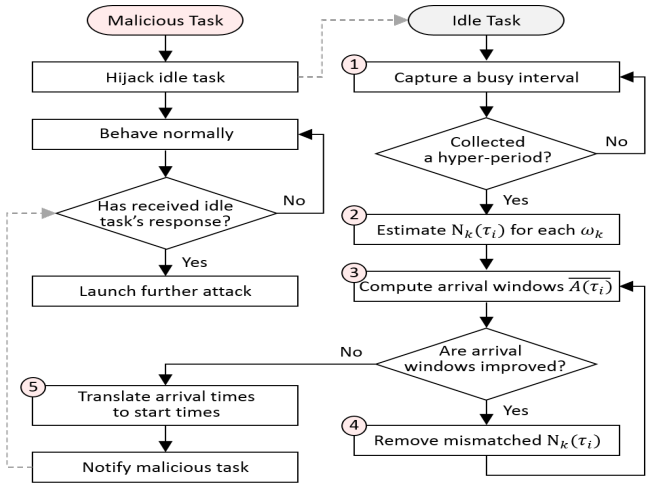


Figure 7: Attack flow of the proposed scheme between malicious task and idle task.

[Step 1:] **Capturing Busy Intervals:** The first step is to capture busy intervals during the execution of the system. The observer task[6] infers the length of busy intervals by noting the points in time when it[7] was scheduled (further details are provided later in the paper). This gives us a set of busy intervals, $W = \{\omega_1, ..., \omega_m\}$ where $T_s(\omega_k)$ is the start time of each interval, ω_k, and $C(\omega_k)$ is its duration. [Section III-A]

[Step 2:] **Estimating $N_k(\tau_i)$:** Using the busy interval set, W, and the system profile (period and execution time of each task) we next estimate the number of tasks involved in each busy interval. That is, finding $N_k(\tau_i)$ for each task τ_i in each busy interval ω_k. [Section III-B]

[Step 3:] **Inferring Arrival Windows** $\overline{A(\tau_i)}$ for each task τ_i: The estimated $N_k(\tau_i)$ values are used to calculate possible arrival windows in ω_k for each τ_i. [Section III-C.]

[Step 4:] **Eliminating Mismatched $N_k(\tau_i)$ Estimates:** In some cases, $\overline{A(\tau_i)}$ for a task τ_i may include multiple uncertain inferences. In order to eliminate this ambiguity, current arrival windows are used to validate the estimated $N_k(\tau_i)$ values from Step 1. The updated $N_k(\tau_i)$ value is used to further narrow down the arrival window. [Section III-D]

[Step 5:] **Reconstructing Schedules:** The arrival window, $\overline{A(\tau_i)}$, of τ_i is then converted to a *specific arrival time point*, $A(\tau_i)$. With the exact $A(\tau_i)$'s available for all busy intervals we can generate the start times of corresponding jobs, say, by using a fixed priority scheduling simulator iterating over the busy interval, ω_k. If necessary, iterating over every busy interval to get $S_k, 1 \leq k \leq m$, we can *reconstruct complete schedule* that can now be used to launch further attacks (see Section IV). [Sections III-E and III-F]

A. Capturing Busy Intervals

Recall from the system model section, a schedule can be represented by $V = ID \cup W$. By getting busy intervals W one can potentially reconstruct the whole intervals V, since they are like a shadow of the schedule. Also, due to the determinism in a fixed-priority hard real-time system the schedule of a given task set is predictable. As a result, schedule repeats every hyper-period. The hyper-period of task set $\Gamma = \{\tau_1, ..., \tau_n\}$ is: *Hyper Period* $= lcm(p_1, ..., p_n)$. Hence, the number of *unique* busy intervals in the schedule is limited. Busy intervals within one hyper-period will suffice for reconstructing schedules of hard real-time systems.

To capture a busy interval, *the idle task checks whether itself has been preempted* (details in later sections). When a preemption is detected, a busy interval, ω_k, is found. The duration between preemptions is $C(\omega_k)$ and the start time of this busy interval is to $T_s(\omega_k)$. The detailed mechanism for capturing a busy interval is presented in Algorithm 3.

The process of capturing one busy interval repeats until the idle task collects all busy intervals within one hyper-period. Eventually, we get a busy interval set $W = \{\omega_1, ..., \omega_m\}$ for one hyper-period. Each busy interval ω_k contains its start time $T_s(\omega_k)$ and measured duration $C(\omega_k)$. W is then passed to the next step of the algorithm.

B. Estimate of $N_k(\tau_i)$

The goal of this step is to estimate the number of arrivals of each task in each of the busy intervals captured in the previous step, i.e., finding $N_k(\tau_i), 1 \leq k \leq m, 1 \leq i \leq n$.

Considering the busy interval ω_k, the duration can be calculated by Equation (1). For any task τ_i, it may either contribute nothing or contribute one or more jobs to ω_k (*i.e.,* $0 \leq N_k(\tau_i)$). But, given the duration $C(\omega_k)$ of the busy interval, the number of jobs for a task τ_i depends on its period and execution time. Therefore, knowing the period and the execution time of every task, we can reduce the number of possibilities for the value of $N_k(\tau_i)$. The estimation of the possible number of jobs for τ_i in busy interval w_k is done using the following theorem. This theorem gives the exact value of $N_k(\tau_i)$ or in the worst case reduces the possible candidates for $N_k(\tau_i)$ to only 2 values.

Theorem 1. For given values of $p_i, c_i, C(\omega_k)$:
(i) If $C(\omega_k)$ satisfies

$$(N \cdot p_i - c_i)^+ \leq C(\omega_k) < N \cdot p_i + c_i, \tag{2}$$

then busy interval ω_k can only contain N jobs for the task τ_i.

(ii) If $C(\omega_k)$ satisfies

$$N \cdot p_i + c_i \leq C(\omega_k) < (N+1) \cdot p_i - c_i, \tag{3}$$

then busy interval ω_k can only contain N or $N+1$ jobs for the task τ_i.

A formal proof of Theorem 1 is provided in Appendix A. The proof relies on the fact that tasks are periodic. The intuition behind the proof is as follows: We first note that a busy interval contains a task if and only if it contains the arrival time of that task, and then we consider the extreme cases of the duration of a busy interval that can contain N arrivals of a task. Note that Theorem 1 does not take into account the impact of jitters, or uncertainty in the values of p_i and c_i. An extended version of the theorem that considers those aspects is provided in Appendix C.

[6]While in our system the observer is the Idle task, this need not be the case in all such attacks. In this paper we will use "observer task" and "idle task" interchangeably and make the distinction when it is necessary.

[7]The observer task itself.

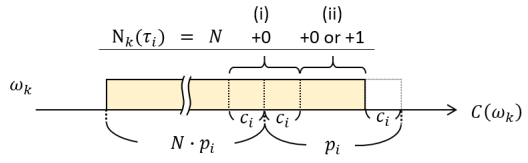


Figure 8: Condition checks for estimating $N_k(\tau_i)$. The axis indicates the busy interval's duration which is a positive number. Based on this duration, we can estimate $N_k(\tau_i)$. The durations for which the estimation is N+0 and N+(0 or 1) is illustrated in the figure.

The conditions of Theorem 1 are depicted in Figure 8. Note that the number of arrivals for each task is estimated separately and independently, and that some tasks may have two candidate values for each interval: N and $N+1$. This ambiguity can only be eliminated when all tasks are considered together with the duration of the specified busy interval. Recall from Equation (1), that the duration of a busy interval is sum of the number of jobs for each task times its execution time. Therefore, in order to sort out the correct combination of $N_k = \{N_k(\tau_1), ..., N_k(\tau_n)\}$ where $N_k(\tau_i)$ is either N or $N+1$, we compute Equation (1) with all possible combinations of $N_k(\tau_i)$. In the worst case that all $N_k(\tau_i)$'s have two possible values and all tasks are in a busy interval, there will be 2^n possible combinations. Eventually, only those combinations that satisfy Equation (1) are shortlisted for further consideration.

The following two examples illustrate the two possibilities: one leads to a unique inference of N_k, and the other results in multiple feasible N_k inferences. Assume that the idle task has captured for one hyper-period (i.e., $LCM(5, 6, 10) = 30$) and records following 4 busy intervals:

Example 1. Consider a task set $\Gamma = \{\tau_1, \tau_2, \tau_3\}$ as follows:

	p_i	c_i
τ_1	5	1
τ_2	6	2
τ_3	10	2

Busy Intervals	Duration	Timestamp
ω_1	8	[0,8]
ω_2	6	[10,16]
ω_3	5	[18,23]
ω_4	4	[24,27]

Taking busy interval ω_1 as an example, by applying Theorem 1 to ω_1 as shown in Figure 9, we can compute $N_1(\tau_1)$, $N_1(\tau_2)$ and $N_1(\tau_3)$ values as follows:

	p_i	c_i	$N_k(\tau_i)$
τ_1	5	1	1 or 2
τ_2	6	2	1 or 2
τ_3	10	2	1

*$C(\omega_1) = 8$

We can now use Equation 1 to find possible N_1 combinations that can lead to the given busy interval duration $C(\omega_1) = 8$ as shown in the following table:

$N_1(\tau_1)$	$N_1(\tau_2)$	$N_1(\tau_3)$	$C(\omega_1)$	
1	1	0	3	
1	2	1	7	
2	1	1	6	
2	2	1	8	√ matched

*$C(\omega_1) = 8$

In this case, only the combination of $N_k = \{2, 2, 1\}$ satisfies Equation 1 for a busy interval length of $C(\omega_1) = 8$.

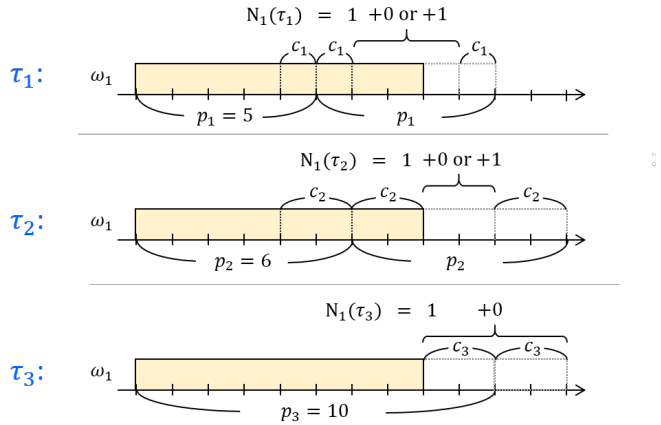


Figure 9: Example 1, computing $N_k(\tau_i)$ values for ω_1 based on Theorem 1.

Example 2. This example demonstrates that multiple task combinations could potentially result in a busy interval of a given length. Here we use the same task set from Example 1 but consider a busy interval ω_4 with duration $C(\omega_4) = 3$. The $N_k(\tau_i)$ values are computed as follows:

	p_i	c_i	$N_k(\tau_i)$
τ_1	5	1	0 or 1
τ_2	6	2	0 or 1
τ_3	10	2	0 or 1

*$C(\omega_4) = 3$

Because every task has two candidate values, there will be at most $2^3 = 8$ possible combinations to be verified using equation (1) as shown below:

$N_4(\tau_1)$	$N_4(\tau_2)$	$N_4(\tau_3)$	$C(\omega_4)$	
0	0	0	0	
0	0	1	2	
0	1	0	2	
0	1	1	4	
1	0	0	1	
1	0	1	3	√ matched
1	1	0	3	√ matched
1	1	1	5	

*$C(\omega_4) = 3$

The result in the table shows that even though most of the task combinations are eliminated, there are still two feasible combinations, $N_4 = \{1, 0, 1\}$ and $N_4 = \{1, 1, 0\}$, that can lead to a busy interval length of $C(\omega_4) = 3$. In this case, both inferences will be retained for further processing.

C. Inference of Arrival Windows

The previous step only infers potential combinations of tasks for a given busy interval but not their ordering within the interval. In order to reconstruct the schedule, it is important to infer the order of the tasks which depends on their start times. However, inferring the start times of tasks is not intuitive since a task may be postponed or preempted by the scheduling algorithm based on its priority. Therefore, it is more reasonable to infer the arrival times first and then estimate the start times based on the arrival times. In order to estimate an arrival time, we compute the possible window for that arrival, which we call arrival window. Eventually we want to obtain a narrow arrival window for each task in this step.

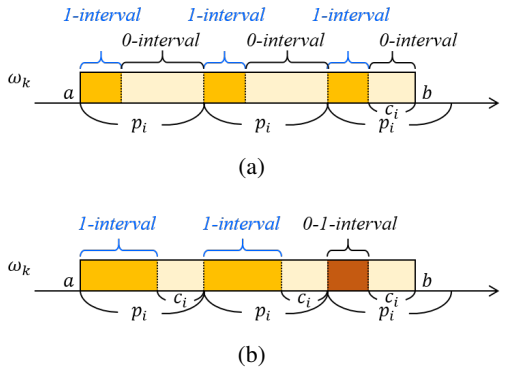


Figure 10: Estimate of arrival time locations.

In order to identify potential arrival windows for a task, we partition the busy interval $\omega_k = [a, b]$ into the following three types of segments for each task τ_i:
– *0-interval*: The segments in which we know there is no arrival.
– *1-interval*: The segments in which we know there exists exactly 1 arrival.
– *0-1-interval*: The segments in which there may exist 0 or 1 arrival.

The partitioning of the busy interval is done using:

Theorem 2. Considering a task τ_i and a busy interval w_k that has start time a and end time equal to b.
The partitioning of the busy interval is done by using the following equations:
(i) If τ_i has arrived exactly N times, then the for $1 \leq j \leq N$ following segments are *1-interval*:

$$\overline{A_k(\tau_i)_j} = [a + (j-1)p_i, \\ min\{b - (N-j)p_i - c_i, a + jp_i - c_i\}]$$

(ii) If τ_i may have arrived either N or $N+1$ times:
the following segments are *1-interval*:

$$\overline{A_k(\tau_i)_j} = [a + (j-1)p_i, a + jp_i - c_i] \qquad 1 \leq j \leq N$$

and the following segments are *0-1-interval*:

$$\overline{A_k(\tau_i)_j} = [a + (j-1)p_i, b - c_i] \qquad j = N+1$$

where $\overline{A_k(\tau_i)_j}$ is the j^{th} arrival window for task τ_i in busy interval ω_k.
In both cases, the remainder of the busy interval is *0-interval*.

See Appendix D for the proof.

The main tool used in this theorem is the periodicity of the tasks. Specifically, we use the fact that if a task τ_i cannot arrive at a given time t, then it also cannot arrive at times $t \pm p_i$. We find time instances where the task cannot arrive and we have considered them as 0-intervals. Figure 10 shows the use of Theorem 2. Part (a) depicts a case in which $N_k(\tau_i) = 3$ and $b - (3-j)p_i - c_i < a + jp_i - c_i$, for $j = 1, 2, 3$. Note the obtained 1-intervals for τ_i in a busy interval should be repeated every p_i. Part (b) depicts a case in which $N_k(\tau_i) = 2 or 3$ and we are not able to determine whether the last interval contains an arrival or not; hence, it will be a 0-1-interval.

Next, we will use the intervals obtained above to find arrival windows. Because of the periodicity assumption, a task must arrive exactly one time in each time interval of length equal to its period. Moreover, without considering jitters, the relative arrival time in each period should be consistent. If we divide one hyper-period into intervals of length p_i, we can find the arrival window of τ_i by getting intersections of the *1-interval* and *0-1-interval* segments. The taken intersections should satisfy the following three rules: *(i)* in each segment of length p_i there should be exactly one arrival; *(ii)* in each 1-interval there should be exactly one arrival; *(iii)* in each 0-1-interval there should be at most one arrival.

Figure 11 shows different possible cases in taking intersections. Figure 11(a) shows the case that we have successfully found a single intersection interval that satisfies the above rules. Since the starting point of a task's period is unknown in this step, it is likely that a *1-interval* and *0-1-interval* segment is fragmented by two separate sections, which may lead to broken intersections as shown Figure 11(b). However, if we restore the time line from the layered ones, adjacent intersections form one continuous interval. In this case, the two intersections can be united to be one arrival time window.

Also, in some cases, there may exist intermittent intersections for a task's arrival time window as depicted in Figure 11(c). This situation is caused by the ambiguity induced from *0-1-interval* segments. Note that having *0-1-interval* segments is a necessary condition for ambiguity but not a sufficient condition. The next step is to estimate the true arrival window and eliminate the uncertainty posed by such cases.

Example 3. Consider the same task set from Example 1, here we take task τ_3 as an example to demonstrate how the arrival window $\overline{A(\tau_3)}$ is derived. By following Theorem 2, four busy intervals can be partitioned by *1-interval*, *0-1-interval* and *0-interval* as shown in Figure 12.

Once *1-interval* and *0-1-interval* are obtained, we can get the intersections as shown in Figure 13(a). In this case, there are two intersections which represents two possible arrival windows $\overline{A(\tau_3)} = [0, 1]$ or $\overline{A(\tau_3)} = [4, 4]$ for task τ_3. Similarly, arrival windows for τ_1 and τ_2 can be computed as $\overline{A(\tau_1)} = [0, 0]$ and $\overline{A(\tau_2)} = [0, 0]$, respectively. Similarly, the arrival window for τ_2 is shown in Figure 13(b) .

D. Eliminating Mismatched N_k Inferences

As illustrated in Figure 11(c), the ambiguity in arrival windows is caused by *0-1-interval* segments in each arrival window. For the case in Figure 11(c), if one of the *0-1-interval* segments can be clarified as *1-interval*, then we know that is the correct arrival window. Since estimation of arrival windows relies on *1-interval* and *0-1-interval* segment that are computed based on $N_k(\tau_i)$ values, it is vital to eliminate ambiguity from inferred $N_k(\tau_i)$ values. This can be done by applying the arrival window $\overline{A(\tau_i)}$ to each busy interval to validate the estimated $N_k(\tau_i)$ values.

Note that the *0-1-interval* of tasks are not independent. For instance, in Example 3, if we knew the correct value of $N_k(\tau_2)$ and thus did not have a *0-1-interval* segment for task τ_2, the *0-1-interval* of task τ_3 will also be removed. Thus,

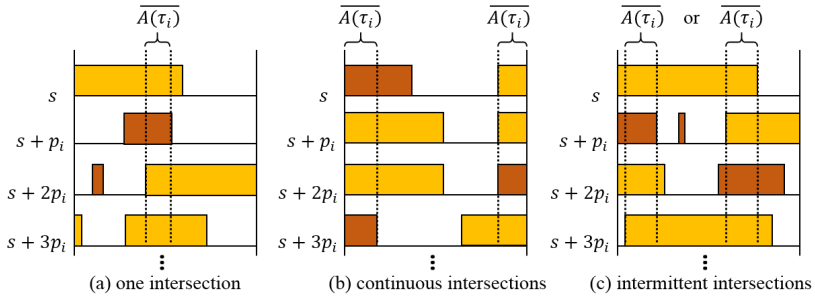


Figure 11: Examples of getting intersections for the arrival window $\overline{A(\tau_i)}$.

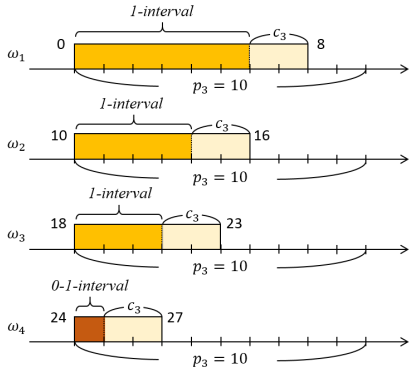


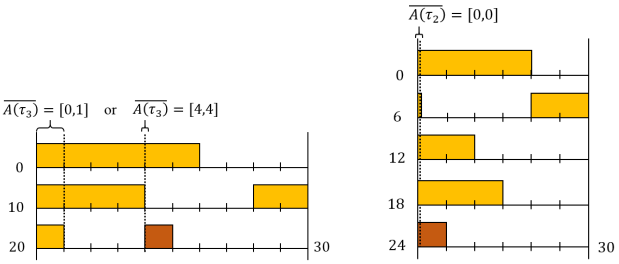
Figure 12: Example 3, partition busy intervals $\{\omega_1, ...\omega_4\}$ for τ_3 based on Theorem 2.

reduced ambiguity in one task can lead to reduced ambiguity in multiple other tasks.

Figure 14 presents the process of removal of mismatched $N_k(\tau_i)$ to a busy interval ω_k with ambiguous $N_k(\tau_i)$ values by applying inferred arrival windows $\overline{A(\tau_i)}$. This figure considers the simple case where the arrival window is a continuous interval (shown by the black rectangle in the figure). That is, the result of the process of getting intersections explained above is the black rectangles, which as expected is repeating every p_i seconds. Part (a) of this figure shows the case that the arrival window has overlap with a 0-1-interval, which implies that the 0-1-interval is in fact a 1-interval (Therefore, the inference (originally N or $N+1$) becomes $N+1$). Part (b) of this figure shows the case that the arrival window does not have overlap with a 0-1-interval, which implies that the 0-1-interval is in fact a 0-interval. Therefore, $N+1$ is removed from the inference, which leaves N as the only possible value.

Algorithm 1 shows the process iterating over every busy interval to validate $N_k(\tau_i)$ values. By removing $N_k(\tau_i)$ values that do not match with inferred arrival windows in a busy interval, the number of possible task combinations for that interval can be reduced, sometimes to a unique combination. This reduced set of possible $N_k(\tau_i)$ values is then used to update the arrival windows. This process of removing mismatched $N_k(\tau_i)$ inferences and updating arrival windows repeats until the values stabilize as illustrated in Figure 7.

Example 4. Consider busy interval ω_4 which has two possible inferences ($N_4 = \{1, 0, 1\}$ and $N_4 = \{1, 1, 0\}$) in Example 2, we apply arrival windows got from previous step to remove



(a) Arrival windows of τ_3 (b) Arrival windows of τ_2

Figure 13: Example 3, compute possible arrival windows by getting intersections from *1-interval* and *0-1-interval*.

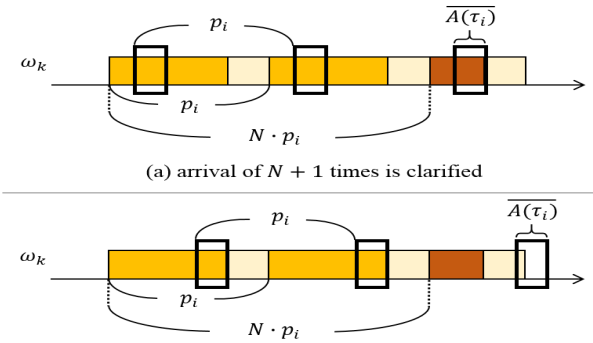


Figure 14: Validation of $N_k(\tau_i)$ values with applying arrival window $\overline{A(\tau_i)}$ to the busy interval ω_k.

Algorithm 1 Removing Mismatched N_k Inferences

```
while any N_k inference is updated do
    compute A(τ_i) for all tasks
    for each busy interval ω_k do
        for each arrival window A(τ_i) do
            remove this N_k if N_k(τ_i) is mismatched
        end for
    end for
end while
```

the mismatched inference.

Here we use arrival window of τ_2 to demonstrate the process since it has two possible $N_4(\tau_2)$ values (0 or 1) and an unambiguous arrival window $\overline{A(\tau_2)} = [0, 0]$ which is perfect for validating N_4 combinations. By applying $\overline{A(\tau_2)}$ to ω_4 as shown in Figure 15, we confirm that τ_2 should have arrived one time, which validates the inference of $N_4 = \{1, 1, 0\}$ and invalidates $N_4 = \{1, 0, 1\}$.

Once the ambiguity in ω_4 is removed, the only *0-1-interval* for inferring $\overline{A(\tau_3)}$ in Example 3 can be identified as *0-interval* based on the correct inference $N_4 = \{1, 1, 0\}$. This leaves the intersection on the left in Figure 13 as the only correct arrival window for τ_3.

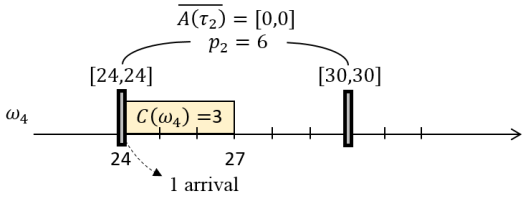


Figure 15: Example 4, apply arrival window $\overline{A(\tau_2)}$ to busy interval ω_4 we can say τ_4 has arrived exactly one time, which removes the original ambiguity in ω_4 shown in Example 2

E. Arrival Windows to Arrival Times

At the end of the refinement loop hitherto discussed, some tasks end up with an arrival window as small as one point giving their exact arrival time. However, arrival windows for other tasks may still have a broader range range and cannot be narrowed further. In order to process the scheduling simulator that will be used in next step, an exact point for each arrival is needed. We consider the *beginning point of the arrival window* as the exact arrival time for such tasks. The reasons for this choice are first, to make sure that the job instances in a busy interval do not become disconnected, and second, since this choice indicates the earliest possible arrival time of a job, attacks launched by the colluding task using this arrival time will never miss the job.

F. Reconstruction of Schedules

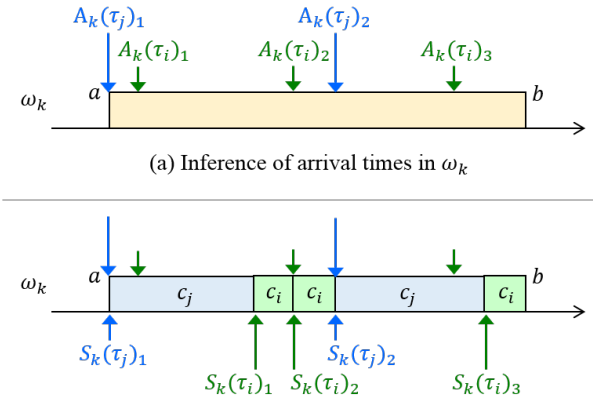
Once the arrival time of every job in each busy interval is obtained, reconstruction of the task schedule is close to completion. By this point, there is sufficient and independent information for every busy interval making it possible to selectively rebuild the schedule of any busy interval using our *compact scheduling translator* (the implementation of such translator is presented in Appendix G). For a selected busy interval ω_k, the *compact scheduling translator* takes arrival times A_k and Γ as input to perform scheduling algorithm. A_k here can be interpreted as a prearranged arrival queue where the scheduling translator only processes the given jobs. The output of this process is the set of start times S_k for all jobs within the busy interval ω_k.

Figure 16 presents an example of schedule reconstruction of busy interval ω_k in the presence of τ_i and τ_j where $N_k(\tau_i) = 3$, $N_k(\tau_j) = 2$ and $pri(\tau_i) < pri(\tau_j)$. Part (a) shows the arrival times of each job (shown using down arrows) that are obtained from arrival windows and (b) presents the start times (shown using up arrows) outputted from the scheduling translator. If necessary, by repeating this process over all busy intervals, the whole schedule can be reconstructed.

IV. Triggering Collaborative Attacks

To demonstrate the the feasibility and utility of the proposed schedule reconstruction scheme on a realistic platform, we implemented a targeted cache timing attack on *Zedboard* that leverages provided schedule information. The cache timing attack is implemented in a collaborative compromised task which behaves normally until it receives a signal from the idle task after the schedule is reconstructed.

A cache timing attack utilizes time difference between cache-hit and cache-miss when accessing data via processor's



(a) Inference of arrival times in ω_k

(b) Translation of start times for corresponding jobs

Figure 16: Translation of start times S_k from arrival times A_k in busy interval ω_k where $pri(\tau_i) < pri(\tau_j)$.

cache to estimate the amount of attacker's data existing in the cache. By knowing how much data is evicted from the cache, an attacker can estimate the cache usage of the victim task and potentially infer its memory usage as well.

In this demonstration, we target a camera task which compresses pictures taken by a camera module. It runs under $30ms$ in period with $10ms$ worst case execution time. The camera task initiates with no-operation mode and starts capturing photos with low resolution by default. It switches to high resolution mode when details need to be captured. The switch between these modes results in a shift in memory usage by the camera task. The goal of our attack from the malicious task, which runs under $10ms$ in period and $2ms$ in worst execution time with priority higher than the camera task, is to detect the switch between the camera modes by monitoring the tasks' cache usage. Such an attack for example could help the attacker identify locations of special interest for an UAV during a reconnaissance mission.

To launch a cache timing attack, the malicious task must locate the start time of the victim task, which is done by referring to the reconstructed schedules. The idle task then issues a signal to malicious task prior to an execution of the victim task, and the malicious task launches the attack by using Algorithm 2. In traditional enterprise computers or cloud computing settings an adversary may be able to periodically launch the attack till he finds the victim task executing, but in a real-time systems such a strategy might result in missed deadlines for legitimate tasks leading to scrutiny of the system and increased risk of detection for the attacker.

Figure 17 presents the inference of the cache usage amount and the memory usage patterns. The results suggest that the transitioning points of the memory usage at sampling count 15, 70 and 90 can be easily observed from the inferred cache usage pattern. Note that, in this demonstration, no other tasks preempted the victim task. Undoubtedly, presence of another task that can preempt the victim would cause some distortion on the cache usage amount. Nevertheless, the pattern will be recognizable in this case since the camera task consumes the most memory, which makes the usage from other tasks subtle and negligible. Furthermore, knowing the schedule could help

Algorithm 2 Cache Timing Attack

1: $\{T_{hit}$: *cache access time with all cache-hit* $\}$
2: $\{T_{miss}$: *cache access time with all cache-miss* $\}$
3: $\{T_{victim}$: *cache access time after victim task* $\}$
4: {*time(CacheRead): measure access time to all cache lines*}
5:
6: *Clear all cache lines*
7: $T_{miss} \leftarrow$ *time(CacheRead)*
8: $T_{hit} \leftarrow$ *time(CacheRead)*
9: **while** *victim task has not ended* **do**
10: *NOP*
11: **end while**
12: $T_{victim} \leftarrow$ *time(CacheRead)*
13: **return** $(T_{victim} - T_{hit})/(T_{miss} - T_{hit})$

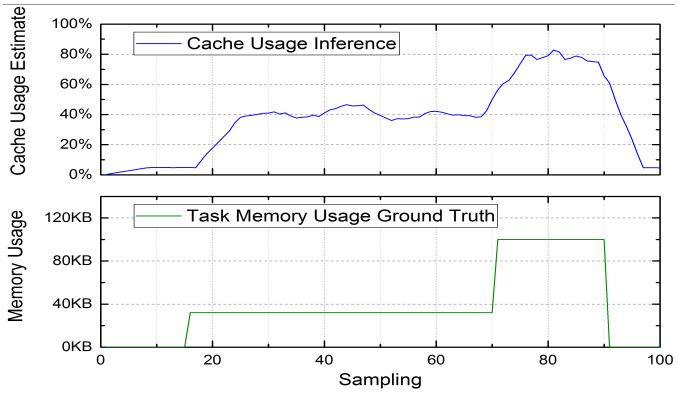


Figure 17: Cache timing attack over a camera task that shows how the transition of the memory usage (green line) can be recovered by the observation of the cache usage (blue line).

the attacker account for noise from other tasks.

V. Implementation and Evaluation

The ScheduLeak algorithm has been implemented on a hardware board running a real-time operating system as well as in a simulation platform. The implementation on the hardware board demonstrates the feasibility of carrying out such attacks on realistic real-time systems while the simulator is used to explore a larger design space for the system parameters. We first provide details of the implementation and simulation engines and then present an evaluation for them.

A. Implementation Environment

We implemented the ScheduLeak algorithm on an ARM-based development board, **Zedboard** [2]. This embedded board includes $512MB$ of DDR3 memory and $256Mb$ Flash and is equipped with a *Xilinx Zynq-7000 XC7Z020 All Programmable SoC*. The *XC7Z020 SoC* includes a *dual-core ARM Cortex-A9 processor* and each core has a private $32KB$ L1 cache and shares a $512KB$ L2 cache. The cores also share a *Xilinx programmable logic FPGA*. The processor runs at $666.67MHz$ and drives a shared 64-bit *Global Timer (GT)* that is clocked at $\approx 333.3MHz$. Our real-time tasks execute on one of the processor cores and do not use the FPGA functions. We tested our implementation with a variety of different task sets, starting from the UAV model (Figure 1 and Section II) to many synthetic task sets.

The real-time task sets ran on a real-time operating system, *FreeRTOS*, which is a lightweight and open source real-time kernel [1]. It is powered by a priority-based preemptive real-time scheduler that allows higher priority tasks to preempt lower priority tasks. Every task in *FreeRTOS* is in one of the four states: *running*, *ready*, *suspended* and *blocked*. The task currently assigned to the CPU is in the running state while one that is in the ready queue but blocked because of a higher priority task is in the ready state. Tasks are in the suspended state when it completes execution of one job or is preempted by a higher priority job.

To build the "ground truth", *i.e.,* to test the correctness of the results obtained from reconstructing the schedules using the ScheduLeak algorithm, the FreeRTOS kernel has been modified to record time stamps for context switches. Also the malicious functions that execute on the idle task delivers the inference of start times to an experiment data handler. This module outputs the log in conjunction with the context switch timestamps through the UART console – these outputs form the ground truth in our system.

The Zedboard implementation is used to demonstrate that the ScheduLeak algorithm can *(a)* execute with good results on a real platform and *(b)* tolerate jitters in a realistic environment. The simulation tool (described later) is used to evaluate the algorithm on a wider range of task sets and to compare results with those from the Zedboard.

B. Launching the Attack

FreeRTOS uses the idle task to handle slack times when no task is executing or waiting in the ready queue[8]. The idle task is created by FreeRTOS in the `vTaskStartScheduler()` function at system startup by calling `xTaskCreate()` – this is the same function used to create user tasks. The main function of the idle task is to consume slack time by looping infinitely. Like other tasks in the system, the idle task has to yield time to the scheduler when required.

We developed a feasible scenario to hijack the idle task. Recall from Section II that large and complex real-time systems are often developed using a vendor-based model and some of the components may not be trusted (or fully secure). Hence, malicious code could enter and be placed (or have already been present) at a certain memory location. The attacker, with prior knowledge of this location could leverage the fact that FreeRTOS has a flat memory model (and no virtual memory or other protection mechanisms such as address-space layout randomization) and execute this code. In fact, we tested out a buffer overflow attack on a *strcpy()* implementation [15] where a specially crafted command was inserted via a remote terminal. Successful execution of the malicious code resulted in a hijack of the idle task.

FreeRTOS also includes an *idle hook function* that allows designers to execute their own functions within the idle task if needed. The idle hook function, `vApplicationIdleHook()`, is called once every cycle in the idle task. However, in practice, it is uncommon for

[8]This is typicaly behavior in real-time systems where stringent timing and safety constraints prevent systems from being suspended when there is nothing to execute. In fact, many such systems use an Idle task to loop and wait for the next task.

an application to use the idle task in this way. This gives attackers an opportunity to hijack the idle hook without downgrading the original system functionality, thus avoiding detection. Instead of directly injecting malicious code into the idle hook function we use the `jump` command. Since FreeRTOS does not support virtual memory we can locate the physical address of the idle hook function by declaring "`extern void vApplicationIdleHook(void);`" and get the address with `&vApplicationIdleHook`. Then we replace the jump command with a new one pointing to our malicious function.

C. Capturing Busy Intervals

The *Global Timer (GT)* that is accessible to user tasks in *FreeRTOS* is used as a time reference to capture busy intervals. Remember that it is clocked at half the processor frequency, *i.e.,* $667MHz/2 = 333.5MHz$. Hence, the resolution of the *Global Timer* can be computed by $1/333.5MHz \approx 3ns$. To capture a busy interval, the idle task uses *Global Timer* as a reference to inspect whether or not the idle task has been preempted. It reads the *Global Timer* in a loop and compares it with the time stored from last time it was preempted. If nothing preempts the idle task during a loop, then the difference for times between consecutive loops remains below one unit of idle task execution time (*i.e.,* $447ns$ as identified in the on-board results part of the Evaluation section). In contrast, if the difference is greater than expected, then a preemption must have occurred.

The algorithm for capturing a busy interval is presented in Algorithm 3. The idle task repeats the same process until it collects all busy interval instances for at least one hyper-period. After that, the program moves to the busy interval analysis stage. Section V presents a discussion on the granularity of the idle task measurements.

Algorithm 3 Capturing A Busy Interval

```
1: {GT : global timer}
2: {t_0 : current time stamp}
3: {t_-1 : last time stamp}
4:
5: t_-1 ← GT
6: duration ← 0
7: while duration ≤ idle task execution time unit do
8:     t_0 ← GT
9:     duration ← (t_0 − t_-1)
10:    t_-1 ← t_0
11: end while
12: return t_0, duration
```

D. Experimental Setup

We also evaluated the ScheduLeak approach using an internally developed *scheduling simulation tool*. This tool is used to test the scalability of the algorithm and also to test with a more diverse set of real-time task combinations.

We apply the analysis algorithm (from Section III) to various random synthetic task sets and checking whether the inference of start times matches the corresponding ground truth. The task sets are grouped by utilization from $[0.01 + 0.1 \cdot i, 0.1 + 0.1 \cdot i]$ where $0 \leq i \leq 9$. Each utilization group contains 6 subgroups that have a fixed number of tasks from 10 to 15. Each subgroup contains 1000 task sets. In other words, 6000 task sets are generated in each utilization group resulting in 60000 task sets to test for *one* condition.

In order to obtain uniform hyper-periods (for ease of comparison), task periods are selected from combinations of numbers in $[2, 3, 5, 7, 11, 13]$ – this results in a hyper-period of 30030 for all task sets. Note that the reason for choosing prime numbers is because we want to generate non-harmonic tasks[9] (we will discuss this later on). Task sets with the same period are excluded from the test to avoid duplicate effort. The priorities of tasks are assigned by the rate-monotonic algorithm [17], *i.e.,* a task with a shorter period is assigned a higher priority. We only pick those task sets (to form the $60,000$) that are schedulable by fixed-priority scheduling algorithms.

Figure 18 shows a snapshot of the simulation tool used in this paper. For each task set, we obtain its schedule using a scheduling simulator. The schedule is simulated for one hyper-period since schedules repeat every hyper-period thereafter. The start time for each job is recorded to form the "ground truth" for comparison purposes. The schedule is then converted to blocks of busy intervals (with no information on arrival times, start times, *etc.*). These chunks of busy intervals (essentially black boxes) are fed as input to the ScheduLeak algorithm for reconstruction. The algorithm outputs a series of start times that are then compared with the ground truth (based on metrics defined in Section V-E). Jitters are not considered in this simulation experiment since we want to inspect the potential errors (if any) introduced by the algorithm itself. An evaluation that includes jitter is discussed in Section V-G.

E. Performance Metrics

For a given task set containing $\{\tau_1, \cdots, \tau_n\}$, let $\mathcal{S}$ be the set of all schedules. Note that $|\mathcal{S}| \leq \prod_{i=1}^{n} p_i$. For an observation O, we define $\mathcal{S}_O \subset \mathcal{S}$ as the set of all schedules that gives observation O. Note that all the $\mathcal{S}_O$'s partition $\mathcal{S}$.

Definition 1. For an observation O, we call elements of $\mathcal{S}_O$ as "indistinguishable schedules" w.r.t. observation O. Hence, two schedules are indistinguishable if they both result in the same observation.

Note that since we only see observation O, no task analysis algorithm can distinguish the elements of $\mathcal{S}_O$. Therefore, we define the performance as follows: In our approach we obtained arrival windows $\{\overline{A(\tau_1)}, \cdots, \overline{A(\tau_n)}\}$ for tasks. Let us define $\mathcal{A}$ as the set of all schedules created by our arrival windows. We define the performance of the algorithm as $\eta = 1 - \frac{|\mathcal{A} \triangle \mathcal{S}_O|}{|\mathcal{A} \cup \mathcal{S}_O|}$, where $\triangle$ is the symmetric difference operator defined as $A \triangle B = (A \cap B^c) \cup (B \cap A^c)$. Note that η will be between 0 and 1 where 1 indicates the algorithm has narrowed down $\mathcal{S}$ to $\mathcal{S}_O$ for the given observation O and has optimum performance. While the ultimate goal of the algorithm is for η to be as close 1 as possible, this metric is not easy to calculate in practice. Hence, we define an alternative metric, "Inference Precision Ratio" to evaluate the performance of the algorithm.

[9] A task set is harmonic if periods of tasks are pairwise divisible.

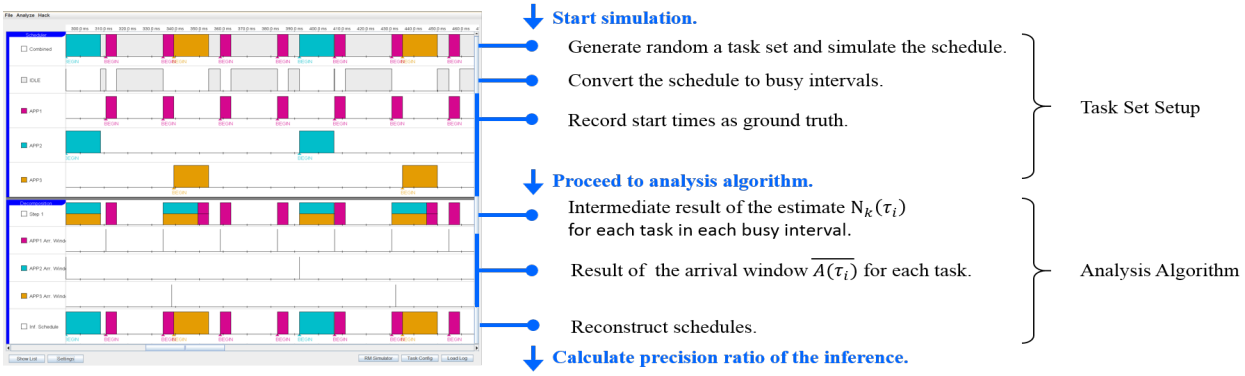


Figure 18: The simulation tool that integrates a fixed-priority scheduling simulator and the analysis algorithm is used to process the simulation experiments conducted in this paper for the evaluation.

Inference Precision Ratio: One way to evaluate the performance of an algorithm is to compare the schedules from the output with that from the ground truth. Let the start times for task τ_i be $\{S^*(\tau_i)_1, \cdots, S^*(\tau_i)_s\}$ and the start times estimated by our algorithm be $\{S(\tau_i)_1, \cdots, S(\tau_i)_s\}$. Let $E(\tau_i)_j = S^*(\tau_i)_j - S(\tau_i)_j$ be the error in estimating the start time of the j^{th} appearance of task τ_i. Therefore, for task τ_i we have the errors $E(\tau_i) = \{E(\tau_i)_1, \cdots, E(\tau_i)_s\}$. We define the standard deviation of these errors from zero as, $SD_i = \sqrt{\frac{1}{s}\sum_{j=1}^{s}(E(\tau_i)_j)^2}$. We now define the efficiency of the estimation of the start time of task τ_i as $(1 - \frac{SD_i}{p_i})$ where p_i is the period of task τ_i. This value is a number in $[0, 1]$ where 1 indicates an *exact estimation* of the start times[10]. For a task set containing n tasks, we define the overall estimation efficiency (precision ratio) of the algorithm as the geometric mean of the estimation efficiency of tasks in the set, as follows:

$$\eta' = (\prod_{i=1}^{n}(1 - \frac{SD_i}{p_i}))^{\frac{1}{n}} \quad (4)$$

F. Simulation-based Results

We first evaluate the precision ratio of our algorithm using the Equation 4. To examine the factors affecting performance of the algorithm we test it using tasks sets that are generated under four different conditions (Table II): *(A)* without initial offset; *(B)* without initial offset and harmonic periods; *(C)* with random initial offset and finally *(D)* with random initial offset but without harmonic periods. Figure 19 shows the precision ratio of the algorithm for the task sets in each condition while Table III shows the geometric mean of the precision ratios for each utilization group in each test condition.

The results suggest that the algorithm performs well when all tasks are synchronized and have no initial offset (*i.e.,* *Condition.A* and *Condition.B* as shown in Figure 19a and 19b) which yields the overall precision ratio of 1.0. This is due to the fact that during the process of converting an arrival window to its corresponding arrival time, we always pick the *start point of the window as the arrival time for simulating the schedules*. This matches the condition that all tasks start at the same time. Hence, task sets with random initial offsets (*Condition.C*) are expected to produce some uncertainties.

[10]Hence we know exactly when the task starts so side-channel attacks launched against it (like Section V-B) have a much higher chance of success.

	Allow Harmonic Tasks	w/o Harmonic Tasks
w/o Initial Offset	*Condition.A*	*Condition.B*
w/ Initial Offset	*Condition.C*	*Condition.D*

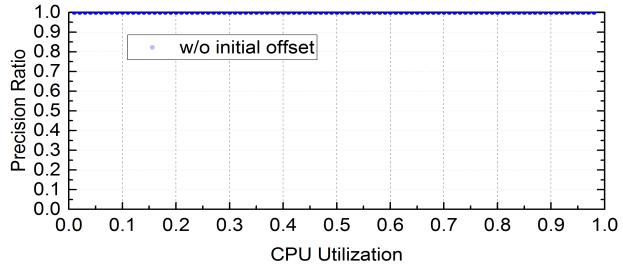
Table II: Task sets generated under four different conditions are tested. Each condition tested with 60000 task sets, 10 utilization groups, 6 subgroups. Number of tasks per task set chosen from 10 to 15 (*i.e.,* 1000 task sets per subgroup.

Results from Table III and Figure 19(c) show that the precision is reduced, albeit a little, for *Condition.C*. The mean of the precision ratio drops when the utilization is above 0.5, *i.e.,* when the system is heavily loaded as expected. Nevertheless, the overall precision is still reasonably high. See the inset in Figure 19c with Y-axis adjusted from 0.95 to 1.0 that provides more details (note: a heavier color indicates higher occurrence). The inset clearly shows that a majority of task sets have a precision close to 1.0 (nearly 93.37% of task sets have precision ratio of 1.0 even for *Condition.C*).

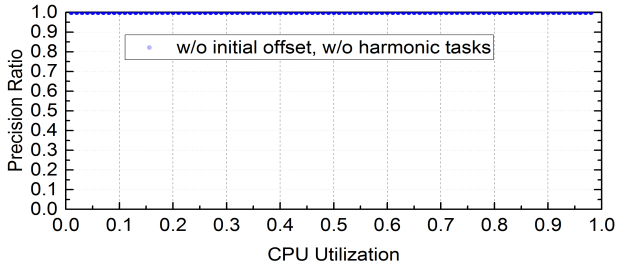
We analyze task sets that result in precision values below 1.0 to find factors contributing to the uncertainty – the results lead us to the presence of *harmonic tasks*[11]. Remember that in our random task set generator we only ensure that each task period is distinct. Hence, it is possible to get a task set with harmonic tasks. Those task sets are either completely harmonic or partially harmonic (only a couple of them are harmonic but the rest are not). The problem is that harmonic tasks often appear together –this may produce bigger busy intervals. In the worst case, a task may always be scheduled together with other tasks throughout the schedule – this makes it harder to tell them apart. To test this hypothesis, the task set generator was used to produce non-harmonic task sets only for *Condition.A* and *Condition.C* – they transform into *Condition.B* and *Condition.D* respectively.

Results are shown in Figure 19(b) and 19(c). Note that *Condition.B* is used to check whether we get results consistent with *Condition.A*, and *Condition.D* is used to compare with *Condition.C* to evaluate the influence of harmonic tasks. As seen in the figure, results from B and A are identical, suggesting that the algorithm works properly regardless of the presence of harmonic tasks. The precision ratio rises instantly for D when harmonic tasks are removed. Note that the drop in precision is very small and for most task sets we are very close to 1.

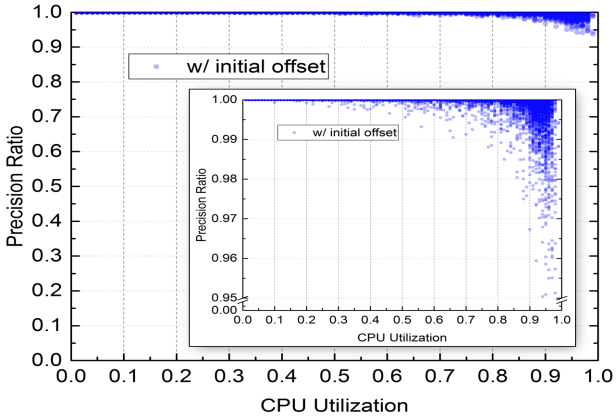
[11]Tasks with periods that are integral multiples of each other.



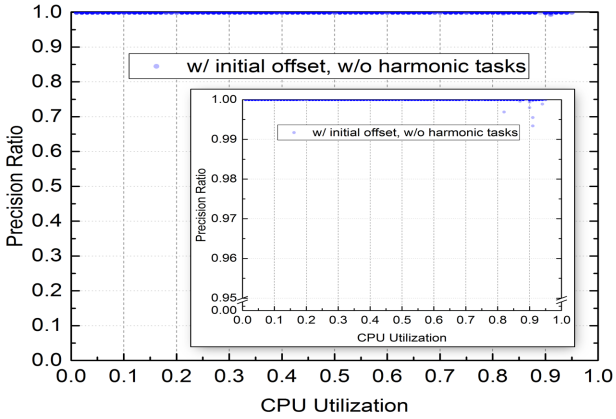
(a) *Condition.A*: without initial offset



(b) *Condition.B*: without initial offset and harmonic tasks



(c) *Condition.C*: with random initial offset



(d) *Condition.D*: with random initial offset but without harmonic tasks

Figure 19: Experiment results of four test conditions. It shows that the analysis algorithm is only downgraded in *Condition.C* and *Condition.D* (both are *with initial offset*) with acceptable precision ratio roughly above 0.95. Note that 60000 task sets are tested in each condition.

Utilization	*Condition.A*	*Condition.B*	*Condition.C*	*Condition.D*
[0.0,0.1]	1.0	1.0	1.0	1.0
[0.1,0.2]	1.0	1.0	1.0	1.0
[0.2,0.3]	1.0	1.0	1.0	1.0
[0.3,0.4]	1.0	1.0	1.0	1.0
[0.4,0.5]	1.0	1.0	1.0	1.0
[0.5,0.6]	1.0	1.0	0.99999	1.0
[0.6,0.7]	1.0	1.0	0.99998	1.0
[0.7,0.8]	1.0	1.0	0.99994	1.0
[0.8,0.9]	1.0	1.0	0.99977	1.0
[0.9,1.0]	1.0	1.0	0.99829	0.99999

Table III: Geometric mean of precision ratio for each utilization group in each condition. Conditions are referred to Table II, which represents (A) w/o initial offset. (B) w/o initial offset and harmonic periods. (C) w/ random initial offset. (D) w/ random initial offset but w/o harmonic periods.

	Condition.A	*Condition.B*	*Condition.C*	*Condition.D*
Max	1.0	1.0	1.0	1.0
Geometric Mean	1.0	1.0	0.99979	1.0
Min	1.0	1.0	0.9504	0.9873

Table IV: Maximum, minimum and geometric mean values of the precision ratio in each test condition. The result indicates that even for the worst case (*Condition.C*: *with initial offset*) it maintains a high accuracy level as 0.9504 in precision ratio.

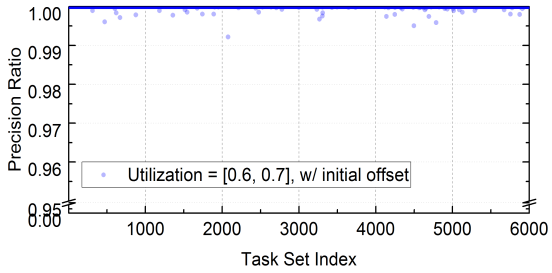
Another important factor that affects the precision of our analysis is the *utilization* of the real-time task sets. Figure 20 shows the precision ratio data for utilization groups [0.6, 0.7] and [0.9, 1.0][12]. Higher utilization indicate *(i)* smaller idle times and *(ii)* larger busy intervals – both of which are detrimental to our analyses. Hence, more ambiguous conditions show up while trying to estimate the number of arrivals in a busy interval. Having said that, even for the worst-case scenario (Table IV) the precision reaches 0.9504 – which is really high and can still be used to launch targeted attacks. Figure 23 in the Appendix presents similar data for the remaining utilization groups.

[12][0.6, 0.7] is the average case while [0.9, 1.0] is the worst case.

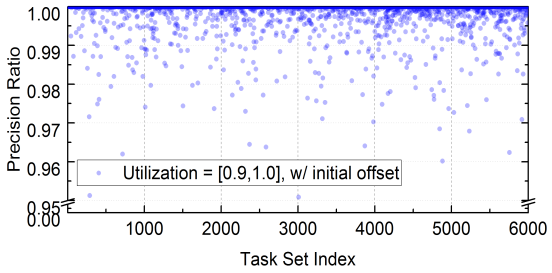
G. Zedboard-based Results

We first examine the cost for execution of the idle task and the impact on the ability to capture busy intervals. As mentioned earlier, the idle task uses a loop to read time values from the *Global Timer*. The shortest busy interval that can be measured depends on the cost for reading the timer and executing the idle task instructions. We measure the cost of each idle loop to be $447ns$, on average, in the absence of preemption. This is really small when compared to the execution times for the real-time tasks (typically more than $10\mu s$). The global timer counts $447ns/3ns \approx 149$ (See Section V-C) times during each idle loop. On the other hand, one of the smallest busy intervals (say, just one task with execution time $10\mu s$) will result in the counter incrementing $10\mu s/3ns \approx 3333$. Hence, it is extremely unlikely that the idle task will miss any busy intervals.

The costs are different when a busy interval is captured since the timing now involves a pair of *context switches* (idle task in and out). From our experiments the cost now becomes $18.48\mu s$ on average. Hence, this adds to the length of any busy interval but the costs are bounded since each measurement only includes two idle task loops (Section V-C). We can confidently remove these costs from every busy interval. This leaves only *jitter* as a source of uncertainty in our measurements. From



(a) Utilization Group [0.6, 0.7] in *Condition.C*



(b) Utilization Group [0.9, 1,0] in *Condition.C*

Figure 20: Detailed experiment data of *Condition.C*. The results suggest that higher utilization leads to lower precision ratio.

our experiments, the removal of the idle task costs results in a $0.28\mu s$ error, on average, for each busy interval. If the delta due to this error is greater than the execution time for the shortest task then it introduces uncertainty into our analysis since we cannot tell it apart from a legitimate task. In practice, this depends on the actual application task sets. For the UAV model (presented earlier on), the shortest task (the network manager) has an execution time of $30\mu s$ – orders of magnitude higher than the error/delta. Hence, the errors in measuring the busy intervals do not really affect our analyses.

We now evaluate the precision ratio of the inference on the *Zedboard*. To tolerate jitters, the equations for estimating $N_k(\tau_i)$ values, the number of arrivals for a task in a busy interval, are replaced by the equations introduced in Appendix C. The remainder of the procedure remains unchanged. The calculation of the precision ratio uses Equation 4. The result shows that, for those task sets that have a precision ratio of 1.0 in the simulations, the average value on *Zedboard* is around 0.9977. The jitters naturally induced by hardware result in uncertainties that lowers the precision ratio. Nonetheless, the result shows that the algorithm can work properly to reconstruct schedules with high accuracy on a realistic platform.

We also propose a modification of part (i) in Theorem 2 to improve the performance of building arrival windows (Section III-C). This change will result in narrower estimated arrival windows and hence improve the performance. We omit the evaluation of this minor change here. The improved version of Theorem 2 is provided as Theorem 4 in the Appendix.

H. Overhead Evaluation

It is important to note that the analysis of busy intervals need not necessarily be performed online. For some attack scenarios, the data can be analyzed offline. Hence, the analysis will not be limited by the performance of the hardware. In this section though, we focus on the overheads for carrying out the analysis on the actual board to demonstrate the feasibility of applying the algorithm in an online fashion.

The estimate of $N_k(\tau_i)$ uses Equation (1) to find the matching combinations for a busy interval. This has order of 2^n time complexity. However, the number of tasks is often fixed for a task set, thus 2^n is bounded in a given real-time system. Table V shows time consumption for completing the estimate for *one busy interval* with 10 to 15 tasks per task set. The results suggest that while checking for valid combinations of $N_k(\tau_i)$ is a time-consuming operation, this data only demonstrates the worst case when every task's $N_k(\tau_i)$ is either N or $N+1$. The real execution time depends on the length of the busy interval and the corresponding $N_k(\tau_i)$ values for each task.

Number of Tasks	Possible Combinations	Total Time Consumption	One Combo Average Time
10	1024	58.94ms	57.55us
11	2048	129.72ms	63.33us
12	4096	283.43ms	69.20us
13	8192	642.12ms	78.38us
14	16384	1444.40ms	88.15us
15	32768	3260.45ms	99.50us

Table V: Execution time measurement for computing all possible N_k values for one busy interval.

The computation of calculating arrival windows depends on the number of instances of each task in a hyper-period. From our experiments, it takes $2.073ms$ to get arrival windows for the 10-task task set mentioned above where each task executes for around $10.5\mu s$ on average. For the elimination of mismatched $N_k(\tau_i)$ values it takes arrival windows of n tasks to inspect each estimate in every busy interval. This has a complexity in the order of n. In the same experiment as above, it takes $17.4us$ to iterate through 14 busy intervals with 10 arrival windows from the correspondent 10 tasks. Finally, for a task set with 10 tasks that have 14 busy intervals on our *Zedboard*, it takes $828.05ms$ to complete the total analysis.

VI. Uncertainty Faced by Adversaries

In the evaluation, we test the algorithm with task sets having three variables: *(i)* initial offset, *(ii)* with or without harmonic tasks and *(iii)* CPU utilization. From the results, we can conclude that the factors influencing the precision of our algorithm are: *(a)* having harmonic tasks (either partially or entirely) in the task set reduces the precision of the inference; *(b)* having random initial offset adds uncertainty of the schedule; *(c)* higher utilization means larger duration and more tasks involved in one individual busy interval, which increases the difficulty of the inference.

The schedule reconstruction mechanism in ScheduLeak leverages the *limited uncertainty* in the repeating schedules generated by a fixed-priority scheduler. Defenders may *randomize* the schedules so that there is no guarantee that the next hyper-period will show the exact same order (and timing) of execution for the tasks. For instance, by picking a random task instead of the one with the highest-priority at each scheduling point, subject to the deadline constraints [29]. Figure 21 shows a part of the schedule for the task set in Example 5 generated by the randomization algorithm. It shows tasks being *shuffled* within each busy interval.

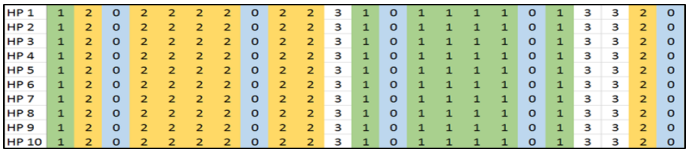
To counter such randomization methods, let the arrival time of one instance of task τ_i be denoted by $A(\tau_i)$ and we want

to predict its corresponding start time $S(\tau_i)$. If at any instance in time there are many other tasks waiting to be executed, then by the properties of such randomization scheduling, $S(\tau_i)$ could be at any time in the interval $[A(\tau_i), A(\tau_i) + p_i - c_i]$. Therefore, even if $A(\tau_i)$ is known, no algorithm can predict $S(\tau_i)$ accurately. On the other hand, if we find a time in the busy interval such that τ_i is the only buffered task and no other task is being executed, then by the work conserving property of the scheduler, τ_i has to be executed and we choose this point as the latest time for $S(\tau_i)$. We can use the arrival window estimations (note: arrival times do not depend on the scheduling policy; so they are still of use in the face of randomization) in our algorithm to find such time instances.

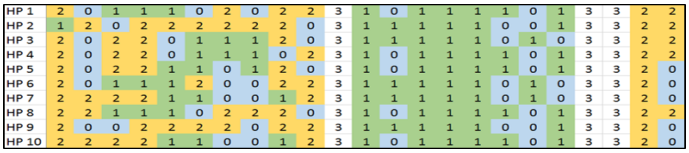
We focus on each busy interval and look for the following condition: Suppose for $i \in \{1, ..., j\}$ the intervals $\overline{A(\tau_i)} = [a_i, b_i]$ with $a_i \leq a_{i+1}$, for $1 \leq i \leq j-1$, are the arrival windows estimated by our algorithm in a given busy interval. If there exists point $x_i \in \overline{A(\tau_i)}$ such that $x_i > a + \sum_{k<i} c_k$ (a is assumed to be the start point of the busy interval) and $x_i < a_{i+1}$, then we change b_i to x_i, and hence we can make the arrival window narrower. Due to the work conserving property of the scheduler we understand that τ_i should have started to be executed by x_i. Hence, if we choose the first point of the arrival window as the starting time, the error in estimating the start time will be in the interval $[0, x_i - a_i]$. For instance, for the special case that τ_i is the only task whose arrival window starts at the beginning of the busy interval (*i.e.,* $a = a_i$), we *can find the exact starting time* of τ_i. This trick could be also used, when randomization is absent, to improve the accuracy of the estimation of arrival windows. After making arrival window $\overline{A(\tau_i)}$ narrower by changing b_i to x_i, we can take its intersection with the arrival window in other segments. Hence, all of them become narrower. This step could be implemented by having a list of ready to execute tasks. If at some point there is only one task in this list and no task is being executed, due to the work conserving property of the scheduler, that task should be executed. So our ScheduLeak mechanism can even deal with some randomization-based defenses.

VII. Related Work

Recent work on real-time systems security has demonstrated the feasibility of information leakage through task scheduiling. Son *et al.* [23] highlighted the susceptibility of rate-monotonic schedulers to covert timing channel attacks due to timing constraints. Völp *et al.* [26] presented modifications to fixed-priority scheduling, altering thread blocks that potentially leak information with the idle thread to avoid the exploitation of timing channels. Völp *et al.* [25] examined shared-resource covert channels in real-time schedules and addressed it by using transformed resource locking protocols.



(a) No randomization



(b) Randomization

Figure 21: Partial Schedules with and without the randomization for 10 hyper-periods.

Kadloor *et al.* [13] introduces a methodology for quantifying side-channel leakage for first-come-first-serve and time-division-multiple-access schedulers. Gong and Kiyavash [10] analyzed deterministic work-conserving schedulers, for which they discovered a lower bound for the total information leakage. The collaborative version of this problem, in which two users form a covert timing channel in a shared scheduler to steal private information from a secure system is studied by Ghassami *et al.* [9]. While in the above works the attacker uses traffic analysis to obtain information about the user activities, our work is primarily concerned with the analysis of individual tasks by reconstructing the original task schedule.

Mohan *et al.* [21] considered the problem of direct information leakage between real-time tasks through architectural resources such as shared cache. They introduced a modified fixed-priority scheduling algorithm that integrates security levels into scheduling decisions. Pellizzoni *et al.* [22] extended the above scheme to a more general task model and also proposed an optimal priority assignment method that determines the task preemptibility while meeting all the timing requirements.

Cache-based side-channels contain the highest bandwidth among all side-channels, making them invaluable for information leakage [11]. The growth of cloud computing has caused such attacks face increased scrutiny [5], [33]. Wang and Lee [27] who presented a hardened cache model to mitigate side-channels by adopting partitioning and memory-to-cache mapping randomization techniques. However, most existing methods aimed at reducing information leakage through cache-based side-channels have not considered real-time systems. While other methods of side-channel attacks exist such as power, electromagnetic and frequency analysis [24] [4], they are not the focus of this paper. Jiang *et al.* [12] developed a framework for examining the effectiveness of scheduling policies in preventing differential power analysis attacks.

VIII. Conclusion

The methods presented here will improve the design of future real-time systems. Designers will have an increased awareness of attack mechanisms that leak crucial information in such systems. Hence, they can develop methods that prevent (or at least reduce the effectiveness of) such attacks.

We intend to further refine the algorithms (and analyses) presented here and also develop solutions to deter such attacks. We believe that the area of security for real-time systems will require the development of a large body of work, along multiple directions, to ensure the overall safety of such systems.

References

[1] Freertos. www.freertos.org.

[2] Xilinx Zedboard. http://zedboard.org/.

[3] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*, pages 312–320, 2007.

[4] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The em side-channel(s). In *the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, pages 29–45, 2003.

[5] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Fine grain cross-vm attacks on xen and vmware are possible!

[6] Theodore P Baker. An analysis of edf schedulability on a multiprocessor. *Parallel and Distributed Systems, IEEE Transactions on*, 16(8):760–768, 2005.

[7] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 1997.

[8] Thomas M. Chen and Saeed Abu-Nimeh. Lessons from stuxnet. *Computer*, 44(4):91–93, April 2011.

[9] AmirEmad Ghassami, Xun Gong, and Negar Kiyavash. Capacity limit of queueing timing channel in shared fcfs schedulers. In *Information Theory (ISIT), 2015 IEEE International Symposium on*, pages 789–793. IEEE, 2015.

[10] Xun Gong and Negar Kiyavash. Timing side channels in shared queues. *CoRR*, abs/1403.1276, 2014.

[11] Wei-Ming Hu. Lattice scheduling and covert channels. In *Research in Security and Privacy, Proceedings., IEEE Computer Society Symposium on*, 1992.

[12] Ke Jiang, L. Batina, P. Eles, and Zebo Peng. Robustness analysis of real-time scheduling against differential power analysis attacks. In *VLSI (ISVLSI), 2014 IEEE Computer Society Annual Symposium on*, pages 450–455, July 2014.

[13] Sachin Kadloor, Negar Kiyavash, and Parv Venkitasubramaniam. Mitigating timing side channel in shared schedulers. *CoRR*, abs/1302.6123, 2013.

[14] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[15] Kyung-Suk Lhee and Steve J. Chapin. Buffer overflow and format string overflow vulnerabilities. *Software: Practice and Experience*, 33(5):423–460, 2003.

[16] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward A. Lee. Predictable programming on a precision timed architecture. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 137–146, 2008.

[17] 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.

[18] Fangfei Liu, Y. Yarom, Qian Ge, G. Heiser, and R.B. Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*, pages 605–622, 2015.

[19] Jane W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[20] Sibin Mohan, Stanley Bak, Emiliano Betti, Heechul Yun, Lui Sha, and Marco Caccamo. S3A: Secure system simplex architecture for enhanced security and robustness of cyber-physical systems. In *ACM Conference on High Confidence Networked Systems*, 2013.

[21] Sibin Mohan, Man-Ki Yoon, Rodolfo Pellizzoni, and Rakesh Bobba. Real-time systems security through scheduler constraints. In *Euromicro Conference on Real-Time Systems*, pages 129–140, July 2014.

[22] Rodolfo Pellizzoni, Neda Paryab, Man-Ki Yoon, Stanley Bak, Sibin Mohan, and Rakesh Bobba. A generalized model for preventing information leakage in hard real-time systems. In *IEEE Real-Time Embedded Technology and Applications Symposium*, 2015.

[23] Joon Son and J. Alves-Foss. Covert timing channel analysis of rate monotonic real-time scheduling algorithm in mls systems. In *Information Assurance Workshop, 2006 IEEE*, pages 361–368, 2006.

[24] Chin Chi Tiu. A new frequency-based side channel attack for embedded systems. Technical report, 2005.

[25] Marcus Völp, Benjamin Engel, Claude-Joachim Hamann, and Hermann Härtig. On confidentiality preserving real-time locking protocols. In *IEEE Real-Time Embedded Technology and Applications Symposium*, 2013.

[26] Marcus Völp, Claude-Joachim Hamann, and Hermann Härtig. Avoiding timing channels in fixed-priority schedulers. In *ACM Symposium on Information, Computer and Communication Security*, pages 44–55, New York, NY, USA, 2008. ACM.

[27] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 494–505, New York, NY, USA, 2007. ACM.

[28] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

[29] Man-Ki Yoon, Sibin Mohan, Chien-Ying Chen, and Lui Sha. SCOBE: A schedule randomization protocol for obfuscation against timing inference attacks in hard real-time systems. Technical report, University of Illinois at Urbana Champaign, 2015. http://hdl.handle.net/2142/88405.

[30] Man-Ki Yoon, Sibin Mohan, Jaesik Choi, Jung-Eun Kim, and Lui Sha. SecureCore: A multicore-based intrusion detection architecture for real-time embedded systems. In *IEEE Real-Time Embedded Technology and Applications Symposium*, 2013.

[31] Man-Ki Yoon, Sibin Mohan, Jaesik Choi, and Lui 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.

[32] Mohammad Mehdi Zeinali Zadeh, Mahmoud Salem, Neeraj Kumar, Greta Cutulenco, and Sebastian Fischmeister. SiPTA: Signal processing for trace-based anomaly detection. In *Proc. of the International Conference on Embedded Software*, 2014.

[33] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 305–316, New York, NY, USA, 2012. ACM.

[34] Yuting Zhang and Richard West. Process-aware interrupt scheduling and accounting. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 191–201, 2006.

APPENDIX

A. Proof of Theorem 1

Consider t_0^i, t_1^i, t_2^i, $\cdots$ as the arrival times of task τ_i and define $\mathcal{S} = \{t_0^i, t_1^i, t_2^i, \cdots\}$. We use the following two lemmas to prove Theorem 1:

Lemma 1. A busy interval contains the k^{th} task of τ_i if and only if it contains t_k^i (either as an interior point or as a boundary point).

Proof: The k^{th} task of τ_i will be released at time t_k^i if there are no tasks with higher priority running at that time, or, will be released immediately after the end of the tasks with higher priority. Hence, in both cases, the system is busy from time t_k^i to at least the finishing time of the k^{th} task of type i. Therefore, a busy interval which contains t_k^i, contains the k^{th} task of τ_i, and a busy interval that contains the k^{th} task of τ_i, should have started at t_k^i or at a time before t_k^i. (Note that the end point of a busy interval cannot belong to $\mathcal{S}$). ■

Lemma 2. If $C(w_k)$ satisfies

$$Np_i < C(w_k) < (N+1)p_i, \qquad N = 0, 1, 2, ... \quad (5)$$

then task τ_i can only has arrived N or $N+1$ times during the busy interval w_k.

Proof: If $Np_i < C(w_k) < (N+1)p_i$, then w_k contains N or $N+1$ points of $\mathcal{S}$. Therefore, by Lemma 1, task τ_i can only arrive N or $N+1$ times during the busy interval w_k. ■

Proof of Theorem 1: (i) If $Np_i - c_i \le C(w_k) < Np_i$, then the busy interval cannot contain $N-1$ points from $\mathcal{S}$, otherwise, a task of type i should have finished in a time interval less that c_i seconds. Therefore, it exactly contains N points from $\mathcal{S}$.
If $Np_i \le C(w_k) < Np_i + c_i$, the start point of the busy interval cannot belong to $\mathcal{S}$ (otherwise, the length of the busy interval should be at least $Np_i + c_i$), therefore, it exactly contains N points from $\mathcal{S}$.
Therefore, by Lemma 1, in both cases, task τ_i can only has arrived N times during the busy interval.
(ii) This part follows from Lemma 2 immediately. ■

B. Uniqueness of the Combination

In this appendix we address the problem of determining under what conditions the task combination found for a busy interval is unique. The key idea is the following:

If we can find two distinct sets of tasks, I^+ and I^- such that the sum of the execution times in these two sets are equal, that is,

$$\sum_{i\in I^+} c_i = \sum_{i\in I^-} c_i, \quad (6)$$

then, ambiguity (non-unique task combination) is possible, and it will happen for a busy interval w_k of length $C(w_k)$, satisfying,

$$C(w_k) = \sum_{i\in I^+} m_i c_i + \sum_{i\in I^-} (m_i+1)c_i + \sum_{i\notin I^+\cup I^-} n_i c_i \quad (7)$$

such that,

$$m_i p_i + c_i \le C(w_k) < (m_i+1)p_i - c_i, \qquad \forall i \in I^+ \cup I^- \quad (8)$$

Note that (8) implies that for $i \in I^+ \cup I^-$, $N_k(\tau_i) = m_i$ or $m_i + 1$.
The reason for non-uniqueness of task combination is as follows:

$$\begin{aligned} &\sum_{i\in I^+} m_i c_i + \sum_{i\in I^-} (m_i+1)c_i + \sum_{i\notin I^+\cup I^-} n_i c_i \\ &= \sum_{i\in I^+} m_i c_i + \sum_{i\in I^-} m_i c_i + \sum_{i\in I^-} c_i + \sum_{i\notin I^+\cup I^-} n_i c_i \\ &= \sum_{i\in I^+} m_i c_i + \sum_{i\in I^-} m_i c_i + \sum_{i\in I^+} c_i + \sum_{i\notin I^+\cup I^-} n_i c_i \\ &= \sum_{i\in I^+} (m_i+1)c_i + \sum_{i\in I^-} m_i c_i + \sum_{i\notin I^+\cup I^-} n_i c_i \end{aligned} \quad (9)$$

Therefore, more than one task combinations are possible for the busy interval of length $C(w_k)$.

C. Values with Uncertainty

Let t_0^i, t_1^i, t_2^i, ... be the arrival times of τ_i and define $\mathcal{S} = \{t_0^i, t_1^i, t_2^i, \cdots\}$. If there is no error in the system we expect $\mathcal{S}$ to be $\{a, a+p_i, a+2p_i, \cdots\}$ for some constant offset a.
We consider the following scenario:

- The k^{th} task of τ_i arrives at time $a + kp_i + \delta_{i,k}$, where $\{\delta_{i,0}, \delta_{i,1}, \delta_{i,2}, ...\}$ is a sequence of $i.i.d.$ real valued random variables with $|\delta_{i,k}| < \delta_i$, $k = 1, 2, ...$
- The execution time of the k^{th} task of type i is $c_i + \gamma_{i,k}$, where c_i is a deterministic and fixed value and $\{\gamma_{i,0}, \gamma_{i,1}, ...\}$ is a sequence of $i.i.d.$ real valued random variables with $|\gamma_{i,k}| < \gamma_i$, $k = 0, 1, ...$

Figure 22 shows an example of the arrival of τ_i.

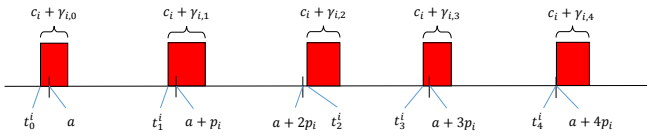


Figure 22

The goal is to find the extension of Theorem 1 for the case that we have uncertainty in the values of task parameters. To this end, we first need the following Lemma:

Lemma 3. For $\delta_i \ll p_i$,
If $0 < C(w_k) < 2\delta_i$, then task τ_i can only has arrived 0 or 1 times during the busy interval.
Furthermore, if $C(w_k)$ satisfies

$$Np_i + 2\delta_i < C(w_k) < (N+1)p_i - 2\delta_i, \qquad N \ge 0 \quad (10)$$

then task τ_i can only has arrived N or $N+1$ times during the busy interval w_k.

Proof: If $0 < C(w_k) < 2\delta_i$, then the busy interval contains 0 or 1 points of $\mathcal{S}$. Therefore, Lemma 1 in Appendix 1 gives the desired result.
If $Np_i + 2\delta_i < C(w_k)$, then the busy interval contains more than $N-1$ points of $\mathcal{S}$, and if $C(w_k) < (N+1)p_i - 2\delta_i$, then the busy interval contains less than $N+2$ points of $\mathcal{S}$. Therefore, by Lemma 1, task τ_i can only arrive N or $N+1$ times during the busy interval.

■

The following theorem is the extension of Theorem 1 for the case that we have uncertainty in the values of execution times and task periods.

Theorem 3. For given values of $C(w_k)$, p_i, c_i, γ_i and δ_i, such that $\gamma_i, \delta_i \ll c_i$,
(i) If $C(w_k)$ satisfies

$$(Np_i - c_i + 2\delta_i + \gamma_i)^+ \leq C(w_k) < Np_i + c_i - 2\delta_i - \gamma_i, \quad (11)$$

then task i can only has arrived N times during the busy interval.
(ii) If $C(w_k)$ satisfies

$$Np_i + c_i - 2\delta_i - \gamma_i \leq C(w_k) < (N+1)p_i - c_i + 2\delta_i + \gamma_i, \quad (12)$$

then task i can only has arrived N or $N+1$ times during the busy interval.

Proof: (i) If the condition in part (i) holds, the busy interval should exactly contain N points from $\mathcal{S}$; otherwise, either the packet corresponding to the first point or the one corresponding to the last point should have been executed in a time interval less than $c_i - \gamma_i$, which is not possible.
Therefore, by Lemma 1, task τ_i should have arrived N times during the busy interval.
(ii) This part follows from Lemma 3 immediately.

■

Corollary 1. Defining $\hat{c}_i \triangleq c_i - 2\delta_i - \gamma_i$, from Theorem 3,

(i) If $C(w_k)$ satisfies

$$(Np_i - \hat{c}_i)^+ \leq C(w_k) < Np_i + \hat{c}_i, \quad (13)$$

then task τ_i can only has arrived N times during the busy interval.
(ii) If $C(w_k)$ satisfies

$$Np_i + \hat{c}_i \leq C(w_k) < (N+1)p_i - \hat{c}_i, \quad (14)$$

then task τ_i can only has arrived N or $N+1$ times during the busy interval.

Recall that $N_k(\tau_i)$ is the number of times that task τ_i has arrived during the busy interval w_k. Therefore, for this busy interval, we have to find $N_k(\tau_i)$'s such that:

$$|\sum_i N_k(\tau_i)c_i - C(w_k)| \leq \sum_i (\gamma_i \cdot \max N_k(\tau_i)) \quad (15)$$

where by the theorem above, for each value $N_k(\tau_i)$, at most 2 values are possible, which are distant by 1.

D. Proof of Theorem 2

We partition the busy interval as follows:

$$[a,b] = [a, a+p_i] \cup [a+p_i, a+2p_i] \cup \cdots \cup [a+jp_i, b]$$

In the proof, we will use the fact that by the periodicity assumption, task τ_i arrives every p_i seconds. Therefore, if we know that there is no arrival at time t, then, $t \pm p_i$ also cannot be arrival times.
(i) If we know the exact value of $N_k(\tau_i)$, we exactly know in how many of the intervals of the partition above, arrival exists. There cannot be an arrival in $[a+jp_i - c_i, a+jp_i]$, otherwise, there should be an arrival in the interval $[a-c_i, a]$ and hence, the busy interval cannot start at a. Also, if we know that there is an arrival in the last interval of the partition, we can make the 1-intervals narrower. Note that there cannot be an arrival in $[b-c_i, b]$, otherwise, the busy interval cannot terminate at b. Therefore, there should be an arrival in $[a+(N-1)p_i, b-c_i]$. Therefore, by the periodicity assumption and by shifting $[a+(N-1)p_i, b-c_i]$ by integer multiples of p_i to the left and taking its intersection with other intervals of the partition, we will get the arrival windows. Therefore, we have (i).
(ii) Using Theorem 1, if $N_k(\tau_i) = N$ or $N+1$, then

$$Np_i + c_i \leq C(w_k) \leq (N+1)p_i - c_i.$$

Therefore, in the partition, $j = N$. We know there will be arrivals in first N intervals of the partition, but, we cannot say anything about the last interval of the partition. Therefore, we mark the last one as a 0-1-interval, with the consideration that similar to part (i), there cannot be an arrival in $[b-c_i, b]$. This gives us the second expression. Also, for other intervals of the partition, similar to part (i), there cannot be an arrival in $[a+jp_i - c_i, a+jp_i]$, for $j = 1, .., N$. This gives us the first expression.

E. Improvement of Theorem 2

The following theorem is the improved version of Theorem 2:

Theorem 4. Considering a task τ_i and a busy interval ω_k that has start time a and end time equal to b.
The partitioning of the busy interval is done by using the following equations:

(i) If τ_i has arrived exactly N times during w_k:

If $N = \left\lceil \frac{C(w_k)}{p_i} \right\rceil$, the following segments are *1-interval*:

$$\overline{A_k(\tau_i)_j} = [a+(j-1)p_i, b-(N-j)p_i - c_i] \qquad 1 \leq j \leq N \quad (16)$$

Else, the following segments are *1-interval*:

$$\overline{A_k(\tau_i)_j} = [b-(N+1-j)p_i, a+jp_i - c_i] \qquad 1 \leq j \leq N \quad (17)$$

(ii) If τ_i may have arrived either N or $N+1$ times during w_k: the following segments are *1-interval*:

$$\overline{A_k(\tau_i)_j} = [a+(j-1)p_i, a+jp_i - c_i] \qquad 1 \leq j \leq N \quad (18)$$

and the following segments are *0-1-interval*:

$$\overline{A_k(\tau_i)_j} = [a + (j-1)p_i, b - c_i] \qquad j = N+1 \quad (19)$$

where $\overline{A_k(\tau_i)_j}$ is the j^{th} arrival window for task τ_i in busy interval ω_k.
In both cases, the remainder of the busy interval is *0-interval*.

Proof: (i) First we note that by Theorem 1, If $N_k(\tau_i) = N$, then

$$(N-1)p_i + c_i \leq C(w_k) \leq (N+1)p_i - c_i.$$

We partition the busy interval as follows:

$$[a, b] = [a, a + p_i] \cup [a + p_i, a + 2p_i] \cup \cdots \cup [a + jp_i, b]$$

If $(N-1)p_i + c_i \leq C(w_k) \leq Np_i$, or equivalently, $N = \lceil \frac{C(w_k)}{p_i} \rceil$, then $j = N-1$ and there should be an arrival in each interval of the partition above. Also, there cannot be an arrival in $[b - c_i, b]$, otherwise, the busy interval cannot terminate at b. Therefore, there should be an arrival in $[a + (N-1)p_i, b - c_i]$.
By the periodicity assumption, task τ_i arrives every p_i seconds, i.e., if we have an arrival at time t, then, $t \pm p_i$ is also arrival times. So, shifting $[a + (N-1)p_i, b - c_i]$ by integer multiples of p_i to the left and taking its intersection with other intervals of the partition, we get (16).

If $Np_i \leq C(w_k) \leq (N+1)p_i - c_i$, then $j = N$ and there should not be any arrivals in the last interval of the partition above. Hence, because of the periodicity, the last arrival should be in the interval $[b - p_i, a + Np_i]$. But, If there is an arrival in interval $[a + Np_i - c_i, a + Np_i]$, then there should be an arrival in the interval $[a - c_i, a]$. Therefore, the busy interval cannot start at a. This implies that the last arrival should be in the interval $[b - p_i, a + Np_i - c_i]$.
Finally, because of the periodicity, by shifting $[b - p_i, a + Np_i - c_i]$ by integer multiples of p_i to the left and taking its intersection with other intervals of the partition, we get (17).
(ii) Using theorem 1 again, if $N_k(\tau_i) = N$ or $N+1$, then

$$Np_i + c_i \leq C(w_k) \leq (N+1)p_i - c_i.$$

Therefore, in the partition, $j = N$, and we cannot say anything about the last interval of the partition. So, we mark it as a 0-1-interval, with the consideration that similar to part (i), there cannot be an arrival in $[b - c_i, b]$. This gives as (19). Also, there should be an arrival in all other intervals of the partition, with the consideration that similar to part (i), there cannot be an arrival in $[a + jp_i - c_i, a + jp_i]$, for $j = 1, .., N$. This gives as (18).

■

F. Extra Example for Estimate of N_k

Example 5. Consider a busy interval ω_k with duration $C(\omega_k) = 16$ and a task set $\Gamma = \{\tau_1, \tau_2, \tau_3\}$ as follows:

	p_i	c_i	$N_k(\tau_i)$
τ_1	5	1	3 or 4
τ_2	17	6	1
τ_3	24	7	0 or 1

$^*C(\omega_k) = 16$

The above $N_k(\tau_i)$ values are estimated using the conditions of Theorem 1 (Equations (2) and (3)). For τ_1, Equation (3) holds when $N_k(\tau_1) = 3$ while Equation (2) does not, hence the value of $N_k(\tau_1)$ could be either 3 or 4 implying that τ_1 is likely to have arrived 3 or 4 times during the busy interval ω_k. Likewise, the estimation of $N_k(\tau_3)$ is similar to $N_k(\tau_1)$ except that τ_3 may have arrived 0 or 1 time. For τ_2, Equation (2) holds when $N_k(\tau_2) = 1$ and thus we can be sure that it arrived only once during $C(\omega_k)$. We can now use Equation (1) to find the combinations of $N_k(\tau_i)$ that can lead to the given busy interval duration $C(\omega_k)$ as shown in the following table:

$N_k(\tau_1)$	$N_k(\tau_2)$	$N_k(\tau_3)$	$C(\omega_k)$	
3	1	0	9	
3	1	1	16	$\surd$ matched
4	1	0	10	
4	1	1	17	

$^*C(\omega_k) = 16$

In this case, only the combination of $N_k = \{3, 1, 1\}$ satisfies Equation (1) for a busy interval length of $C(\omega_k) = 16$.

G. Compact Scheduling Translator

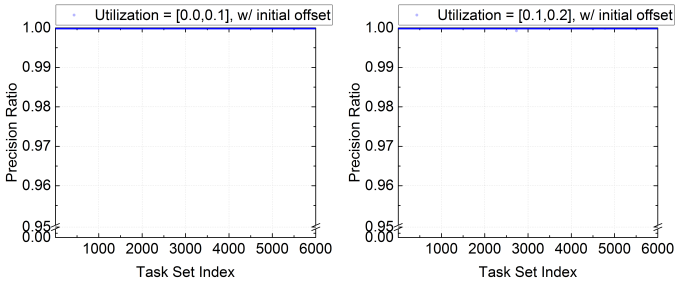
In order to reconstruct the schedule for a specified busy interval, a *compact scheduling translator* is used to convert arrival times of participating tasks to corresponding start times - the instant when a task actually starts running.

Similar to a regular fixed-priority scheduler, it applies the scheduling algorithm to organize the sequence and preemption of each job execution. However, in contrast to a real scheduler, it omits the real execution and does only the estimation of the start times. The translator starts with the first arrival in the busy interval and stops when there is nothing is to be scheduled in the simulation queue, which is equivalent to the close of the given busy interval. The detailed algorithm is presented in Algorithm 4.

Algorithm 4 Compact Scheduling Translator

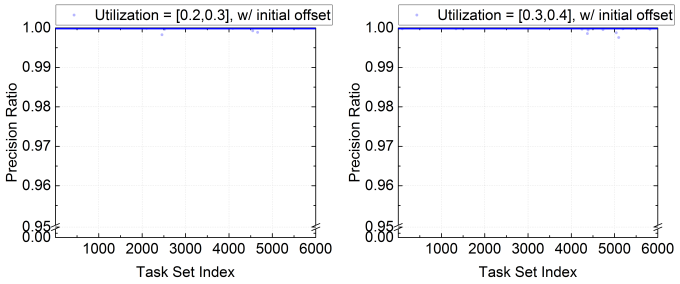
$\{A_k$: *arrival time array of* $\omega_k\}$
$\{S_k$: *task start time array of* $\omega_k\}$
$\{AQueue$: *arrival queue storing initial arrival times*$\}$
$\{SQueue$: *ready queue for suspended jobs*$\}$
$\{A_{this}$: *arrival time of current running job* $\tau_{this}\}$
$\{A_{next}$: *arrival time of next job* $\tau_{next}\}$
$\{c_{this}$: *remaining execution time of job* $\tau_{this}\}$
$\{E\&HP$: *"earliest and highest priority job"*$\}$

$AQueue.pushAll(A_k)$
$A_{this} \leftarrow$ *AQueue.pop(E&HP)*
$S_k.add(\{A_{this}, \tau_{this}\})$
while $AQueue$ *and* $SQueue$ *are not empty* **do**
  $A_{next} \leftarrow$ *{AQueue, SQueue}.pop(E&HP)*
  if $(A_{this} + c_{this}) > A_{next}$ **then**
    $c_{this} \leftarrow c_{this} - (A_{next} - A_{this})$
    SQueue.push$(\tau_{this}, A_{next}, c_{this})$
  end if
  if A_{next} *is from AQueue* **then**
    $S_k.add(\{A_{next}, \tau_{next}\})$
  end if
  $A_{this} \leftarrow A_{next}$
end while
return S_k



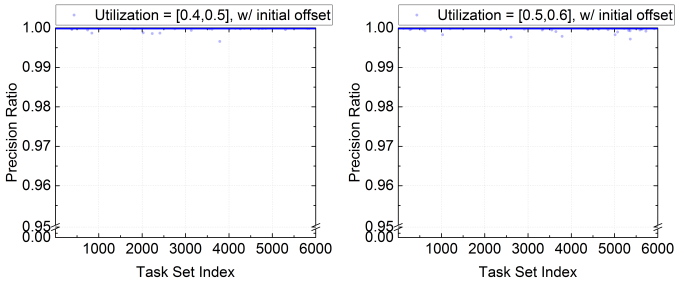
(a) Utilization = [0.0, 0.1]

(b) Utilization = [0.1, 0.2]



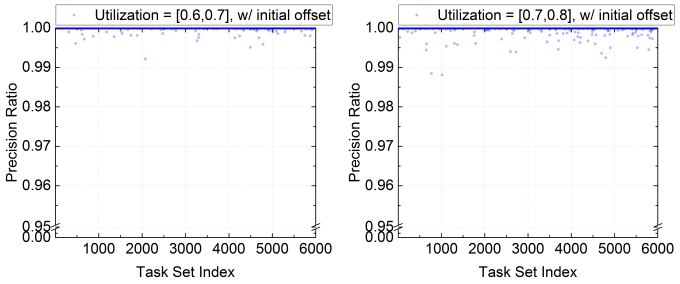
(c) Utilization = [0.2, 0.3]

(d) Utilization = [0.3, 0.4]



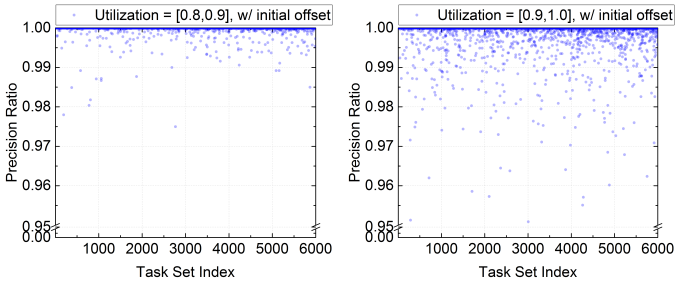
(e) Utilization = [0.4, 0.5]

(f) Utilization = [0.5, 0.6]



(g) Utilization = [0.6, 0.7]

(h) Utilization = [0.7, 0.8]



(i) Utilization = [0.8, 0.9]

(j) Utilization = [0.9, 1.0]

Figure 23: Zoom-in view of precision ratio in each utilization group in *Condition.C*.