On the Pitfalls and Vulnerabilities of Schedule Randomization against Schedule-Based Attacks

Nasri, Mitra; Chantem, Thidapat; Bloom, Gedare; Gerdes, Ryan M.

DOI
10.1109/RTAS.2019.00017

Publication date
2019

Document Version
Accepted author manuscript

Published in
Proceedings - 25th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2019

On the Pitfalls and Vulnerabilities of Schedule Randomization against Schedule-Based Attacks

Mitra Nasri[1], Thidapat Chantem[2], Gedare Bloom[3], and Ryan M. Gerdes[2]
[1] *Delft University of Technology, Netherlands*
[2] *Department of Electrical Engineering and Computer Engineering, Virginia Tech, USA*
[3] *Department of Electrical Engineering and Computer Science, Howard University, USA*

***Abstract*—Schedule randomization is one of the recently introduced security defenses against schedule-based attacks, i.e., attacks whose success depends on a particular ordering between the execution window of an attacker and a victim task within the system. It falls into the category of *information hiding* (as opposed to deterministic isolation-based defenses) and is designed to reduce the attacker's ability to infer the future schedule. This paper aims to investigate the limitations and vulnerabilities of schedule randomization-based defenses in real-time systems. We first provide definitions, categorization, and examples of schedule-based attacks, and then discuss the challenges of employing schedule randomization in real-time systems. Further, we provide a preliminary security test to determine whether a certain timing relation between the attacker and victim tasks will never happen in systems scheduled by a fixed-priority scheduling algorithm. Finally, we compare fixed-priority scheduling against schedule-randomization techniques in terms of the success rate of various schedule-based attacks for both synthetic and real-world applications. Our results show that, in many cases, schedule randomization either has no security benefits or can even increase the success rate of the attacker depending on the priority relation between the attacker and victim tasks.**

I. Introduction

Real-time systems are often designed to be predictable to simplify the worst-case execution time (WCET) and schedulability analyses, and to enforce deterministic runtime behaviors. This timing predictability, however, can be exploited by attackers to either directly influence a system's behavior and/or steal information, some of which can be used to increase the accuracy of future attacks [1]–[3]. Such an exploitation is called a *schedule-based attack*, where the success of the attack depends on a particular ordering between the execution window of the attacker and its targeted task [1]–[5]. For example, a cache-timing attack becomes more efficient and accurate if the attacker executes right before and after the execution window of its target [1,2]. Similarly, in the domain of *cyber-physical systems* (CPS) security, the success of a large number of attacks that focus on compromising data integrity (e.g., to deceive the control task, degrade the performance of the system, or damage the environment) depends on the execution window of the attacker and the time at which the system interacts with its physical environment. For example, *bias-injection attacks* [6], *zero-dynamics attack*, [6]–[11], and *replay attacks* [12] affect the output of the controller and hence must be performed *after* the targeted task completes, while *false-data injection* attacks [13]–[17] must execute *before* the targeted task accesses its input data.

Related work. There exist several defenses against schedule-based attacks among which *schedule randomization* is the main focus [1,3,5]. The goal of schedule randomization is to diversify the schedule frequently enough so that it becomes harder for the attacker to successfully guess when the targeted task is going to be executed. For example, Yoon et al. [1] introduced an online schedule randomization technique called *TaskShuffler* that schedules a randomly chosen task from the ready queue at each scheduling point. In order to guarantee deadlines, the authors first derived the slack of each task according to a fixed-priority scheduling policy, and then, at runtime, the scheduler steals these slacks in order to execute a randomly chosen (and potentially lower-priority) task. In contrast, Krüger et al. [3] used fine-grained slot-level slacks to provide more choices for the scheduler to select a random task at runtime. The slot-shifting algorithm [18] was leveraged for this purpose. Krüger et al. [3] also suggested an offline solution that is based on pre-storing a set of randomly generated offline schedules. Then, at runtime, the system non-deterministically selects among these schedules after each hyperperiod.

Randomization falls into the category of *information hiding*, where the security defense is based on hiding crucial/critical information from the attacker by means of *probabilistic pseudo-isolation* (instead of a strong deterministic isolation) [19]. It has been primarily used in *address-space layout randomization* (ASLR) for user- and kernel-space memory protection [20]–[23] to defend the system against memory-corruption attacks such as buffer overflows [24], format string exploits [25], double-free attacks [26], etc. It is also used in control-flow randomization [27] as a defense against code-reuse attacks.

It has been shown, however, that defenses that are based on randomization (information hiding) can be broken easily and efficiently, regardless of the size of the hidden objects or the randomization *entropy* (i.e., the degree of uncertainty in the random variables). For example, various types of ASLR have been successfully brocken by leveraging *allocation oracles* [28] and memory-disclosure vulnerabilities such as cache side channels [29]–[31]. Trilla et al. [32] showed that cache randomization solutions do not protect against side-channel attacks in time-critical systems. The following surveys introduce a large number of successful and efficient

penetrations to randomization-based defenses: [19,28,33] (for ASLR) and [30,34,35] (for control-flow randomization). Similar to this body of work, this paper discusses the limitations of schedule randomization-based defenses for real-time systems and evaluates their effectiveness against various types of schedule-based attacks.

This paper. Following the work on schedule-based attacks [1]–[3,36], we assume that the attacker has taken advantage of the existing vulnerabilities in an untrusted task (e.g., a third-party or an open source application) to hijack a task (called the attacker task) in order to steal information from or to influence the performance of another task (called the victim).

The paper starts with the system and threat model (Sec. II), followed by definition, categorization, and examples of schedule-based attacks in the context of uniprocessor real-time systems (Sec. III). We then discuss the pitfalls and limitations of schedule randomization methods as a security defense in real-time systems (Sec. IV). We show that the existing schedule randomization techniques may even increase the success rate of certain types of attacks since they are oblivious to the potential attacks and system vulnerabilities. For example, a false-data injection attack may not be possible when the system is scheduled using a fixed-priority scheduling policy if the victim has a higher priority than other untrusted tasks. however, schedule randomization may considerably increase the chance that the attacker can manipulate the sampled data stored in an I/O device buffer *before* the victim task accesses the data.

To provide a better understanding of security vulnerabilities of the fixed-priority scheduling against schedule-based attacks, we provide a preliminary security test that determines whether a certain attack (i.e., a certain timing relation between two tasks) can happen in the system (Sec. V). Finally, we evaluate the existing schedule randomization methods and the proposed security test against various types of attacks and priority relations between the attacker and the victim tasks (Sec. VI).

II. System and Adversary Models

To provide a better understanding of the vulnerabilities of schedule randomization techniques and fixed-priority scheduling, we developed system and threat models inspired by the adversary case studies of [2] and the schedule-based attack scenario of [5]. As [2,5] do not provide a detailed system model, we sought to specify the least complex, common system configuration and weakest attacker that could realistically enable the attacks described therein (i.e., timing-based side-channels and actuation attacks), in addition to like threats (e.g., false-data injection) stemming from the underlying system configuration that are common to CPS [37]. We will use this attack model as a basis to provide a preliminary security test for fixed-priority scheduling (in Sec. V) and to evaluate the effectiveness of schedule randomization techniques (in Sec. VI).

System model. We consider a general system that is capable of receiving external inputs and performing computations over those inputs to arrive at a decision that results in an output that is acted upon by an external agent. Additionally, the system may make use of sensitive information during the reception and/or transmission of inputs/outputs. The system makes use of peripherals to obtain inputs and produce outputs. An instance of our general system would be a real-time system for industrial process monitoring; e.g., the Tennessee Eastman chemical process [38]. In these systems individual tasks are responsible for obtaining and filtering data from sensors about the state of the process (reception of external inputs), determining the proper response to meet process objectives (computations over inputs), and issuance of commands to actuators to control the process (action taken by external agent).

Specifically, we envision a system wherein: **(i)** a dedicated peripheral (e.g., an analog-to-digital converter) is interfaced with process monitor sensors (e.g., fluid level sensors) and periodically reports the values of the sensors, which are then filtered by an input task, using for example a moving average filter or Kalman filter, to remove noise; **(ii)** the resulting filtered data is written to a memory location and then retrieved by a control task to determine appropriate actuation commands according to some control logic; and **(iii)** the actuation command is written to an output buffer, possibly by the control task, of a peripheral interfaced to an actuator (e.g., a pulse-width modulation peripheral that controls a servo on a valve). We allow that communication between the system and sensors/actuators could be digital and encrypted; thus, the tasks associated with acquiring/producing input/output data may use sensitive information (i.e., an encryption key).

Critically, a shared memory model is assumed that allows tasks to read and write to (data) memory locations that are used by all tasks. For example, a task that is responsible for relaying sensor information to a display terminal could read and write to the memory location(s) associated with filtered sensor data.

More generally, since the paper is focused on the limitations of schedule randomization methods, we follow the same real-time system model used in [1,3,5]. The system consists of n periodic tasks $\tau = \{\tau_1, \ldots, \tau_n\}$ scheduled upon a uni-processor platform. Each task τ_i is identified by a WCET, denoted by C_i, and a period T_i. The deadline of each task is equal to its period. We assume that the best-case execution time (BCET) of a task is an arbitrary non-zero value that is smaller than C_i. Tasks are indexed according to their periods so that $T_1 \leq T_2 \leq \ldots \leq T_n$.

A hyperperiod H is the least-common multiple of the periods. The utilization of a task τ_i is denoted by $u_i = C_i/T_i$ and the total system utilization is $U = \sum_{i=1}^{n} u_i$. Similar to [1,3], we assume that tasks do not have precedence constraints.

To ensure that attacks must happen with stringent timing accuracy, we assume a *logical execution time* (LET) paradigm; e.g., Berkley's Giotto architecture[1] [39,40] which is a time-triggered language and architecture for designing hard real-time control systems. The LET approach has a wide applicability in the automotive industry as it allows separation of the platform-independent concerns such as software functionality and I/O timing from platform-dependent concerns such as software

[1] https://ptolemy.berkeley.edu/projects/embedded/giotto/

scheduling and execution [41]. In particular, in Giotto and its successors [42,43], the system interacts with the I/O devices only at certain time instances such as task releases. This enables a jitter-free sampling and actuation and hence improves system's predictability. Giotto has been used in applications such as an autonomously flying model helicopter [44,45] and electronic throttle controllers [39,40] to mention a few.

Adversary model. We consider an adversary that has compromised a single task on the system and is in control of/able to modify the control flow of the task (i.e., during the execution window of the task under control, the attacker can run arbitrary code). We assume that the attacker cannot change the scheduling parameters of her/his task or the parameters of other tasks, hence, the attacker task must wait until being scheduled by the scheduling policy. We consider an attacker who can only observe the execution windows of the task under its control, and that is unaware of the scheduling policy being used. In Sec. III we define such an attacker as a WEAKEST and DEFENSE UNAWARE attacker.

With respect to the system model, an attacker can have one of three goals: **(i)** modify data at a memory location before it is read by another task (ANTERIOR attack); **(ii)** modify data at a memory location after it has been written by another attack (POSTERIOR attack); or **(iii)** access data at a memory location both before and after a task modifies data at the location (PINCER attack). We define the ANTERIOR, POSTERIOR, and PINCER attacks formally in Section III. An example of an ANTERIOR attack would be a false-data injection attack [13,14,16,17] against the sensing task described above, while an ANTERIOR attack could take the form of a zero-dynamics actuation attack [10] against the control task above. Certain cache-style attacks [46] can be seen as instances of the PINCER attack and could be useful in recovering sensitive information that the task would not otherwise have access to. For example, assume that an encryption key is kept in inaccessible memory until needed by a task (as per above). The PINCER attack could be used by the attacker to write to the temporary location before the key is written there and then query the same location after it is removed to infer key bits.

Our ANTERIOR attack is successful so long as it is performed anytime *after* the arrival of the victim task, where the sampled data becomes available and *before* the start time of the victim task, where it reads the data from the memory. Our POSTERIOR attack is successful as long as it is carried out *after* the victim's completion and before the actuation command is transmitted to the actuator (which happens synchronously at the victim task's deadline) or the physical plant has the opportunity to respond to the command. Our PINCER attack is successful if the attacker is successful in landing both the ANTERIOR and POSTERIOR attacks on the same job of the victim task. Namely, it must be executed between the release and start time of the victim task as well as between the completion and deadline of the victim task. For the rest of the paper, we will make the pessimistic, i.e., weakest, assumption that an opportunity for an attack always results in a successful attack.

III. SCHEDULE-BASED ATTACKS

While schedule-based attacks have gained a significant amount of attention by the real-time systems community in the past couple of years, there still lacks a clear definition for what it means for an attack to be successful, e.g., the type of timing relation between the execution windows of an attacker task and a victim task that is considered to be *harmful*. For example, an attacker, whose goal is to manipulate the output of a control task by overriding the data in the I/O device buffer, is successful only if it is scheduled *after* the victim task writes the outputs in the buffer and *before* the I/O device pulls the new data from the buffer.

A formal definition of an attack is a fundamental step for designing a defense mechanism and proving its success. For example, the attack just described can be provably avoided by a schedule-based defense mechanism that does not allow any *untrusted* task to be scheduled after the victim task and before the I/O device pulls the data from the buffer.

Schedule-based attacks are attacks whose success depends on a particular timing relation between the execution windows of the attacker and victim tasks. Hereafter, we use *victim* to refer to a task that is targeted by the attacker and *attacker task(s)* to refer to the task(s) that are already hijacked by the attacker. In order to hijack a task, we assume that the attacker has already taken advantage of the existing vulnerabilities in software provided by third-party vendors or open source codes as in existing work [1]–[3,36]. In the rest of this section, we formally define schedule-based attacks and characterize an attacker's ability and knowledge according to what has been proposed in the state of the art. Depending on the desired timing relation between the attacker's and victim's execution windows, schedule-based attacks can be categorized into four groups: POSTERIOR, ANTERIOR, PINCER, and CONCURRENT attacks.

Definition 1. *A* POSTERIOR *attack is an attack that must be performed* after *the victim task completes its execution.*

Examples. In control security, a large number of attacks have focused on manipulating the outputs of a control task before it is applied to the physical plant [6]. These include *bias-injection attacks* [6], *zero-dynamics attack*[2] [6]–[11], *replay attacks* [12], etc. For example, a zero-dynamics attack happens when the attacker generates outputs that maliciously disguise as the unstable zero dynamics of the plant [8]. Zero-dynamics vulnerability is created when the widely used *sample and hold* mechanism is employed to convert an analog sample to a digital one [10], and hence, exists in a large number of control systems. These attacks, however, are very hard (or in many cases, provably, impossible) to detect [10]. The following studies provide a categorization of detectable and undetectable (stealthy) zero-dynamics attacks [7,10,47].

Chen et al. [2] provided an example of a POSTERIOR attack in real-time embedded systems: a rover robot that is manually controlled by a remote controller. Here, the wifi-receiver task

[2] https://www.youtube.com/watch?v=rqE9lewmRTk shows a video of zero-dynamics actuation attack [10].

is the victim and the goal of the attacker is to override the commands that the wifi-receiver task writes to the I/O device buffers before the data is pulled by the wheel controllers and applied to the wheels[3].

Schedule-based attacks usually have an event-based deadline, i.e., a time frame during which they must happen or they are ineffective otherwise. For example, the attacker mentioned earlier must execute before the data produced by the victim is used in any part of the system. Such timing constraint for the attacker usually depends on the underlying software and hardware platforms and varies from system to system.

Definition 2. *An* ANTERIOR *attack is an attack that must be performed* before *the execution of a victim task.*

Examples. *False-data injection* attacks are a widely studied class of ANTERIOR attacks that compromise data integrity by manipulating the inputs of a control system [13,14]. Recently, it has been shown that a large number of false-data injection attacks can be theoretically stealthy [15]–[17]. This confirms the importance of having a proactive defense against these attacks in order to eliminate the attack before it can potentially occur rather than having a passive defense that *reacts* towards a detected attack, since many false-data injection attacks are too difficult to detect. ANTERIOR attacks can also target timing constraints of a real-time system [3], e.g., by creating interference on shared system resources such as caches.

Definition 3. *A* PINCER *attack is an attack that must be performed* before *and* after *a victim task, e.g., to observe (or pre-load) a side channel before the victim is executed and probe it afterwards. The time interval between observing and probing the side channel is called the* attacker's net.

Examples. Some examples of PINCER attack have been proposed [1,2]. Chen et al. [2] designed a spyware whose goal is to find the locations at which the system (a drone in this case) takes high-resolution pictures. The attacker has hijacked a task that can access GPS data, but which has no access to the rest of the system except the cache. Since high-resolution images can drastically change the state of the cache, the attack is carried out by probing the cache before and after the imaging task is executed in order to detect large cache footprints. As soon as such a large footprint is detected, the attacker stores the current GPS location[4].

Definition 4. *A* CONCURRENT *attack is an attack that must be performed while the victim task is running. Depending on the attacker's goal, this can be equivalent with executing between the execution windows of a job of the victim task.*

Examples. A CONCURRENT attacker might have several goals, for example, s/he might want to get the energy profile of various parts of a victim task so that s/he can have a fine-grained understanding of the functionalities performed by the task and the order by which they happen. As demonstrated by Delimitrou et al. [48] and Nathuji et al. [49], such an understanding allows the attacker to maximize the interferences that s/he can cause on shared system resources in order to reduce the performance or quality of service of the victim task, or to cause a large timing jitter (e.g., sampling and actuation jitters) for a victim task. In safety-critical control tasks, these sampling and actuation jitters reduce the quality of service to the extent that the system may even become unstable [50].

In the rest of this section, we introduce some essential properties of a schedule-based attacker that must be clarified when defining an attack model.

Attacker's prior knowledge. Yoon et al. and Krüger et al. [1,3] assumed that the attacker has *full knowledge* of the task set including tasks' worst-case execution times (WCET) and periods. Generally speaking, according to Kerckhoff's principle [51], the attacker can access any information about the system's parameters, architecture, and defense mechanism being used, e.g., by acquiring and then reverse engineering an instance of the system being attacked. Hence, we define a DEFENSE AWARE attacker as an attacker who knows the online schedule randomization method being used as well as the offline schedules that are stored in memory.

However, the security defenses proposed by Yoon et al. and Krüger et al. [1,3] are crucially based on hiding the randomization method and the offline schedules. We call such attacker a DEFENSE UNAWARE attacker[5].

Attacker's abilities to infer schedule-related information at runtime. Some existing work [1,3] assumes that the attacker can take control of some tasks in the system, hence, the attacker is an *insider*. These work, however, do not provide a clear and uniform description of the attacker abilities to infer schedule-related information. For example, Yoon et al. [1] state that *"We do not make any specific assumptions on the attackers ability to infer task schedule and to pinpoint the victim task(s). The attacker may even have an ability to deduce the exact schedule"*. Our interpretation from this explanation is that the attacker knows what has been scheduled in the past, but does not know what will be scheduled in the future (and that is why it makes sense to have an online randomization solution). We call such attacker a STRONG attacker and assume that it can observe the contents of the ready queue and knows the remaining execution budget of the tasks. In practice, any operating system that does not provide memory separation, such as the OSEK family, can easily reveal such information to the attacker (e.g., see Fig. 2 in Sec. IV).

Krüger et al. [3] assume that the attacker cannot access kernel memory space. However, since the attacker is in the system, it can at least observe its own execution windows. Depending on how many tasks are in the control of the attacker, we consider two other attacker types: the WEAKEST and WEAK attackers. The WEAKEST attacker is in control of only one task in the system and can only observe its own execution windows. The WEAK attacker controls multiple tasks in the system, but can

[3] https://youtu.be/xVclU4rthOM shows a video of a POSTERIOR attack [2].

[4] https://youtu.be/27zmJD0jMbM shows a video of a PINCER attack.

[5] We believe that hiding the architecture from the attacker by removing the possibility of reverse engineering is not a realistic way to achieve security.

only observe their execution windows. These tasks can either directly communicate with each other or use covert channels.

Attacker's ability to change task parameters. Except for separation kernels, e.g., ARINC-653, an RTOS does not control the activation pattern of a task. Instead, releases occur according to the state of the system a task controls. Periodic control loops are often implemented by an initial call to the RTOS to set a timeout incident to the next period (for implicit deadline) at the start of the job, and the job ends with an RTOS call to sleep until a timeout is reached. If the timeout fires while the task is still executing, then a budget overrun is detected. Aperiodic and sporadic tasks either program a timer with their next release, or the next release is triggered by an external interrupt in an event-driven manner. In either case, tasks can manipulate their own activation patterns, e.g., by changing arguments to timer calls.

The state-of-the-art schedule randomization defenses [1,3] are crucially based on having periodic behavior and no execution-time overrun because otherwise they cannot guarantee deadlines. These defenses assume that the system executes tasks within a reservation server which is periodically activated by the trusted part of the system. Hence, the attacker cannot influence its task's parameters or overrun its budget. Consequently, the attacker has a limited ability (if any) to affect its own execution windows at runtime.

IV. Limitations, Vulnerabilities, and Challenges of Schedule Randomization

This section discusses limitations of schedule randomization-based defenses as well as the design challenges that they introduce to real-time systems.

A. Limitations and Vulnerabilities

Attack-oblivious defense. Due to the size, weight, and power (SWaP) constraints, embedded systems often do not have state-of-the-art hardening techniques that are designed for servers or personal computers. As a result, they may be more vulnerable against certain types of attacks. For example, a system may use cache partitioning to provide a strong cache isolation (e.g., for timing predictability requirements), however, it may not apply an access-control policy on the I/O device buffers, e.g., due to performance requirements. While this exemplary system is strong against PINCER cache-side channel attacks, it is vulnerable against ANTERIOR and POSTERIOR attacks on data integrity. To defend such a system against the latter attacks, one can, for example, design a scheduler that does not schedule an untrusted task before a trusted task accesses its data. However, since the existing schedule-randomization techniques [1,3] are oblivious to the potential attacks and system vulnerabilities, they cannot secure a particular system and, hence, cannot eliminate the attacks (as shown in Sec. VI).

Opening Pandora's box. Due to their probabilistic nature, schedule randomization defenses raise the success rate of certain types of attacks. In fact, they may even allow a schedule-based attack that would have been impossible when using a fixed-priority scheduling policy. For example, in a periodic task set shown in Fig. 1-(a) scheduled by fixed-priority scheduling policy, τ_3 does not have any chance to be directly executed before or after its target task τ_1. However, a randomized schedule such as that in Fig. 1-(b) leads to successful ANTERIOR, PINCER, and CONCURRENT attacks. Our experiments in Sec. VI show that, for example, while there is no successful instance of an ANTERIOR attack when the system is scheduled by the rate-monotonic scheduler and the victim is the highest-priority task, the TaskShuffler scheduler [1] causes 20% of the jobs of the victim task to be affected by the ANTERIOR attack.

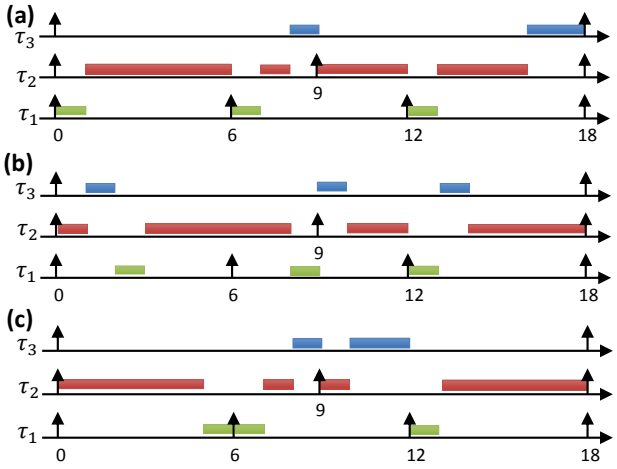

Fig. 1. Tasks $\tau_1 = (1, 6)$, $\tau_2 = (6, 9)$, and $\tau_3 = (3, 18)$, where $\tau_i = (C_i, T_i)$ is a periodic task whose WCET and period are C_i and T_i, respectively. **(a):** Fixed-priority (rate-monotonic) schedule, **(b,c)**: randomized schedules.

Limited choices. Unlike other randomization-based defenses such as ASLR, schedule randomization is applied on two dimensions, i.e., time and tasks. However, since these two dimensions are dependent on one another due to the timing requirements and constraints of the system (i.e., a task has a bounded execution time that must be completed by its deadline), the space of feasible choices of an online schedule randomization method is limited to a subset of pending tasks.

Furthermore, the scheduler's prior decisions will affect its future decisions and may result in situations where there is only one (or few) feasible choice(s) left. This creates a large vulnerability surface, which makes it easier for the attacker to predict the execution window of its victim. For example, in Fig. 1-(c), the attacker (τ_2) knows that the victim (τ_1) has no other choice but to be scheduled at time 4 or miss its deadline.

Schedule randomization may not prevent opportunistic schedule-based attacks. As mentioned in Sec. III, some data-integrity attacks are provably stealthy, e.g., they cannot be distinguished from noise. Hence, the only way to deal with them, when using a model-based detection approach, is to use a deterministic solution to guarantee that these attacks cannot happen. Another example is the opportunistic cache-based attacks that may not even need an accurate guess about the execution window of their target as long as other tasks that are scheduled in the attacker's net do not have a large

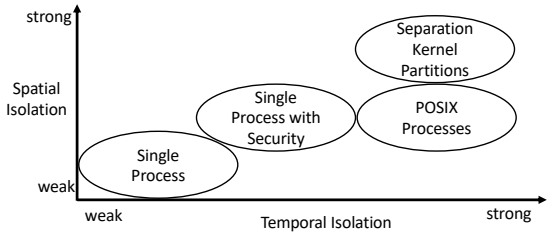

Fig. 2. The RTOS design space covers a spectrum of solutions for providing isolation between tasks in space (memory) and time (processing).

cache footprint. This, for example, happen in the scenario of the PINCER attacker in [2].

Isolation may prevent attacks easily and efficiently. A basic premise of schedule-based attacks is that an attacker knows when a specific victim task will execute. This knowledge may be prevented simply by providing strong temporal and spatial isolation between the attacker and the victim. As shown in Fig. 2, different RTOS designs provide a spectrum of isolation from weak isolation, which is trivial to overcome by an adversary, to strong isolation that requires compromising the kernel to violate. Spatial isolation, i.e., memory protection and access control, can prevent direct attacks on data by precluding access to I/O buffers or sensitive controller registers except to tasks that require such access as suggested by the principle of least privilege. All but the simplest single-process RTOS are capable of enforcing sufficiently strong enough spatial isolation to effect such access control.

That is, the types of attacks schedule randomization is meant to guard against largely concern preventing an adversary in control of one task from accessing (i.e., reading and/or writing) another task's data within a given time window (see Sec. II). Traditional techniques to prevent unauthorized access to data, i.e., memory isolation, have been eschewed in limited resource real-time systems due to overhead concerns [52]. Increasingly, however, 32-bit microcontrollers come equipped with integrated memory protection units (MPU) [53,54] with low enough overhead to be used in real-time applications [55]–[57].

The MPU integrated with ARM Cortex-M3 and higher microcontrollers[6], for example, can accomodate memory isolation for real-time tasks [58]. Specifically, in ARM parlance memory can be assigned to a *region* by the MPU, which controls how that memory is (or is not) accessed [59]. Memory isolation could be achieved for tasks by defining a region to cover all memory locations associated with task data and deny all access by default (the memory locations are assumed to be contiguous, for ease of exposition). Upon switching tasks the memory associated with the new task would be assigned a new region that allowed for read and/or write access (Fig. 3). The total number of instructions necessary to load the address of a task's memory and update the MPU is somewhat implementation specific but can be accomplished in as few as thirteen instructions [60].

Therefore, with respect to spatial isolation, schedule randomization should only be considered as potentially applicable

[6]Though the MPU is optional for this device family, it is extremely popular.

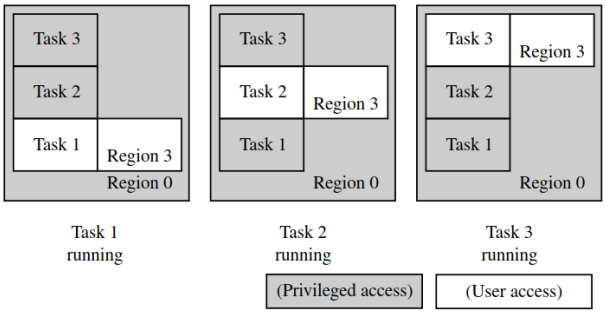

Fig. 3. Memory isolation for real-time systems using an MPU [59]: Region 0, an access policy, encompasses all tasks' data and disallows all access. Before execution of a task, its data is switched to Region 3 by the MPU, which allows read/write access; i.e., each task's memory is unlocked before execution and locked after execution. The settings of Region 3 take precedence over Region 0 so the MPU only needs to adjust a single access policy (Region).

for systems in which it is impossible to organize task data to accomodate the MPU or the system lacks an integrated MPU (typically 8- or 16-bit microcontrollers). It should be noted that opensource cores for 8-bit microcontrollers have been supplemented with efficient MPU-like capabilities [61].

Side-channel attacks may still be feasible despite spatial isolation, but strong temporal isolation can reduce the effectiveness of side-channels because the attacker has limited windows of opportunity to observe the side-channel information. The strongest temporal isolation occurs naturally with an RTOS that uses a separation kernel, e.g., hypervisor solutions such as PikeOS or avionics RTOSs with ARINC-653 partitions like Deos or VxWorks653. Increasingly, safety-critical RTOS adopt the separation kernel partitioning scheduler approach to improve fault tolerance. Partitions are scheduled by the kernel, typically from a static scheduling table or with static time slices, and each partition schedules its own tasks. Often, the scheduler inside a partition is a fixed-priority preemptive scheduler, but other scheduling algorithms are possible to use internally to the partition.

There is no valid security metric to evaluate schedule-randomization defenses. While there exist several metrics to measure information leakage [62,63], only one metric has so far been used in real-time systems. Yoon et al. [1] introduced *schedule entropy* to quantify schedule uncertainty against schedule-based attacks. Schedule entropy is based on Shannon entropy [64] in information theory, which describes the amount of information (or uncertainty) in a random variable, or equivalently, the uncertainty that a specific outcome actually occurs. For a random variable X that can take values $\{x_1, x_2, \ldots, x_m\}$, Shannon entropy is $H(X) = -\sum_{i=1}^{m} P(x_i) \cdot \log_2(P(x_i))$, where $P(x_i)$ is the probability that the random variable X takes value x_i. For example, the entropy of a random variable that represents tossing an unbiased coin is 1, because we have no information that helps us to guess the final outcome. However, if the coin is biased and has 90% chance to be a *head*, then the entropy reduces to 0.46 because now we have more information about the outcome.

Yoon et al. [1] defined schedule entropy as the uncertainty in the schedule of one hyperperiod. It is obtained from the *joint*

Time slots	Schedules											
	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}	S_{12}
1	τ_1	τ_1	τ_1	τ_3	τ_3	τ_3	τ_2	τ_2	τ_2	τ_2	τ_2	τ_2
2	τ_2	τ_2	τ_3	τ_1	τ_2	τ_2	τ_1	τ_1	τ_2	τ_2	τ_3	τ_3
3	τ_2	τ_3	τ_2	τ_2	τ_1	τ_2	τ_2	τ_3	τ_1	τ_3	τ_1	τ_2
4	τ_3	τ_2	τ_2	τ_2	τ_2	τ_1	τ_3	τ_2	τ_3	τ_1	τ_2	τ_1

Fig. 4. All 12 random schedules that can be generated for three tasks $\tau_1 = (1, 4)$, $\tau_2 = (2, 4)$, and $\tau_3 = (1, 4)$.

entropy of a set of random variables $S = (S_1, S_2, \ldots, S_L)$ for L equal-length time slots in the hyperperiod. These random variables are called *slot variables* and their domain is the set of task indices, i.e., $\{1, 2, \ldots, n\}$. Namely, $S_t = i$ means that task τ_i is executed at time slot t.

Definition 5. ***(Definition 4 from [1]: schedule entropy)*** *The schedule entropy of a task set τ is the Shannon entropy of the distribution of the hyperperiod schedules $\mathcal{S}$. Hence,*

$$H_\tau(S) = -\sum_{s_0=1}^{n}\sum_{s_1=1}^{n}\cdots\sum_{s_{L-1}=1}^{n} P(s_0, s_1, \ldots, s_{L-1}) \times \log_2\big(P(s_0, s_1, \ldots, s_{L-1})\big), \quad (1)$$

where $s_i \in \{1, 2, \ldots, n\}$ denotes the index of a task being executed at the i^{th} time slot of the hyperperiod, and $P(s_0, \ldots, s_{L-1})$ is the probability mass function of the schedule. The summand is 0 if $P(s_0, \ldots, s_{L-1}) = 0$.

Schedule entropy, however, is *not* a security metric for schedule-based attacks since it does not take the *attack*, *attacker's goal*, and *attacker's partial observations about the system* into account. A security metric must quantify the success of a defense mechanism against the attack being considered. Next, we show how drastically the attacker's uncertainty changes when considering a particular attack.

Example 1. *Consider the task set in Fig. 4 and an* ANTERIOR *attacker who has hijacked τ_1 and wants to be scheduled right before τ_2. In the first hyperperiod, the attacker will be successful if any of the three schedules $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_4$ happen. Hence, the attacker's success probability will be $\frac{3}{12} = 0.25$. Now if we derive the Shannon entropy of a random variable that shows attacker's success, it will be $-\big(0.25 \cdot \log(0.25) + 0.75 \cdot \log(0.75)\big) = 0.81$ (since the probability that the attacker fails is 0.75). However, according to Equation* (1)*, the schedule entropy of this example is 3.58 which is more than four times larger than the attack-aware entropy. In other words, the schedule entropy creates an illusion of security.*

The second concern about the schedule entropy is that it does not account for the attacker's partial observations about its execution windows. When the outcome of a random variable provides information about the outcome of another random variable, one must use *conditional entropy* instead of a simple Shannon entropy. For example, if there are two boxes and two balls that can go to either of these boxes, knowing what has been put in the first box allows predicting which ball will go in the second box. The following example shows how attacker's observation changes the game of schedule uncertainty.

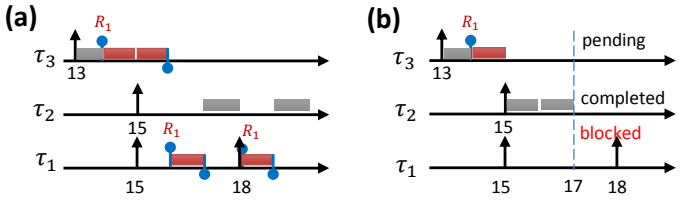

Fig. 5. Task set $\tau_1 = (1, 3)$, $\tau_2 = (2, 7.5)$, and $\tau_3 = (3, 13)$, scheduled by: **(a)** fixed-priority scheduling policy and priority inheritance protocol and **(b)** randomized scheduling policy. In this example, τ_1 requires resource R_1 from its start time while τ_3 requires R_1 one unit of time after its start.

Example 2. *Assume the* WEAKEST *attacker (that is in the possession of τ_2) whose goal is to predict the execution window of τ_1 in a schedule shown in Fig. 1-(c). Since the attacker has observed its own schedule from time 0 to 5, it accurately predicts that the first job of task τ_1 will be scheduled in the interval $[5, 6]$ since otherwise τ_1 will miss its deadline. Similarly, at time 13, where the attacker is activated, it knows that it has been certainly scheduled right after the third job of τ_1 because it knows that it still has 5 units of execution time that must be scheduled before time 18.*

In this example, the schedule entropy according to Definition 5 is 18.96 while the conditional probability $P\big(S_5 = 1 \mid (S_0 = 2, S_1 = 2, S_2 = 2, S_3 = 2, S_4 = 2)\big) = 1$ and hence the conditional entropy of S_5 is zero in this case. Namely, there is no uncertainty in predicting what will be scheduled at time 5.

B. Incompatibility with Real-Time Systems

Inflating WCET and CRPD. Schedule randomization drastically increases WCET and CRPD of the tasks since it allows preemptions at any point in time. Furthermore, any combination of co-running tasks can happen at runtime which increases the WCET of a task due to interference on the shared resources, e.g., cache. Fine-grained schedule randomization can have the same effect on task's execution as having no cache in the system. Consequently, the WCET of the tasks calculated for schedule randomization methods will be much larger than the WCETs obtained for a deterministic scheduling policy.

Lack of seamless support for existing synchronization policies. The existing schedule randomization techniques assume that tasks are independent and do not share resources. Going beyond this assumption, however, initiates fundamental challenges for schedule randomization as randomizing the schedule can easily cause unbounded blocking times. The example in Fig. 5 shows that synchronization-oblivious schedule randomization can cause deadline misses and unbounded blocking. As shown in Fig. 5-(a), the task set is schedulable when priority-inheritance protocol is used since it guarantees that τ_1 will not be blocked more than 2 units of time. However, a randomized algorithm may preempt τ_3 at time 15, where the second job of τ_2 is released (Fig. 5-(b)). Since τ_1 is blocked at this time and since τ_3 has enough slack, the scheduler may select τ_2 to be scheduled for the next two time units. This, however, either results in a deadlock situation or a deadline

miss for τ_1. The former happens if the only criteria for selecting a random task is the priority-inversion budget calculated merely based on the WCETs and periods because, then, the priority inversion budget of τ_1 becomes 0 at time 17 and, hence, no other lower-priority task is allowed to be scheduled.

Incompatibility with isolation-based defenses. Since schedule randomization is designed to break down the predictability of a schedule, it significantly increases the cost of closing side channels with methods such as flushing the cache after task's execution [36,65,66].

Limited to task sets with fixed parameters. Since schedule randomization enforces priority inversions to tasks with urgent deadlines, it requires that other tasks in the system, including the hijacked ones, always behave well, i.e., assuming that these tasks do not overrun their WCET and do not change their period or deadline. Such assumptions hold only if the operating system is able to support reservation-based scheduling and does not allow user-level tasks to add new tasks/processes to the system or change their activation frequency.

Incompatibility with sporadic tasks. A sporadic task may release a job when there is no other task in the system. This significantly reduces the space of possible choices that are available for schedule randomization. If the attacker keeps generating jobs as frequently as possible, then it will be able to pinpoint its victim task as soon as it finds a situation where there is no other pending task in the system. Hence, opportunistic attackers and/or STRONG attackers can easily break through the schedule randomization when the target task is sporadic.

V. A SECURITY TEST FOR FIXED-PRIORITY SCHEDULING

This section provides a preliminary security test for task sets scheduled by the fixed-priority scheduling. The test allows evaluating whether a given victim task $\tau_v \in \tau$ can be attacked by a potentially untrusted task $\tau_a \in \tau$. We design the test particularly on the *adversary model introduced in Sec. II*. The following lemmas provide a set of conditions for the possibility (or impossibility) of ANTERIOR, POSTERIOR, and PINCER attacks occurring in fixed-priority scheduling. Intuitive proofs have been omitted.

Lemma 1. *A lower-priority attacker task can never perform an* ANTERIOR *attack on a higher-priority victim task.*

Lemma 2. *An* ANTERIOR *attack is always successful if the attacker has a higher priority than the victim and its period divides victim's period.*

Proof. Since the victim job is always released together with a job of the attacker task and since the attacker has a higher priority than the victim, it is always scheduled between the release time and start time of the victim. □

Lemma 3. *Task τ_a always performs a successful* PINCER *or* POSTERIOR *attack on any job of a victim task τ_v if τ_a has a higher priority than τ_v, T_a divides T_v, and $R_v^w < T_v - T_a$, where R_v^w is the worst-case response time (WCRT) of task τ_v.*

Proof. Since τ_a is always released together with τ_v and since τ_a has a higher priority than τ_v, at least one of its jobs is always scheduled between the release time and start time of task τ_v. Moreover, since $R_v^w < T_v - T_a$, every job of task τ_v, that is released at time t, certainly completes before the release of the latest job of τ_a that released at $t + (T_v - T_a)$ because $t + R_v^w < t + T_v - T_a$. Note that since the two periods are harmonic, during the interval $[t, t+T_v)$, exactly $k = \frac{T_v}{T_a} - 1$ jobs of τ_a are released at time instants $t, t+T_a, t+2\cdot T_a, \ldots, t+k\cdot T_a$. In other words, the k^{th} job of τ_a after time t is released at $t + (T_v - T_a)$. Since $t + R_v^w < t + T_v - T_a$, at least one job of τ_a will be scheduled *after* the completion of any job of τ_v and *before* the deadline of τ_v. Thus, τ_a lands both ANTERIOR and POSTERIOR and hence, PINCER attack on every job of τ_v. □

Lemma 4. *Task τ_a cannot perform a successful* POSTERIOR *or* PINCER *attack on any job of task τ_v if τ_a has a higher priority than τ_v, T_a divides T_v, and $R_v^b > T_v - T_a$, where R_v^b is the best-case response time (BCRT) of τ_v.*

Proof. Since the periods of the attacker and victim tasks are harmonic, any job of τ_v is released together with a job of τ_a. Similar to the proof of Lemma 3, assume that τ_v is released at time t, hence, the job releases of τ_a from time t happen at $t, t + T_a, t + 2 \cdot T_a, \ldots, t + k \cdot T_a$, where $k = \frac{T_v}{T_a} - 1$. Hence, the latest job of τ_a before time $t + T_v$ is released at $T_v - T_a$.

According to the assumption, $R_v^b > T_v - T_a$, hence, $R_v^b + t > T_v - T_a + t$, namely, the earliest completion time of the current job of τ_v will be later than the release time of the k^{th} job of the attacker, i.e., at $t + k \cdot T_a$. Since τ_v has a lower priority than the attacker, starting from time $T_v - T_a$, it cannot be executed unless τ_a completes. Since its BCRT is larger than $T_v - T_a$, its completion time must be larger than the completion time of the k^{th} job of τ_a. Hence, the attacker cannot execute *after* the completion and *before* the deadline of τ_v. This means that the attacker will not be successful in landing a POSTERIOR or a PINCER attack on any job of τ_v. □

The BCRT and WCRT of a task for fixed-priority scheduling can be calculated using various methods, such as Audsley's response-time analysis [67] or [68]. The correctness of Lemmas 3 and 4 does not depend on the accuracy of the method used to calculate the BCRT and WCRT of the tasks as long as the method is sound, i.e., the actual BCRT is larger than or equal to (and the actual WCRT is smaller than or equal to) what the response-time analysis method calculates.

Theorem 1. *A task set τ is immune to an* ANTERIOR *attack from the untrusted tasks $\tau^u \subset \tau$ if Lemma 1 holds for any two tasks $\tau_v \in \tau \setminus \tau^u$ and $\tau_a \in \tau^u$.*

Theorem 2. *A task set τ is immune to* PINCER *and* POSTERIOR *attacks from the untrusted tasks $\tau^u \subset \tau$ if Lemma 4 holds for any two tasks $\tau_v \in \tau \setminus \tau^u$ and $\tau_a \in \tau^u$.*

Similarly, one can use Lemmas 2 and 3 to build a test that determines whether an ANTERIOR (POSTERIOR) attack certainly happens in the system.

VI. Empirical Results

We conducted experiments to evaluate the success of existing schedule randomization defenses against the schedule-based attacks introduced in Sec. II. We considered a fixed-priority scheduler (with rate-monotonic priorities) as a baseline and compared it with three versions of TaskShuffler [1] with randomization on tasks (TS1), on tasks and the idle task (TS2), and a fine-grained randomization on tasks and idle times (TS3). We have also implemented the online schedule randomization proposed by Krüger et al. [3] which uses *slot shifting* (SS). Unfortunately, it was too slow to provide us any results for the Autosar-like task sets that we have generated in the experiments since scheduling decisions must be made for every time quantum. Hence, we limited the evaluation of SS to a case study.

It is worth noting that this paper only partially evaluates the schedule-randomization methods since currently there is no sound and accepted way to measure the *schedule uncertainty*. As mentioned in Sec. IV, the current schedule entropy is optimistic and does not capture the attacker's partial observations. We believe that a thorough evaluation of schedule randomization methods requires two further steps: first, an actual case study with an actual schedule-based attack that cannot be carried out without an *accurate schedule inference*, and second, fundamental theories that allow quantifying the uncertainty of schedule w.r.t. to a particular attack model. We leave these steps as future work and focus on evaluating the effect of schedule randomization on the *attack success ratio* (ASR) for various schedule-based attacks. The ASR is measured as the ratio of successful attacks (e.g., successful ANTERIOR) to the number of jobs of the victim task. It also represents the chance that a victim job is (positively) attacked.

Since the ASR depends on the timing properties of the attacker and victim tasks, different attacker/victim assignment scenarios result in different ASR. As a basis, we assume that tasks are prioritized by the rate-monotonic priority ordering and each task has a unique priority value (ties are broken arbitrarily but consistently). Namely, assigning the attacker and victim tasks is equivalent with assigning them to a priority level. For example, *v:HP a:LP* means that the victim and attacker are the highest- and lowest-priority tasks in the task set, respectively. The horizontal axis of Fig.6-(g) to (i) shows our scenarios for assigning the attacker and victim attacks, where *a* denotes the attacker's priority and *v* denotes the victim's priority. An attacker (victim) whose priority is denoted by *HP* (or *LP*) has the highest (the lowest) priority in the task set. Similarly, *randomHP*, *randomMP*, and *randomLP* mean that the priority of the target task is chosen randomly from $\{1, \ldots, \frac{n}{3}\}$, $\{\frac{n}{3}+1, \ldots, \frac{2n}{3}\}$, and $\{\frac{2n}{3}+1, \ldots, n\}$, respectively, where n is the total number of tasks in the task set. We then used the attacker's and victim's priority on our security test.

A. Simulation Results

Task set generation. We conducted experiments by generating periodic task sets following the guidelines set forth by Kramer et al. [69] from Automotive benchmark applications. Specifically, for a given number of tasks n and utilization U, we sampled the (non-uniform) distribution of common periods ($\{1, 2, 5, 10, 20, 50, 100, 200, 1000\}$ms) reported by Kramer et al. [69] to randomly draw a realistic period for each task. Then we used the *RandFixSum* algorithm [70] to generate random utilization u_i for each task (with a total sum U) to obtain C_i by $u_i \cdot T_i$. We considered two experiments where we varied U and n, respectively. For the first experiment **(ExpU)**, we considered $n = 10$ tasks and for the second experiment **(ExpN)**, we considered U to be between $[0.1, 0.3]$. Figs. 6 and 7 show the results of these experiments, respectively. For each data point in the diagrams, at most 1000 random task sets were generated and each task set was executed for 10 hyperperiods. The experiments were performed on a machine with 8-core Xeon processor, 32GB of RAM, and a 1TB SSD.

Overall observations. Our results show that randomizing the schedule does not eliminate ANTERIOR, POSTERIOR, or PINCER attacks as shown in Fig. 6. In average, the chance that ANTERIOR, POSTERIOR, and PINCER attacks successfully affect a victim job despite using one of the schedule randomization methods is about 38%, 60%, and 30%, respectively (see Fig. 6-(a) to (c)). Moreover, these algorithms do not perform noticeably better than the rate-monotonic scheduler. Namely, in most cases, they either have no or a very limited reduction in the attack success ratio. Furthermore, as shown in Fig.6-(d) to (f), the TaskShuffler algorithm even increases the ASR in comparison with RM scheduler. As we discussed earlier, some attacks (e.g., ANTERIOR) are impossible when RM is used and the victim has the highest priority.

In some other cases, e.g., *v:LP*, RM is totally vulnerable against ANTERIOR, POSTERIOR, and PINCER attacks (shown in Fig.6-(g) to (i)). However, even in these cases, the schedule-randomization methods are either inefficient (e.g., against POSTERIOR attacks shown in Fig.6-(h)) or do not considerably reduce the attack's success, e.g., against ANTERIOR and PINCER attacks shown in Fig.6-(g) and (i).

The effect of utilization. We observed that in most of our experiments, the task set utilization had only a limited effect on the attack success ratio (e.g., in Fig. 6-(a) to (c)). However, it did have a large impact on the *v:HP a:LP* scenario (as shown in Fig. 6-(d) to (f)) because when the attacker's task has a higher utilization, it has a larger execution interval and can be preempted more often. Consequently, its chance to be scheduled before one of the victim's jobs increases when a schedule randomization method is applied.

The effect of priority assignment. As shown in Fig. 6-(g) to (i), the relation between the attacker's and victim's priorities (and periods) plays a key role in the attack success ratio. For example, when the victim task has a lower priority than the attacker (e.g., *v:LP*), victim's jobs are more exposed to the attacker's jobs since the attacker has more jobs in a hyperperiod. For instance, the chance that the victim's job is affected by a POSTERIOR attack reaches 100% for all scheduling policies when the attacker is the highest-priority task (Fig. 6-(h)).

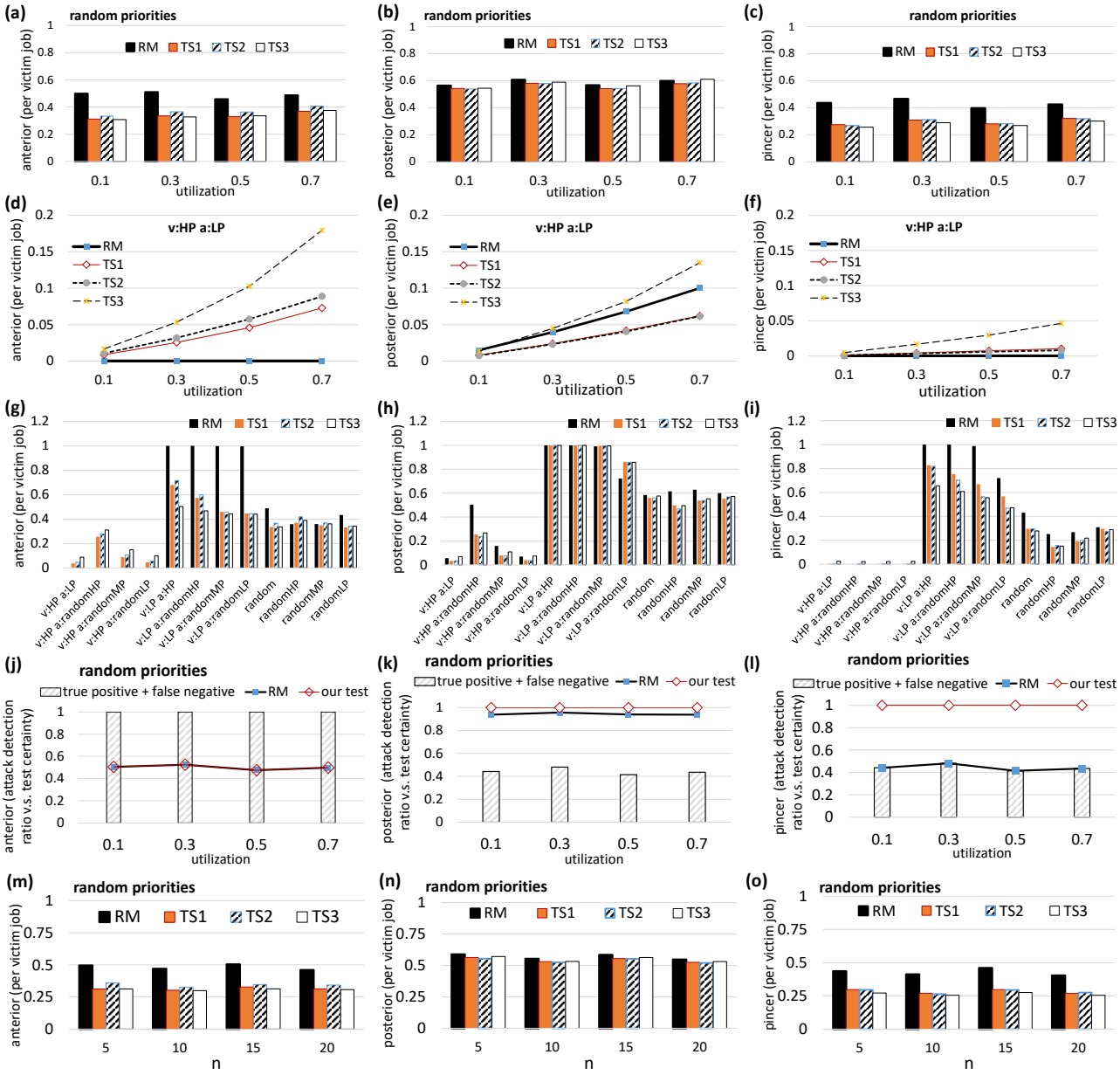

Fig. 6. **(a, b, c, d, e, f):** attack success ratio when the priorities of the attacker and the victim have been chosen randomly (a, b, c) or the victim is the highest-priority task and the attacker is the lowest-priority one (d, e, f) in ExpU, **(g, h, i):** attack success ratio for different priority relations between the attacker and the victim and different types of attacks in ExpU, **(j, k, l):** attack detection rate and coverage of our security test w.r.t. RM's result in ExpU, **(m, n, o):** attack success ratio when the priorities of the attacker and the victim have been chosen randomly in ExpN.

On the other hand, when the attacker task has a lower priority (larger period) than the victim task, it has fewer opportunities to land an attack. Hence, the system is less vulnerable against ANTERIOR and PINCER attacks (as shown in Fig. 6-(g) and (i)). Noticeably, RM has a promising performance in those cases since it does not allow any ANTERIOR or PINCER attack to happen as discussed in Sec. V. This shows the importance of analyzing the system before deciding to use a randomized schedule, as the system may already be strong enough against a certain type of schedule-based attacks.

Performance of the proposed security test. Fig. 6-(j)–(l) report the performance of our security test measured by *attack detection rate* (ADR), i.e., the ratio of task sets that deemed *not* to be secure (i.e., rejected by Theorems 1 or 2), and *test coverage* (TC), i.e., the number of task sets that we could certainly decide about their security or insecurity. Higher values of ADR represents higher attack success rate. For example, Fig. 6-(j) shows that with the confidence 1 (shown with the bar chart), we could identify cases where an ANTERIOR certainly happens (true positive) or it certainly does not happen (false negative). In that experiment, about 50% of the task set were immune to ANTERIOR attack and the other 50% were certainly prune to the ANTERIOR attack.

For POSTERIOR and PINCER attacks, however, we were only

TABLE I
THE TASK SET USED AS A CASE STUDY FROM [36].

Task Name	Worst Case Timing(ms)	Period(ms)
Software Control Tasks	2	20
Mission Planner	0.002	100
Encryption	3	42
Image Encoding	18	42
Image I/O	1.46	42
Network Manager	0.03	10

confident about 44% of the task sets. As shown in Fig. 6-(k), our test identifies all task sets to be insecure (i.e., the victim task in the task set can be successfully attacked by the attacker task). Yet, the test is not too pessimistic since the actual results from simulating RM schedules and counting the attacks is not very different from the result of the test. The error is in average about 5%. The error, however, is larger for PINCER attacks as they are less frequent to happen.

Effect of varying the number of tasks. As it can be seen in Figs. 6-(m)–(o), the ASR does not change much when the number of tasks varies. This is due to the fact that the period set used in the experiment is almost harmonic, hence, the number of tasks does not play a significant role in changing jobs' release pattern in a hyperperiod as they are usually released together with the tasks with smaller periods.

B. Case Studies

We consider two case studies to represent both the small-scale systems with only a few periodic tasks to a larger system with 18 periodic tasks.

Case study 1 (avionics system). As the first case study for a small scale embedded system, we used the *electronic control unit* (ECU) of an unmanned aerial vehicle (UAV) system introduced in [36]. The system is composed of 6 periodic tasks. The ECU communicates with the sensor devices such as the GPS as well as the actuators and the camera that are embedded in the system. Table. I reports the parameters of the tasks in the case study. More details can be found in [36].

Case study 2 (fire control system). Here, we consider a real-world application that implements a land-based fire control system that is used for target tracking. The application is implemented with a mix of C and Ada tasks and executes on a PowerPC platform. It is a multi-mode application that consists of 18 periodic tasks, but only a subset of them may be active in any given mode. The application also has 28 background tasks and the system idle task.

Table II shows the task set characteristics. For the WCET we report the largest measured CPU usage over all the periodic jobs of that task. The data was collected with the application running for 5 minutes. Tasks 8 and 18 were unused in the mode of operation that this table was generated from. Task 1 handles all the discrete and analog inputs in bulk. The other tasks at 10ms period are the control loops. The task with 16ms period does graphics processing, and the longer periods are for refreshing an LCD screen, human I/O, and logging activities.

Results. Figs. 7-(a)–(c) and (d)–(f) report the ASR of different attacks for the two case studies. The first observation is that the

TABLE II
PERIODIC TASK SET OF FIRE CONTROL SYSTEM APPLICATION

Task	1	2	3	4	5	6
Period (ms)	10	16	500	10	100	20
WCET* (ms)	0.465	2.794	2.461	2.986	3.627	0.703
ACET (ms)	0.138	0.578	0.553	1.337	1.588	0.226
Task	7	8	9	10	11	12
Period (ms)	500	0	1000	200	100	1000
WCET* (ms)	1.162	0	0.497	0.919	1.414	0.643
ACET (ms)	0.566	0	0.143	0.262	0.261	0.187
Task	13	14	15	16	17	18
Period (ms)	200	10	200	500	10	0
WCET* (ms)	4.317	0.424	1.299	2.901	1.062	0
ACET (ms)	2.914	0.114	0.42	2.064	0.323	0

schedule randomization methods do not eliminate the attacks, in particular, they almost have no effect on POSTERIOR attacks when the victim has a lower priority (see Figs. 7-(b) and (e)). However, they can slightly reduce the ASR in case of ANTERIOR attacks, where the victim task has the lowest priority, i.e., *v:LP*, in comparison to RM. The reason is that randomization may allow the victim (which is a lower priority task) to be executed after its release and before the attacker.

We observed that SS (slot shifting-based randomization) is more vulnerable than TaskShuffler w.r.t. PINCER and POSTERIOR attacks because it significantly increases the interleaving between the tasks and hence increases the chance of an attacker to be executed after the victim. However, on the other hand, it is more successful to reduce ANTERIOR attacks in cases where the victim is the lowest-priority task (Figs. 7-(a) and (d)).

Comparing the two case studies, we observe that when in the first one the number of ANTERIOR attacks is generally smaller than the second case study, in particular for *v:HP, a:randomHP* and *v:HP, a:randomMP*. The reason is that in the first case study, the period of other higher or medium priority tasks is 42, which is not harmonic with 10, hence, the chance that they are scheduled before the highest-priority task due to schedule randomization is lower than the second case study, where there are more harmonic period combinations among high and medium priority tasks.

Average number of preemptions. Fig. 8-(a) compares the average number of calls-to-scheduler for the two case studies. The slot shifting-based schedule randomization has up to three orders of magnitude more calls to the scheduler than the other online policies. For example, in the first case study, RM, TS1, and TS2 have 1.5, TS3 has 1.8, and SS has 4320 call-for-scheduler per job.

As mentioned by Yoon et al. [1], TaskSuffler *does increase* the average number of preemptions per job, however, this increase is not too large for TS1 and TS2 as shown in Fig. 8-(b) for ExpU. TS3, on the other hand, performs fine-grained randomization and hence allows a task to be preempted more often. That is why with the increase in the utilization, the number of preemptions of TS3 increases.

Summary. Our results for the case studies confirm that: **(i)** the slot shifting-based schedule randomization [3] significantly increases the number of preemptions per job to more than 3 orders of magnitude in comparison with RM or TaskShuffler,

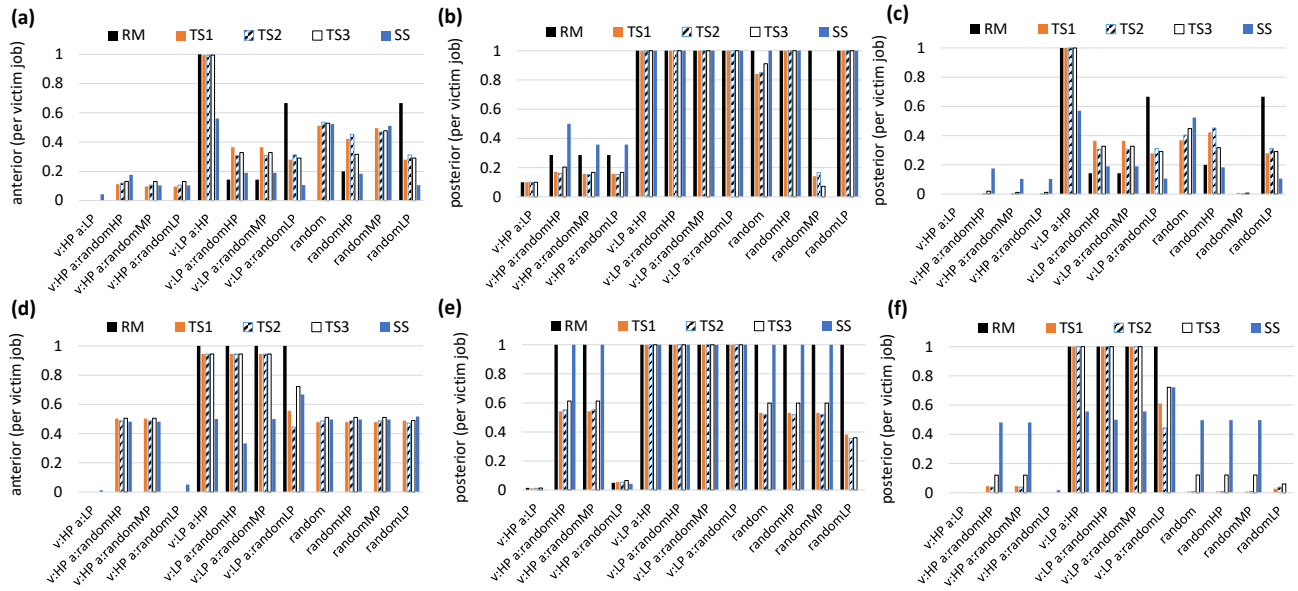

Fig. 7. **(a, b, c):** the ASR of the *case study 1* for various attacks, **(d, e, f):** the ASR of the *case study 2* for various attacks.

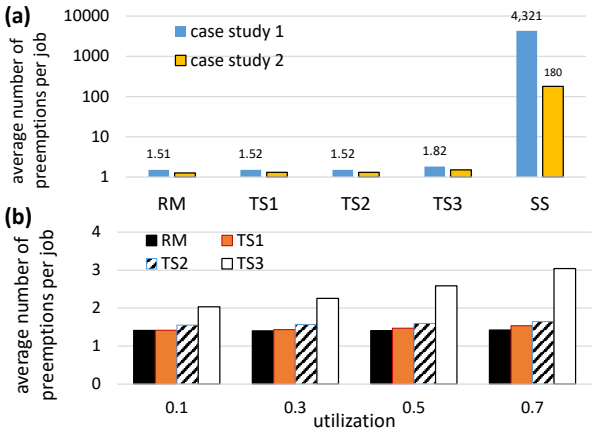

Fig. 8. The average number of calls for scheduler per job in each hyperperiod for **(a):** the two case studies, **(b):** ExpU in Sec. VI-A.

(ii) the schedule randomization algorithms do not eliminate the attacks, in particular, they almost have no effect on POSTERIOR attacks when the victim has a lower priority, **(iii)** SS algorithm increases the number of POSTERIOR and PINCER attacks when the victim is a higher-priority job. This happens because it significantly increases the number of interleaving between tasks and hence increases the ASR. Our results confirms the need to design an attack-aware defense mechanism against schedule-based attacks.

VII. CONCLUSIONS

Summary. In this paper, we focused on the limitations of schedule randomization as a security defense against schedule-based attacks, i.e., attacks whose success depends on a specific ordering between a set of events, such as manipulating data in a shared I/O device buffer *before* the victim task reads the data. We provided definitions and examples of these attacks and discussed the limitations of the schedule randomization-based defenses against these attacks. Our results showed that in some cases, e.g., when the victim is the highest-priority task, randomizing the schedule increases the success rate of a certain class of schedule-based attacks by 20% while those attacks would have been impossible in a system scheduled by a fixed-priority scheduling algorithm. We also observed that certain classes of attacks cannot be avoided at all using either schedule randomization or the fixed-priority scheduling. This raises the need for incorporating a combination of defenses, e.g., isolation-based techniques and schedule-based techniques, to guarantee a system's immunity against schedule-based attacks.

Future work. Our preliminary security tests are mostly focused on harmonic and semi-harmonic tasks since they are conditioned to periods that divide each other. As future work, we will focus on designing a more general (and accurate) security analysis for fixed-priority scheduling that not only identifies immunity to a certain type of attacks but also determines the *worst-case pattern of successful attacks* on a given victim. Such analysis allows us to co-design controllers with the scheduling policy in order to reduce the impact of a potential data-integrity attacks on a control system.

As mentioned in the paper, the existing *schedule entropy* metric, which is designed to evaluate the randomness (uncertainty) of a randomized schedule, is actually optimistic in the presence of an attacker's partial observations, e.g., the attacker can observe its own execution windows. Hence, as future work, we will focus on deriving a sound metric for quantifying schedule uncertainty and evaluating the effectiveness of the schedule randomization methods against attackers that try to infer the future schedule using learning-based techniques. We will also work on a concrete attack case where such a high-precision schedule-inference is essential for the attacker's success.

ACKNOWLEDGEMENTS.

This work was supported in part by NSF under award CSR-1618979, CNS-1646317, and OAC-1839321, and by DHS under award 2017-ST-062-000003. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the U.S. Department of Homeland Security.

REFERENCES

[1] M.-k. Yoon, S. Mohan, C.-Y. Chen, and L. Sha, "Taskshuffler: A schedule randomization protocol for obfuscation against timing inference attacks in real-time systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016, pp. 1–12.

[2] C.-Y. Chen, S. Mohan, R. B. Bobba, R. Pellizzoni, and N. Kiyavash, "ScheduLeak: A Novel Scheduler Side-Channel Attack Against Real-Time Autonomous Control Systems," arXiv:1806.01814v1, Tech. Rep., 2018.

[3] K. Krüger, M. Völp, and G. Fohler, "Vulnerability Analysis and Mitigation of Directed Timing Inference Based Attacks on Time-Triggered Systems ," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2018, pp. 22:1–22:17.

[4] C.-Y. Chen, A. Ghassami, S. Mohan, N. Kiyavash, R. B. Bobba, R. Pellizzoni, and M.-k. Yoon, "A Reconnaissance Attack Mechanism for Fixed-Priority Real-Time Systems," arXiv:1806.01814v1, Tech. Rep., 2017.

[5] C.-Y. Chen, M. Hasan, A. Ghassami, S. Mohan, and N. Kiyavash, "REORDER: Securing Dynamic-Priority Real-Time Systems Using Schedule Obfuscation," arXiv:1806.01393v1, Tech. Rep., 2018.

[6] A. Teixeira, D. Pérez, H. Sandberg, and K. H. Johansson, "Attack models and scenarios for networked control systems," in *International Conference on High Confidence Networked Systems(HiCoNS)*, 2012, pp. 55–64.

[7] A. Teixeira, I. Shames, H. Sandberg, and K. H. Johansson, "Revealing stealthy attacks in control systems," in *Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, 2012, pp. 1806–1813.

[8] G. Park, H. Shim, C. Lee, Y. Eun, and K. H. Johansson, "When adversary encounters uncertain cyber-physical systems: Robust zero-dynamics attack with disclosure resources," in *IEEE Conference on Decision and Control (CDC)*, 2016, pp. 5085–5090.

[9] J. Kim, G. Park, H. Shim, and Y. Eun, "Zero-stealthy attack for sampled-data control systems: The case of faster actuation than sensing," in *IEEE Conference on Decision and Control (CDC)*, 2016, pp. 5956–5961.

[10] H. Jafarnejadsani, H. Lee, N. Hovakimyan, and P. Voulgaris, "Dual-rate l1 adaptive controller for cyber-physical sampled-data systems," in *IEEE Annual Conference on Decision and Control (CDC)*, 2017, pp. 6259–6264.

[11] J. Kim, G. Park, H. Shim, and Y. Eun, "A zero-stealthy attack for sampled-data control systems via input redundancy," arXiv:1801.03609v1, Tech. Rep., 2018.

[12] Y. Mo and B. Sinopoli, "Secure control against replay attacks," in *Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, 2009, pp. 911–918.

[13] R. S. Smith, "A decoupled feedback structure for covertly appropriating networked control systems," *IFAC World Congress*, vol. 44, no. 1, pp. 90–95, 2011.

[14] A. Teixeira, I. Shames, H. Sandberg, and K. H. Johansson, "A secure control framework for resource-limited adversaries," *Automatica*, vol. 51, pp. 135–148, 2015.

[15] R. Zhang and P. Venkitasubramaniam, "Stealthy control signal attacks in linear quadratic gaussian control systems: Detectability reward tradeoff," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 7, pp. 1555–1570, 2017.

[16] F. Pasqualetti, F. Dörfler, and F. Bullo, "Cyber-physical attacks in power networks: Models, fundamental limitations and monitor design," in *IEEE Conference on Decision and Control and European Control Conference (CDC)*, 2011, pp. 2195–2201.

[17] H. Sandberg and A. M. H. Teixeira, "From control system security indices to attack identifiability," in *Science of Security for Cyber-Physical Systems Workshop (SOSCYPS)*, 2016, pp. 1–6.

[18] G. Fohler, "Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems," in *IEEE Real-Time Systems Symposium (RTSS)*, 1995, pp. 152–161.

[19] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No need to hide: Protecting safe regions on commodity hardware," in *European Conference on Computer Systems (EuroSys)*, 2017, pp. 437–452.

[20] "Address space layout randomization (ASLR)," PaX Team, Tech. Rep., 2003. [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt

[21] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Obfuscation: an efficient approach to combat a broad range of memory error exploits," 2003, pp. 105–120.

[22] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software," in *Annual Computer Security Applications Conference (ACSAC)*, 2006, pp. 339–348.

[23] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *USENIX Security Symposium (USENIX Security)*, 2012, pp. 475–490.

[24] Aleph One, "Smashing the stack for fun and profit," Phrack Magazine, Tech. Rep., 1996. [Online]. Available: http://www-inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf

[25] Scut. and PaX Team, "Exploiting format string vulnerabilities," Tech. Rep., 2001. [Online]. Available: http://julianor.tripod.com/teso-fs1-1.pdf.

[26] Anonymous, "Once upon a free()," Phrack Magazine, 11(57), Tech. Rep., 2001. [Online]. Available: http://hamsa.cs.northwestern.edu/media/readings/free.pdf

[27] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *IEEE Symposium on Security and Privacy (SP)*, 2013, pp. 559–573.

[28] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida, "Poking holes in information hiding," in *USENIX Security Symposium (USENIX Security)*, 2016, pp. 121–138.

[29] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space ASLR," in *IEEE Symposium on Security and Privacy (SP)*, 2013, pp. 191–205.

[30] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi, "Missing the point(er): On the effectiveness of code pointer integrity," in *IEEE Symposium on Security and Privacy (SP)*, 2015, pp. 781–796.

[31] Y. Jang, S. Lee, and T. Kim, "Breaking kernel address space layout randomization with intel TSX," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016, pp. 380–392.

[32] D. Trilla, C. Hernandez, J. Abella, and F. J. Cazorla, "Cache side-channel attacks and time-predictability in high-performance critical real-time systems," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 2018, pp. 1–6.

[33] E. Göktaş, R. Gawlik, B. Kollenda, E. Athanasopoulos, G. Portokalidis, C. Giuffrida, and H. Bos, "Undermining information hiding (and what to do about it)," in *USENIX Security Symposium (USENIX Security)*, 2016, pp. 105–119.

[34] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *USENIX Security Symposium (USENIX Security)*, 2014, pp. 401–416.

[35] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *IEEE Symposium on Security and Privacy (SP)*, 2014, pp. 575–589.

[36] R. Pellizzoni, N. Paryab, M. Yoon, S. Bak, S. Mohan, and R. B. Bobba, "A generalized model for preventing information leakage in hard real-time systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015, pp. 271–282.

[37] Y. Z. Lun, A. D'Innocenzo, I. Malavolta, and M. D. Di Benedetto, "Cyber-physical systems security: a systematic mapping study," *arXiv preprint arXiv:1605.09641*, 2016.

[38] J. J. Downs and E. F. Vogel, "A plant-wide industrial process control problem," *Computers & chemical engineering*, vol. 17, no. 3, pp. 245–255, 1993.

[39] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Embedded control systems development with giotto," in *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES)*, 2001, pp. 64–72.

[40] ——, "Giotto: a time-triggered language for embedded programming," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 84–99, 2003.

[41] P. Derler, E. A. Lee, S. Tripakis, and M. Törngren, "Cyber-physical system design contracts," in *ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*. ACM, 2013, pp. 109–118.

[42] A. Ghosal, A. Sangiovanni-Vincentelli, C. M. Kirsch, T. A. Henzinger, and D. Iercan, "A hierarchical coordination language for interacting real-time tasks," in *ACM IEEE International Conference on Embedded Software (EMSOFT)*, 2006, pp. 132–141.

[43] W. Pree and J. Templ, "Modeling with the timing definition language (TDL)," in *Model-Driven Development of Reliable Automotive Services*. Springer Berlin Heidelberg, 2008, pp. 133–144.

[44] M. Sanvido, "A Computer System for Model Helicopter Flight Control; Technical Memo 3: The Software Core," Technical Report 317, Institute for Computer Systems, ETH Zürich, Tech. Rep., 1999.

[45] C. M. Kirsch, M. A. A. Sanvido, T. A. Henzinger, and W. Pree, "A giotto-based helicopter control system," in *Embedded Software*. Springer Berlin Heidelberg, 2002, pp. 46–60.

[46] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *Cryptographers Track at the RSA Conference*, 2006, pp. 1–20.

[47] G. Park, C. Lee, and H. Shim, "On stealthiness of zero-dynamics attacks against uncertain nonlinear systems: A case study with quadruple-tank process," in *International Symposium on Mathematical Theory of Networks and Systems (ISMTNS)*, 2018, pp. 10–17.

[48] C. Delimitrou and C. Kozyrakis, "Bolt: I know what you did last summer... in the cloud," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017, pp. 599–613.

[49] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: Managing performance interference effects for qos-aware clouds," in *European Conference on Computer Systems (EuroSys)*, 2010, pp. 237–250.

[50] A. Cervin, B. Lincoln, K.-E. Arzen, and G. Buttazzo, "The Jitter Margin and Its Application in the Design of Real-Time Control Systems," in *International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA)*, 2004, pp. 1–9.

[51] A. Kerckhoffs, "La cryptographie militaire," *J sci militaires*, vol. IX:5–38, pp. 161–191, 1883.

[52] R. Kumar, E. Kohler, and M. Srivastava, "Harbor: software-based memory protection for sensor nodes," in *Proceedings of the 6th international conference on Information processing in sensor networks*. ACM, 2007, pp. 340–349.

[53] AVR, "Avr32: Architecture document," 2007, reference manual.

[54] ARM, "Cortextm-m3 devices: Generic user guide," 2010, reference manual.

[55] C. H. Kim, T. Kim, H. Choi, Z. Gu, B. Lee, X. Zhang, and D. Xu, "Securing real-time microcontroller systems through customized memory view switching," in *Network and Distributed Systems Security Symp.(NDSS)*, 2018.

[56] A. A. Clements, N. S. Almakhdhub, K. S. Saab, P. Srivastava, J. Koo, S. Bagchi, and M. Payer, "Protecting bare-metal embedded systems with privilege overlays," in *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 2017, pp. 289–303.

[57] A. A. Clements, N. S. Almakhdhub, S. Bagchi, and M. Payer, "{ACES}: Automatic compartments for embedded systems," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 65–82.

[58] "Memory Protection Unit (MPU) Support," 2018. [Online]. Available: www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html

[59] A. Sloss, D. Symes, and C. Wright, *ARM system developer's guide: designing and optimizing system software*. Elsevier, 2004.

[60] Texas Instruments, "Tiva TM4C123GH6PM microcontroller," 2013, datasheet.

[61] R. Kumar, A. Singhania, A. Castner, E. Kohler, and M. Srivastava, "A system for coarse grained memory protection in tiny embedded processors," in *Design Automation Conference, 2007. DAC'07. 44th ACM/IEEE*. IEEE, 2007, pp. 218–223.

[62] N. Bielova, "Short paper: Dynamic leakage: A need for a new quantitative information flow measure," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. ACM, 2016, pp. 83–88.

[63] S. Chattopadhyay, M. Beck, A. Rezine, and A. Zeller, "Quantifying the information leakage in cache attacks via symbolic execution," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 1, p. 7, 2019.

[64] C. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.

[65] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A fast and stealthy cache attack," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2016, pp. 279–299.

[66] S. Mohan, M.-k. Yoon, R. Pellizzoni, and R. Bobba, "Real-time systems security through scheduler constraints," in *2014 26th Euromicro Conference on Real-Time Systems*, July 2014, pp. 129–140.

[67] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings, "Applying new scheduling theory to static priority preemptive scheduling," *Software Engineering Journal*, vol. 8, no. 5, pp. 284–292, 1993.

[68] M. Nasri and B. B. Brandenburg, "An exact and sustainable analysis of non-preemptive scheduling," in *IEEE Real-Time Systems Symposium (RTSS)*, 2017, pp. 1–12.

[69] S. Kramer, D. Ziegenbein, and A. Hamann, "Real world automotive benchmark for free," in *International Workshop on Analysis Tools and Methodologies for Embedded Real-Time Systems (WATERS)*, 2015.

[70] R. Stafford, "Random vectors with fixed sum," University of Oxford, Technical Report, 2006. [Online]. Available: http://www.mathworks.com/matlabcentral/fileexchange/9700