Randomizing Parallel Real-Time Tasks: A Scheduler-Oblivious Mechanism to Harness Security

Xiuqi Zhang
Department of Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, Sweden

Risat Mahmud Pathan
Department of Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, Sweden

Abstract

Periodic parallel task models, such as Directed Acyclic Graph (DAG), offer significant potential for modeling safety-critical real-time applications, but can introduce security vulnerabilities on multicore platforms. The deterministic schedules of periodic real-time tasks can be exploited through schedule-based side-channel attacks – causing missed deadlines or leaking sensitive information. Schedule randomization can address this vulnerability, yet existing work targets sequential tasks and lacks models that incorporate security for parallel workloads while meeting the real-time constraints. Furthermore, no widely accepted metric quantitatively measures the security gained from randomization against different attacker classes.

This paper proposes Controlled DAG Randomization (CDR), a scheduler-oblivious framework that randomizes the schedule of periodic real-time DAG tasks while preserving hard deadline guarantees. Each DAG is transformed into an Augmented DAG (ADAG) via two mechanisms: *dependency augmentation*, which adds edges to constrain concurrency, and *temporal augmentation*, which inserts additional vertices/subtasks that introduce random delays. At runtime, two algorithms randomize the structure of each released instance of the DAG: one adds random precedence edges, and the other assigns randomized execution budgets to the augmented subtasks of the ADAG – both while maintaining schedulability. Two attacker-aware metrics, *System Threat* and *Task Distribution Entropy*, are proposed, targeting intrusive and observation-based attackers, respectively. Extensive simulations on single- and multi-task systems show significant vulnerability reductions. We also observe that increased resources do not always strengthen security and can diminish randomization effectiveness, highlighting the need for carefully designed schedules to realize real security benefits.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Security and privacy → Operating systems security

Keywords and phrases Real-time systems, DAG scheduling, side-channel defense

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2026.9

1 Introduction

The increasing complexity of modern Real-Time Systems (RTS) – from autonomous vehicles to advanced robotics – demands multicore platforms to meet stringent timing and computational requirements. The DAG task model, representing parallel workloads via vertices (subtasks) and edges (dependencies), is a major contender for such systems. Many real-time tasks recur periodically with deterministic execution patterns. While essential for schedulability analysis and certification, this determinism paradoxically exposes timing side-channels: an attacker who has compromised a non-critical task can exploit predictable resource access patterns to disrupt execution or extract sensitive data [10, 2, 27]. Even without direct compromise, observation-based side-channel attacks benefit from such predictability.

© Xiuqi Zhang and Risat Mahmud Pathan;
licensed under Creative Commons License CC-BY 4.0

These risks motivate disrupting deterministic scheduling patterns while preserving schedulability guarantees. Despite decades of research on periodic DAG schedulability [13, 18, 22, 15], the systematic integration of security into parallel real-time scheduling remains largely unexplored. Schedule randomization is a promising defense: varying execution order of the subtasks obscures deterministic patterns. Early approaches like TaskShuffler [36] randomized fixed-priority uniprocessor schedules, but Nasri et al. [25] showed that naive randomization and attacker-oblivious metrics (e.g., generic schedule entropy) can fail to provide security or even create new attack vectors.

For periodic parallel workloads modeled as DAGs, the problem runs deeper than shuffling a ready queue: concurrency creates many execution scenarios that determine which attacker–victim execution overlaps are possible. A usable defense requires three properties: (1) a *constructive mechanism* that produces diverse parallel execution scenarios *without modifying the scheduler* – i.e., a scheduler-oblivious approach – since altering a certified RTOS invalidates certification and incurs re-verification costs; (2) a *hard real-time safety argument* that the injected randomness cannot cause missed deadlines; and (3) *attacker-aware metrics* that quantify security from randomization. To that end, we propose Controlled DAG Randomization (CDR), a framework that enables execution of DAG subtasks in an order that is unpredictable to external observers while guaranteeing schedulability. The main contributions are:

1. **Theoretical Framework for Randomization:** We propose the Augmented DAG (ADAG) model, which transforms a DAG by inserting additional edges and "fake" subtasks. We prove a *sufficiency* result: this model can represent any valid randomized schedule achievable by work-conserving or even non-work-conserving non-preemptive schedulers.
2. **Runtime Randomization Algorithms:** The ADAG model defines a large space of randomized DAGs. We design two runtime algorithms (one for edges, one for fake subtasks) that efficiently sample one such random DAG from this space while preserving schedulability. This enables evaluating the security provided by randomization across the full range of uniformly distributed achievable schedules.
3. **Attacker-Aware Security Metrics:** We propose two security metrics: *System Threat*, quantifying the probability of exploitable attack vectors and measuring defense against *Schedule-Intrusive Attacker*s (i.e., who can change system behavior), and *Task Distribution Entropy*, measuring the uncertainty of each subtask's execution window and measuring defense against *Schedule-Non-Intrusive Attacker*s (i.e., who do not change behavior but only observe the system).
4. **Empirical Evaluation and Insights:** Extensive simulations reveal significant vulnerability reductions across multiple attack classes. We observe a counterintuitive phenomenon: adding cores can sometimes *reduce* security, showing that a DAG's structure and the available slack jointly determine the security achievable through randomization.

The paper proceeds as follows: Section 2 reviews related work; Sections 3–4 present the system and adversary models; Section 5 introduces the security metrics; Section 6 presents the randomization technique and parameterization algorithm; Section 7 extends to multi-task systems; Section 8 reports the evaluation; and Section 9 discusses integration and future work.

2 Related Work

The determinism of RTS, while essential for schedulability analysis, paradoxically introduces security vulnerabilities: adversaries can exploit predictable execution patterns to infer schedules and establish side channels against both fixed-priority [25] and dynamic-priority [9] schedulers [36, 19]. Shared resources such as caches and memory further widen the attack surface via side-channel leakage [32] and false data injection [5, 31].

The vulnerability in deterministic resource allocation is not unique to RTS; it parallels challenges in cloud computing. Han et al. [14] demonstrated that cloud tenants can exploit deterministic virtual machine (VM) placement to co-locate malicious VMs with victims, extracting sensitive information via cross-VM side channels. However, this "co-location problem" is more serious in RTS. In the cloud, the primary defense is *avoidance* – altering placement policies to minimize co-residence probability. In RTS, particularly those modeled as DAGs, absolute avoidance is often infeasible: strict precedence constraints and tight deadlines compel the scheduler to pack tasks closely in both time and across cores. Malicious and victim tasks are therefore frequently forced to execute concurrently or in close sequence. This inherent co-location makes RTS more susceptible to side-channel attacks than cloud environments, necessitating defenses based on randomization rather than mere separation.

Randomization has been explored as a defense against deterministic execution patterns. TaskShuffler [36, 35] introduced randomness into task execution orders for uniprocessor systems, but Nasri et al. [25] showed that naive (i.e., attacker-oblivious) randomization can be ineffective or even introduce new vulnerabilities due to uncontrolled priority inversion, underscoring the need for attacker-aware defenses. Follow-up work such as REORDER [28] extended these ideas to dynamic-priority systems but largely inherited the same limitations in security quantification. A critical challenge in randomized scheduling is balancing security gains with schedulability. Juma et al. [16] formally analyzed this "security-schedulability trade-off" in cloud systems, showing that optimizing for side-channel resistance is often NP-hard. In the cloud, the cost of security is typically measured as migration overhead or reduced throughput. In RTS, the cost is *schedulability* – binary and critical: excessive overhead may cause deadline misses and system failure. Our framework addresses this trade-off for the real-time domain. Instead of expensive runtime migrations, we employ a graph transformation (DAG to ADAG) combined with two low-overhead runtime algorithms that randomly select precedence edges and assign execution times to fake subtasks at each job release, so that different instances exhibit unpredictable execution patterns. By bounding the randomization budget via Graham's bound [13, 22, 18], we guarantee that security overhead never violates hard deadline constraints. Although we use Graham's bound, any other DAG schedulability test may be used instead, e.g., the path-based analysis of He et al. [15].

In dependable computing more broadly, research has moved beyond pure deadline feasibility toward augmented guarantees. For instance, Nair et al. [23] developed fault-tolerant scheduling for multiprocessors that manages redundancy to tolerate processor failures. Our approach aligns with this trend by treating security as a quality-of-service dimension: just as fault tolerance ensures reliability through redundancy, our randomized scheduling ensures confidentiality through unpredictability. While OS-level defenses such as SchedGuard [11] modify the scheduler itself, our approach transforms the task graph, remaining compatible with standard non-preemptive schedulers and complementary to schedule-modification based approaches.

Finally, quantifying security from randomization requires appropriate metrics. Although "schedule entropy" [36] remains widely used [4, 33], one limitation is that it is attacker-oblivious [25]. Other domains also share this need; for instance, Han et al. [14] defined quantitative measures for co-residence probability in cloud computing. Building on this, we introduce two attacker-aware metrics – *System Threat* and *Task Distribution Entropy* – that evaluate the security of randomized schedules against specific adversary classes.

3 System Model

We consider a real-time system comprising a set of n independent periodic parallel tasks $\mathcal{T} = \{G_1, \ldots, G_n\}$ executing on M identical cores. Each task $G_i \in \mathcal{T}$ is modeled as a Directed Acyclic Graph (DAG) $G_i = (V_i, E_i)$, where $V_i = \{v_{i,1}, \ldots, v_{i,n_i}\}$ is the set of $n_i = |V_i|$ vertices (subtasks), each representing a sequential execution unit, and $E_i \subseteq V_i \times V_i$ is the set of directed edges. An edge $(u, v) \in E_i$ encodes precedence: v may execute only after u completes. We denote the sets of predecessors and successors of a subtask u by $\text{pred}(u)$ and $\text{succ}(u)$, respectively. A subtask with no predecessor is the *source*; one with no successor is the *sink*. Without loss of generality, we assume each DAG has exactly one source and one sink.

Each task G_i recurs infinitely with period T_i and relative deadline D_i: all subtasks of an instance released at time r must complete by $r + D_i$. We consider implicit deadlines: $D_i = T_i$. For each $v \in V_i$, let $c(v)$ denote the known Worst Case Execution Time (WCET), which is the maximum time subtask v takes to execute on any core. A path λ in $G_i = (V_i, E_i)$ is a sequence of vertices $(v_{i,0}, \ldots, v_{i,k})$ such that $(v_{i,j}, v_{i,j+1}) \in E_i$ for all $0 \leq j < k$. The *length* of a path is the sum of the WCETs of its vertices. We denote the length of the longest path in G_i as $\text{len}(G_i)$. The *volume* (total work) of G_i is $\text{vol}(G_i) = \sum_{v \in V_i} c(v)$. The utilization of task G_i is $U_i = \text{vol}(G_i)/T_i$, and the total system utilization is $U = \sum_{i=1}^{n} U_i$.

We first present the randomization technique for one DAG task and later discuss the extension to multiple DAG tasks. When clear from context, we omit the subscript i; for example, subtask $v_{i,j}$ of DAG G_i will be written as v_j. We assume a discrete-time model: all execution times, periods, and deadlines are integers.

For the scheduling algorithm, we adopt *federated scheduling* [18]: each DAG task G_i receives m_i dedicated cores (with $\sum m_i \leq M$). The subtasks of each DAG G_i are scheduled on m_i cores under a Non-Preemptive Work-Conserving (NPWC) policy – a subtask becomes ready once all predecessors complete, ready subtasks are dispatched to idle cores immediately, and each dispatched subtask runs to completion without preemption. Many parallel runtime schedulers (e.g., breadth-first, Cilk's work-stealing) are NPWC [24, 12, 29]. Our randomization technique only alters the DAG structure (adding edges and fake subtasks); no scheduler modification is required. Although each G_i's subtasks execute exclusively on its m_i cores, malicious subtasks may still attack victims in other tasks via chip-level shared resources (caches, I/O channels). To this end, we denote $\hat{\mathcal{T}} = \bigcup_{G_i=(V_i,E_i)\in\mathcal{T}} V_i$ as the set of all subtasks across all DAGs in the system.

4 Adversary Model

This section presents the adversary model from two perspectives: what attackers *want* – their goals and capabilities (Section 4.1) – and what they *need* – the attack vectors that enable those goals (Section 4.2).

By Kerckhoffs' principle [17], a system's security should not depend on the secrecy of its design. Accordingly, we assume the attacker has full knowledge of the system, including task set parameters (e.g., periods, deadlines, WCETs) and the scheduling algorithm.

4.1 What an attacker wants: Capabilities and Targets

We distinguish two classes of schedule-based attackers by their goal and mode of interaction: the *Schedule-Intrusive Attacker* and the *Schedule-Non-Intrusive Attacker*.

▶ **Definition 4.1.** *A Schedule-Intrusive Attacker is a type of schedule-based attacker that has compromised at least one subtask of a DAG and then aims to affect or change the behavior of some other (victim) subtasks of the same or other DAGs.*

The *Schedule-Intrusive Attacker* cannot modify scheduling parameters (e.g., period, deadline, priority), but can execute arbitrary code within the compromised subtask's execution window, gaining access to all resources available to that subtask. This enables false-data injection [37] or resource-contention attacks such as cache-based Denial-of-Service [6]. Additionally, we assume that nodes from other ADAGs cannot tamper with the ADAG generation code or its associated data. This can be ensured by executing the generation algorithm within a trusted component or under standard system-level isolation mechanisms (e.g., MMU/MPU-based memory protection). Finally, we assume the runtime enforces per-node execution budgets to mitigate WCET overruns and prevent DoS attacks through temporal isolation.

▶ **Definition 4.2.** *A Schedule-Non-Intrusive Attacker is a type of schedule-based attacker that simply aims to observe the behavior of some victim subtask to gain additional information even without having compromised some subtask.*

The *Schedule-Non-Intrusive Attacker* does not directly interfere with a victim's execution but gathers critical information through side channels. For example, an attacker may passively observe power, electromagnetic, or WiFi fluctuations to infer sensitive data such as keystrokes or the type of computation performed [2], then use the extracted information to compromise other system components (e.g., using a recovered password to access a critical application) or reveal private data. This distinction matters because the two types rely on different attack vectors (as will be discussed in Section 4.2) and require separate metrics and mitigation strategies, both addressable by schedule randomization.

4.2 What an attacker needs: Attack Vectors

An attack vector is the means by which an attacker exploits a vulnerability. Schedule-based vectors fall into two categories: those that depend on the attacker's execution *relative* to a victim subtask's execution window, and those that depend on knowing the exact (absolute) time of the victim's execution. Nasri et al. [25] identified the following four relative-timing vectors:

▶ **Definition 4.3.** *An* Anterior *vector is an attack vector in which an attacker task executes* ***before*** *the victim task.*

▶ **Definition 4.4.** *A* Posterior *vector is an attack vector in which an attacker task executes* ***after*** *the victim task.*

▶ **Definition 4.5.** *A* Pincer *vector is an attack vector in which an attacker task executes both* ***before*** *and* ***after*** *the victim.*

▶ **Definition 4.6.** *A* Concurrent *vector is an attack vector in which an attacker task executes* ***at the same time*** *as the victim task.*

The Pincer vector demands that the attacker execute **both** before and after the victim, making it the hardest to exploit. Nasri et al. [25] adopted this definition possibly to expose weaknesses in existing defenses – showing that even a hard-to-launch attack can succeed. From a *defensive* standpoint, eliminating both the Anterior vector and Posterior vector automatically eliminates Pincer vector, so we adopt a strictly stronger variant that models an attacker who needs *either* (i.e., not necessarily both) relative-timing vectors:

▶ **Definition 4.7.** *A* PincerOR *vector is an attack vector in which an attacker task executes either* ***before or after*** *the victim task.*

These four vectors require the attacker to *co-execute* with the victim – running immediately before, after, or concurrently. They characterize the *Schedule-Intrusive Attacker*, which interferes with or probes the victim through shared resources. Since chip-level shared resources (last-level caches, memory buses) span all cores, we assume an attacker subtask on any core can target a victim on any other core whenever an attack vector's relative timing relationship is satisfied.

A *Schedule-Non-Intrusive Attacker*, by contrast, remains passive and relies on *timing alignment*: it succeeds when the attacker can predict *when* a victim computation occurs, so that external observations (e.g., power, electromagnetic, or WiFi traces) can be sampled at the right moment. We model this as a fifth vector, the Absolute-Time-aware vector.

▶ **Definition 4.8.** *An* Absolute-Time-aware *vector is an attack vector in which an attacker is able to predict* ***when*** *(i.e., the absolute time) the victim task will be executed.*

For example, if a *Schedule-Non-Intrusive Attacker* knows the exact time a decryption routine executes, it can sample power or thermal traces at that moment to infer the key length or the specific algorithm in use.

5 Metrics for Measuring Security

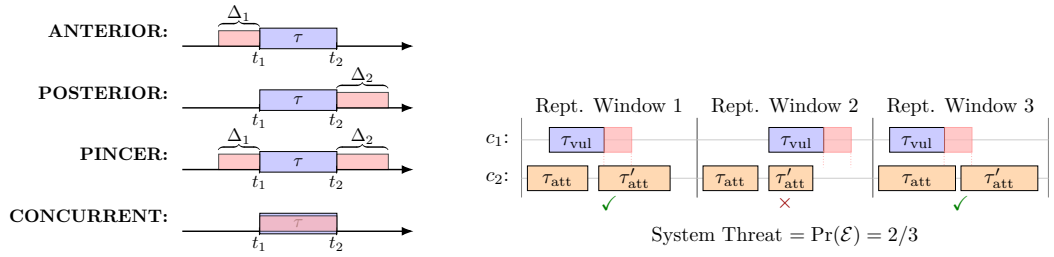
This section introduces two metrics quantifying how effectively CDR defends against schedule-based attacks. *System Threat* (Section 5.1) measures defense against the *Schedule-Intrusive Attacker*, and *Task Distribution Entropy* (Section 5.2) measures defense against the *Schedule-Non-Intrusive Attacker*. We then compare, using an example, our proposed Task Distribution Entropy with the *Schedule Entropy* of TaskShuffler [36] (Section 5.3).

5.1 System Threat

A *Schedule-Intrusive Attacker* requires both a victim and an attacker subtask on the same system (Section 4.2), and that any subtask may act maliciously toward any other – regardless of which core the subtasks execute on (Section 4.1). Recall that $\hat{\mathcal{T}}$ is the set of all subtasks across all DAGs. Let $\hat{\mathcal{T}}_{\text{vul}} \subseteq \hat{\mathcal{T}}$ and $\hat{\mathcal{T}}_{\text{att}} \subseteq \hat{\mathcal{T}}$ denote the sets of vulnerable and attacker subtasks, respectively. Such information may be obtained using threat analysis and risk assessment [34]. We assume an attacker must execute within a bounded interval relative to the victim, called the *vulnerable window*, because attacks far from the victim's execution are unlikely to succeed.

▶ **Definition 5.1** (Vulnerable window)**.** *The vulnerable window of a vulnerable subtask is the time interval during which an attacker subtask's execution can negatively affect the victim's behavior.*

Let $A = \{\text{Anterior}, \text{Posterior}, \text{PincerOR}, \text{Concurrent}\}$ denote the set of attack vectors available to a *Schedule-Intrusive Attacker* (Section 4.2). Consider a subtask $\tau \in V_i$ of DAG task G_i, scheduled under the NPWC algorithm (Section 3), whose j^{th} instance starts executing at time t_1 and finishes at t_2. The vulnerable window of this instance for attack vector $a \in A$ is the interval defined as follows:



(a) Vulnerable windows by attack vector for victim subtask τ.

(b) *System Threat* example: two-core schedule over three repeating windows (✓ = system threat event occurs).

Figure 1 (a) Vulnerable windows by attack vector. (b) Illustrative *System Threat* calculation.

$$\mathcal{V}(a,\tau,j) = \begin{cases} [t_1-\Delta_1, t_1] & \text{if } a = \textsc{Anterior}, \\ [t_2, t_2+\Delta_2] & \text{if } a = \textsc{Posterior}, \\ [t_1-\Delta_1, t_1] \cup [t_2, t_2+\Delta_2] & \text{if } a = \textsc{PincerOR}, \\ [t_1, t_2] & \text{if } a = \textsc{Concurrent}, \end{cases}$$

where Δ_1 and Δ_2 are the durations of the windows immediately before and after subtask τ's execution, respectively, during which an attacker subtask can successfully launch an attack (Fig. 1a).

We assume that if the attacker does not execute within the vulnerable window, then no attack can be launched (Fig. 1a). Let $\mathcal{C}_a(\tau)$ denote the minimum time an attacker must execute within the vulnerable window of victim τ to launch an attack of type $a \in A$.

Because the tasks are periodic, the schedule repeats after a fixed *repeating window*[1] (equal to the least common multiple of all task periods when all are released simultaneously). An attacker can therefore study one repeating window to predict all future schedules when no randomization is performed. Next, we define *threat* and *system threat event*. Our metric *System Threat* is the probability of a system threat event.

▶ **Definition 5.2** (Threat). *A subtask* τ_{att} threatens *a subtask* τ_{vul} *if* τ_{att} *executes within* $\mathcal{V}(a,\tau_{\text{vul}},j)$ *for at least* $\mathcal{C}_a(\tau_{\text{vul}})$ *time units in the* j^{th} *repeating window for some* $j \geq 1$ *and* $a \in A$. *Such a threat event is denoted as* $\text{th}(\tau_{\text{att}}, \tau_{\text{vul}}, a)$.

▶ **Definition 5.3** (System Threat Event). *A* ***system threat event*** $\mathcal{E}$ *occurs when at least one vulnerable subtask is threatened by at least one attacker subtask within an arbitrary repeating window. Formally,* $\mathcal{E}$ *is the union of all individual threat events in a repeating window:*

$$\mathcal{E} = \bigcup_{\tau_{\text{vul}} \in \hat{\mathcal{T}}_{\text{vul}}} \bigcup_{\tau_{\text{att}} \in \hat{\mathcal{T}}_{\text{att}}} \bigcup_{\alpha \in \mathcal{A}(\tau_{\text{vul}})} \text{th}(\tau_{\text{att}}, \tau_{\text{vul}}, \alpha),$$

where $\mathcal{A}(\tau_{\text{vul}}) \subseteq A$ *is the subset of attack vectors to which subtask* τ_{vul} *is vulnerable.*

▶ **Definition 5.4** (System Threat). *The* ***System Threat****, denoted* $\text{TH}(\hat{\mathcal{T}}_{\text{att}}, \hat{\mathcal{T}}_{\text{vul}})$*, is the* ***probability*** *that a system threat event occurs in an arbitrary repeating window:*

$$\text{TH}(\hat{\mathcal{T}}_{\text{att}}, \hat{\mathcal{T}}_{\text{vul}}) = \Pr(\mathcal{E}). \tag{1}$$

[1] With a single task, the repeating window is just the period; with multiple tasks it is the hyperperiod.

Fig. 1b illustrates System Threat on a two-core schedule. Victim subtask τ_{vul} runs on core c_1 and is only vulnerable to a POSTERIOR attack; its vulnerable window (red) follows each execution of τ_{vul}. Two attacker subtasks τ_{att} and τ'_{att} run on core c_2 across three repeating windows with different (i.e., randomized) start times. In Repeating Window 1, τ'_{att} overlaps with the vulnerable window, producing a threat $\text{th}(\tau'_{\text{att}}, \tau_{\text{vul}}, \text{POSTERIOR})$; a system threat event $\mathcal{E}$ therefore occurs (✓). In Repeating Window 2, neither attacker overlaps the window, so no system threat event occurs (×). In Repeating Window 3, both τ_{att} and τ'_{att} overlap the window – two individual threats – but these together still constitute a single system threat event (✓), since $\mathcal{E}$ is defined as the union of all threats within one repeating window. Overall, 2 out of 3 repeating windows contain a system threat event, giving $\text{TH}(\hat{\mathcal{T}}_{\text{att}}, \hat{\mathcal{T}}_{\text{vul}}) = 2/3$.

5.2 Task Distribution Entropy

System Threat captures whether an attacker subtask overlaps a victim's vulnerable window but does not address vulnerability to the ABSOLUTE-TIME-AWARE vector. An ABSOLUTE-TIME-AWARE vector succeeds when an attacker can predict *when* a subtask executes, so the defense must make execution timing hard to predict.

We propose a second metric, *Task Distribution Entropy*, a Shannon-style [30] entropy measuring the *slot uncertainty of a given subtask* – how the start time of the subtask varies across the time slots within the period of the task. This *task-oriented* perspective contrasts with the *Schedule Entropy* of TaskShuffler [36], which is *slot-oriented*: for each time slot, [36] measures how many different subtasks could execute in that time slot. As we will show in Section 5.3, the task-oriented view provides a more accurate evaluation of security.

▶ **Definition 5.5** (Task Distribution Entropy). *For a subtask $\tau_i \in V_k$ belonging to DAG G_k with period T_k, the Task Distribution Entropy is*

$$H_{dist}(\tau_i) = -\sum_{j=1}^{T_k} p_{i,j} \log_2 p_{i,j} \tag{2}$$

where $p_{i,j}$ is the probability that τ_i starts execution in time slot t_j, and $t_j = h \times T_k + j$, $\forall h \in \{0, 1, 2, \ldots\}$ and $1 \leq j \leq T_k$.

Task Distribution Entropy quantifies a subtask's temporal uncertainty. If a subtask tends to start in the same time slot t_j relative to the start of the period, the probability $p_{i,j}$ dominates and $H_{\text{dist}}(\tau_i)$ is low, making the subtask easy to predict. Conversely, higher $H_{\text{dist}}(\tau_i)$ means start times are spread more uniformly, making an ABSOLUTE-TIME-AWARE vector harder.

5.3 Comparison with TaskShuffler

TaskShuffler [36] uses a *slot-oriented* view of uncertainty: for each time slot, it asks *which subtask could execute here?* We instead use a *task-oriented* view: for each subtask, we ask *when could this subtask execute (i.e., how its start time is distributed across the time slots within the period of the task)?* We compare the two approaches with a simple example.

Consider a DAG G with a large number of subtasks, $N = |V| \gg 1$, and a large period T. In each period, the start time of each subtask is uniformly random (hard to predict) except for one subtask τ_i (easy to predict). The subtask τ_i executes with equal chance in the m^{th} or n^{th} time slots in every period (as shown in Fig. 2).

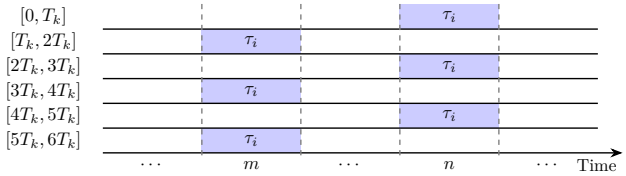


Figure 2 Example illustrating how slot-oriented entropy can overestimate schedule unpredictability. Each horizontal timeline represents one period of length T_k for DAG G_k; subtask $\tau_i \in V_k$ executes in either slot m or slot n in every period, while all other subtasks are fully randomized.

▶ **Definition 5.6** (Schedule Entropy [36]). *The Schedule Entropy of TaskShuffler measures, for each time slot t_j, the uncertainty over which subtask[2] executes in that slot:*

$$H_\Gamma(S_j) = -\sum_{i=1}^{N} p_{i,j} \log_2 p_{i,j} \tag{3}$$

where $p_{i,j}$ is the probability that τ_i executes in time slot t_j, and $t_j = h \times T_i + j$, $\forall h \in \{0, 1, 2, \ldots\}$ and $1 \leq j \leq T_i$.

Note that Eq. (2) computes the entropy of a *subtask* over different time slots, whereas Eq. (3) computes the entropy of a *time slot* over different subtasks. Applying Eq. (3) to the example in Fig. 2, all subtasks other than τ_i have an equal probability of $1/(N-1)$ of appearing in any slot other than m and n. The slot entropy for these slots is $H_\Gamma(S_t) = -(N-1) \times \frac{1}{N-1} \log_2 \frac{1}{N-1} = \log_2 (N-1)$. For slots m and n, subtask τ_i executes with probability $p_{i,m} = p_{i,n} = 0.5$. The remaining 0.5 probability is distributed equally among the other $(N-1)$ subtasks, giving each a probability of $0.5/(N-1)$. The entropy is thus:

$$H_\Gamma(S_m) = H_\Gamma(S_n) = -(N-1) \times \frac{1}{2(N-1)} \log_2 \frac{1}{2(N-1)} - 0.5 \log_2(0.5)$$
$$= 1 - \frac{1}{2} \log_2 \frac{1}{N-1} = 1 + \frac{1}{2} \log_2 (N-1).$$

By contrast, our proposed Eq. (2) yields $H_{\text{dist}}(\tau_i) = -2 \times 0.5 \log_2 0.5 = 1$, which directly reveals τ_i's high predictability – a fact that the slot-oriented schedule entropy obscures. The extension of TaskShuffler [35] takes the minimum slot entropy as its system-level metric;[3] for this example, that minimum is $H_\Gamma(S_m) = 1 + 0.5 \log_2 (N-1)$, which grows logarithmically with N and far exceeds 1 when N is large. By contrast, $H_{\text{dist}}(\tau_i) = 1$ is constant regardless of N, correctly indicating that the execution time slot of τ_i is nearly deterministic.

Task Distribution Entropy thus evaluates security on a subtask-by-subtask basis, leaving it to the system designer to determine how to assess the overall system's security (e.g., via the average or minimum *Task Distribution Entropy* across all subtasks). The example above exposes two limitations of schedule entropy [36]: (i) it is slot-oriented rather than task-oriented, so high entropy in some slots can mask the vulnerability of specific subtasks; and (ii) aggregating into an "overall system entropy" overstates security, because high entropy in one part of the schedule provides little protection for a more predictable part. In practice, an attacker is more likely to target individual subtasks rather than the entire system at once, so an aggregated entropy metric tends to overestimate security.

[2] TaskShuffler [36] does not consider parallel tasks; we use "subtask" here for consistent terminology.
[3] The original TaskShuffler [36] sums the entropy for all slots as the system security evaluation, which suffers even more from the same limitation.

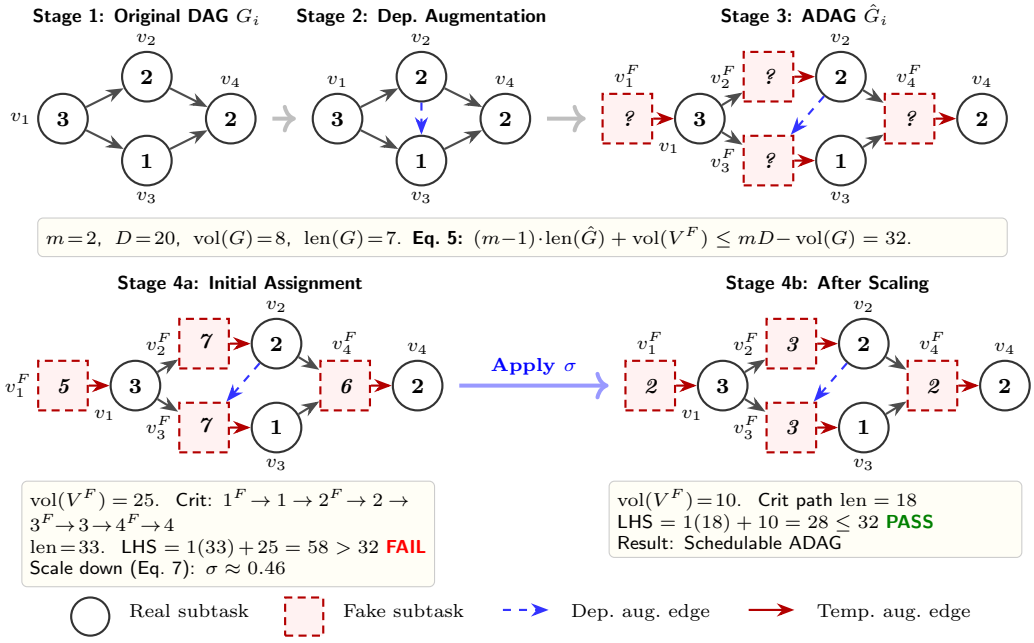


Figure 3 End-to-end illustration of the CDR framework for one job release. **Stage 1**: the original DAG G_i with four subtasks (WCET $c(v)$ shown inside each subtask). **Stage 2**: Dependency Augmentation randomly adds a precedence edge $e \in E_\Delta$ (dashed blue) between formerly concurrent subtasks. **Stage 3**: Temporal Augmentation inserts a fake subtask v^F before each real subtask, forming the ADAG $\hat{G}_i$. **Stage 4a**: the Fake Subtask Refresh (FRESH) algorithm assigns random WCETs to fake subtasks; the schedulability check (Eq. (5)) fails, triggering a scale-down of each fake subtask's execution time by factor σ. **Stage 4b**: after scaling, the ADAG satisfies the schedulability bound and is ready for execution.

6 Randomization Approach

This section presents our proposed randomization framework, Controlled DAG Randomization (CDR), which transforms a standard DAG task into an ADAG (Augmented DAG). We first define the ADAG *model* – a parameterized construction that, given a DAG G_i and a set of additional precedence edges E_Δ, produces an augmented graph $\hat{G}_i$. Two mechanisms drive the augmentation: *Dependency Augmentation* (adding E_Δ to restrict execution orders) and *Temporal Augmentation* (inserting "fake" subtasks that introduce controlled delays). We prove this model *sufficient*: for any valid non-preemptive schedule $S^\star$, there exist a choice of E_Δ and fake-subtask WCETs that generate an ADAG whose execution using the NPWC scheduler reproduces $S^\star$ (Theorem 6.3). We then present two runtime algorithms that, at each job release, efficiently select a random E_Δ (Section 6.4) and assign random execution budgets to the fake subtasks (Section 6.5) while guaranteeing deadlines. This section focuses on single-DAG randomization; the extension to multi-task systems appears in Section 7. Fig. 3 illustrates the complete framework.

6.1 DAG Augmentation Framework

CDR converts an original DAG G_i into an ADAG $\hat{G}_i$ through two phases.

Phase 1: Dependency Augmentation. This phase adds ordering constraints to the DAG. The original $G_i = (V_i, E_i)$ defines the minimal precedence for correctness but typically permits multiple topological orderings, implying multiple execution scenarios. To prevent attackers from exploiting specific concurrency patterns (e.g., a CONCURRENT vector), we enforce sequential ordering between independent subtasks (illustrated by the dashed edge E_Δ in Stage 2 of Fig. 3).

We add new precedence edges between subtask pairs that have no existing dependency. Since multiple additions may jointly induce a cycle even if each is individually valid, we require the following property when randomly picking E_Δ.

▶ **Definition 6.1** (Valid Dependency Extension). *Given a DAG $G_i = (V_i, E_i)$, a set of directed edges E_Δ over V_i disjoint from the original set of edges E_i is a **Valid Dependency Extension** if the resulting graph $G'_i = (V_i, E_i \cup E_\Delta)$ is acyclic.*

We select a valid dependency extension E_Δ to generate an intermediate graph G'_i; we perform this selection online as will be presented in Section 6.4.

Phase 2: Temporal Augmentation. Given the intermediate graph $G'_i = (V_i, E'_i)$ where $E'_i = E_i \cup E_\Delta$, this phase injects "fake" workloads to introduce controllable delays. We construct the ADAG $\hat{G}_i$ by inserting a fake subtask v^F for each original node $v \in V_i$. Every incoming edge $(u, v) \in E'_i$ to v is replaced by two edges (u, v^F) and (v^F, v), as shown in Stage 3 of Fig. 3. The execution time of v^F is unspecified at this point (denoted as '?' in Fig. 3) and will be assigned at runtime by the budget allocation algorithm.

Dependency augmentation narrows feasible topological orders, fixing *which parallel execution orders are allowed*. Temporal augmentation inserts fake subtasks so that every precedence edge in the intermediate DAG becomes a controllable delay chain $u \to v^F \to v$, controlling *when each eligible subtask becomes runnable at the earliest*. At this stage, the construction treats E_Δ as a given input; how E_Δ is selected is a runtime concern addressed in Section 6.4. As we prove in Theorem 6.3, this construction of the ADAG does not limit the space of achievable randomizations.

Algorithm 1 Construct Augmented DAG (ADAG).

```
1:  procedure CONSTRUCT ADAG(G_i = (V_i, E_i), E_Δ)
2:      Input: Original DAG G_i, Set of extension edges E_Δ
3:      Output: ADAG Ĝ_i
4:      V̂_i ← V_i, Ê_i ← E_i ∪ E_Δ
5:      for all v ∈ V_i do
6:          v^F ← new fake subtask
7:          V̂_i ← V̂_i ∪ {v^F}, Ê_i ← Ê_i ∪ {(v^F, v)}
8:          P ← {u | (u, v) ∈ E_i ∪ E_Δ}
9:          for all u ∈ P do
10:             Ê_i ← (Ê_i \ {(u, v)}) ∪ {(u, v^F)}
11:     return Ĝ_i = (V̂_i, Ê_i)
```

Algo. 1 formalizes the ADAG construction for a given E_Δ. Algo. 1 initializes $\hat{E}_i$ with the original and extension edges (line 4). For each original subtask v, a new fake subtask v^F is created and added as a predecessor of v (lines 6–7). Line 8 extracts all predecessors u of v that are in $(E \cup E_\Delta)$; for each such u, the original edge (u, v) is replaced by (u, v^F) (line 10), so that every path to v now passes through v^F.

Let V_i^F denote the set of fake subtasks in the final ADAG $\hat{G}_i = (\hat{V}_i, \hat{E}_i)$, where $\hat{V}_i = V_i \cup V_i^F$. The total volume of $\hat{G}_i$ is $\mathrm{vol}(\hat{G}_i) = \mathrm{vol}(G_i) + \mathrm{vol}(V_i^F)$. It is not difficult to see that the construction of the ADAG preserves the original precedence constraints. In practice, the

ADAG construction can be implemented as a compile-time source-to-source transformation, as discussed in Section 9. The fake subtasks can execute any piece of code (e.g., a cache-clearing routine, an empty loop) and can even accommodate the overhead of online ADAG generation while respecting their allocated execution budgets.

By construction, the transformation replaces every original edge (u, v) with a two-edge path $u \to v^F \to v$, preserving all original reachability relations and keeping $\hat{G}_i$ acyclic (since E_Δ is a Valid Dependency Extension and splitting edges into longer paths introduces no cycles).

6.2 Sufficiency of the ADAG Model

Having established the mechanisms for dependency and temporal augmentation, we now address the framework's expressive power: *Can the ADAG model represent any desired execution behavior?* An *execution timeline* (or simply *timeline*) is a concrete assignment of start times to subtasks consistent with non-preemptive scheduling on m cores, whether work-conserving or not. This section proves a sufficiency result: for *any* valid timeline, there exists a choice of added edges E_Δ and fake subtasks' WCETs that compels the NPWC scheduler to exactly reproduce the timeline. Let S be a schedule for a DAG $G = (V, E)$ on m identical cores, mapping every subtask to a start time.

▶ **Definition 6.2** (Valid Timeline). *A schedule $S : V \to \mathbb{N}$, where $S(v)$ denotes the start time of subtask v, is a **Valid Timeline** for G on m cores if:*

1. ***Precedence:*** *$\forall (u, v) \in E,\ S(v) \geq S(u) + c(u)$.*
2. ***Capacity:*** *At most m subtasks execute at any time.*
3. ***Non-Preemption:*** *Each subtask executes till completion without being preempted.*

This definition captures all possible non-preemptive schedules, whether work-conserving or not: the only restrictions are resource capacity (m cores) and precedence – a core may idle even if ready subtasks exist. We now state a theorem confirming that the ADAG model can express every non-preemptive schedule. Its implications are discussed after the proof.

▶ **Theorem 6.3** (Model Sufficiency). *Let $S^\star$ be a Valid Timeline for a DAG G on m cores. There exist a Valid Dependency Extension E_Δ and a set of fake subtasks' WCETs such that the NPWC scheduler executing the resulting ADAG $\hat{G}$ produces a schedule where every original subtask $v \in V$ starts at $S^\star(v)$ (i.e., the same $S^\star$ schedule is generated).*

Proof. We will first construct $\hat{G}$ and then show how the NPWC scheduler can generate $S^\star$.

Phase 1 (Dependency Augmentation). For every pair $u, v \in V$ with $S^\star(u) + c(u) \leq S^\star(v)$, add the edge (u, v) to E_Δ. That is, every subtask that finishes at or before v starts its execution becomes a new predecessor of v. A cycle in $G' = (V, E \cup E_\Delta)$ would yield $S^\star(v) > S^\star(v)$ for some v on the cycle – a contradiction, so G' is acyclic.

Phase 2 (Temporal Augmentation). For each $v \in V$, insert a fake subtask v^F: every edge $(u, v) \in E'$ is replaced by two edges (u, v^F) and (v^F, v). Let $R(v) = \max_{u \in \mathrm{pred}_{G'}(v)}(S^\star(u) + c(u))$ (or 0 if v is the source) and we set the execution budget of the fake subtask v^F as $c(v^F) = S^\star(v) - R(v) \geq 0$.

After Phase 1, every subtask that finishes before v starts is a predecessor of v, so all such subtasks complete no later than $R(v)$. The only tasks that are still running at $R(v)$ in NPWC are those that finish strictly after $S^\star(v)$. Since v itself occupies one of the m cores at $S^\star(v)$ in $S^\star$, at most $(m-1)$ other tasks (whether started before or exactly at $S^\star(v)$) occupy cores at that instant due to non-preemptive behavior. Hence at least one core is idle during

$[R(v), S^\star(v)]$; the fake subtask v^F fills exactly this gap by following the work-conserving behavior, so v only becomes ready at $S^\star(v)$. Since at most $(m-1)$ subtasks are in execution at $S^\star(v)$, at least one core is free for v to start executing at time $S^\star(v)$. Since each subtask v in NPWC starts at $S^\star(v)$, the NPWC scheduler generates the same schedule as $S^\star$. ◀

Why the Sufficiency Result Matters. Theorem 6.3 shows that the ADAG construction reproduces *any* execution behavior possible under non-preemptive scheduling, whether work-conserving or not. This means randomization can be realized entirely by transforming the task graph (choosing E_Δ and WCETs for the fake subtasks). The underlying scheduler remains untouched, avoiding the intrusive and costly process of modifying a certified OS kernel. Moreover, the proof is constructive: it shows *how* to translate a desired valid timeline into concrete graph constraints (added edges and fake-subtask delays) that make that timeline emerge under NPWC scheduling.

The preceding subsections defined the ADAG model and established that it can reproduce any valid non-preemptive schedule. We now address the complementary runtime question: given a DAG task, how should E_Δ and the execution times of the fake subtasks be chosen at each job release without missing the deadline?

6.3 Schedulability Analysis

At runtime, fake subtasks receive random execution times based on the spare processing capacity (i.e., available slack) before the deadline. The random execution time for all the fake subtasks is determined based on schedulability analysis of the DAG. The question is: for a given number of dedicated cores m, how large can the fake-work volume $\mathrm{vol}(V^F)$ be while keeping $\hat{G}_i$ schedulable under NPWC scheduler? We use Graham's classic bound [13] to determine the random execution time of the subtasks.

▶ **Theorem 6.4** (Graham's Bound [13, 22]). *A periodic DAG G scheduled with a NPWC policy on m dedicated cores meets its deadline D if:*

$$\mathrm{len}(G) + \frac{\mathrm{vol}(G) - \mathrm{len}(G)}{m} \leq D. \tag{4}$$

Transforming G_i into $\hat{G}_i$ increases volume and potentially the critical path. We adapt Graham's bound to the ADAG model as follows:

▶ **Lemma 6.5.** *The ADAG $\hat{G}_i$ derived from DAG G_i is schedulable under NPWC scheduling if:*

$$(m-1) \cdot \mathrm{len}(\hat{G}_i) + \mathrm{vol}(V^F) \leq m \cdot D_i - \mathrm{vol}(G_i). \tag{5}$$

Proof. Applying Graham's bound (Eq. (4)) to $\hat{G}_i$ and substituting $\mathrm{vol}(\hat{G}_i) = \mathrm{vol}(G_i) + \mathrm{vol}(V^F)$ yields the result after rearranging. ◀

Eq. (5) makes the *security–schedulability trade-off* explicit. The right-hand side, $m \cdot D_i - \mathrm{vol}(G_i)$, is the *total slack* offered by m cores over a deadline window of length D_i after accounting for the original work $\mathrm{vol}(G_i)$. This slack is the *randomization budget*: it can be "spent" on (i) additional fake-work volume $\mathrm{vol}(V^F)$ (Temporal Augmentation) and (ii) any increase in the augmented critical path $\mathrm{len}(\hat{G}_i)$. The critical-path cost is amplified by the factor $(m-1)$ in Eq. (5). When $\mathrm{vol}(G_i)$ is close to $m \cdot D_i$, scheduling flexibility vanishes, leaving little room for schedule-based randomization. Stages 4a and 4b of Fig. 3 illustrate this interplay: a randomly sampled set of fake WCETs may violate the bound, requiring a scale-down of the WCETs of the fake subtasks before the ADAG becomes schedulable.

Since both augmentation mechanisms consume slack – Dependency Augmentation can increase $\text{len}(\hat{G}_i)$ and Temporal Augmentation increases $\text{vol}(V^F)$ – each must respect Eq. (5). The following two runtime algorithms jointly construct a random ADAG from the space of all possible ADAGs at each job release:

1. **Runtime Dependency Augmentation:** randomly selects E_Δ to reduce exploitable concurrency, restricting choices to those that (i) preserve acyclicity and (ii) keep the intermediate DAG feasible under Graham's bound (Section 6.4).
2. **Runtime Fake Subtask Budget Assignment (FRESH):** assigns random WCETs to fake subtasks and iteratively scales them down until Eq. (5) is met (Section 6.5), since fake WCETs can change the critical path and thus $\text{len}(\hat{G}_i)$.

6.4 Runtime Dependency Augmentation

Dependency augmentation randomly adds precedence edges to constrain concurrency, reducing exploitable overlaps (in particular, CONCURRENT vectors). Adding edges does not change the randomization budget $m \cdot D - \text{vol}(G)$ (the right-hand side of Eq. (5)), but can increase the critical-path length $\text{len}(G')$ of the intermediate graph, consuming a larger share of that budget on the left-hand side. Accordingly, our augmentation mechanism admits an edge only if it (i) does not create a cycle and (ii) preserves feasibility under Graham's bound.

Algo. 2 generates a randomized valid dependency extension by sampling candidate edges with probability p. An edge (u, v) is *safe* if adding it to the current graph keeps the graph acyclic and satisfies Graham's bound, i.e., (i) it does not introduce a directed cycle (equivalently, there is no existing path $v \rightsquigarrow u$ before adding (u, v)), and (ii) the intermediate graph still satisfies Eq. (5) for the given (m, D).

Algorithm 2 Schedulability-Constrained Edge Augmentation.

```
1:  procedure AugmentDAG(G = (V, E), m, D, p)
2:      Input: DAG G = (V, E), cores m, deadline D, sampling probability p ∈ [0, 1].
3:      Output: Augmented edge set E'.
4:      E' ← E, C ← {(u, v) | u, v ∈ V, u ≠ v, (u, v) ∉ E}
5:      for all (u, v) ∈ C do
6:          if Random(0, 1) ≤ p then
7:              if not PathExists(v, u, (V, E')) then
8:                  E_tmp ← E' ∪ {(u, v)}
9:                  if GrahamBoundSatisfied((V, E_tmp), m, D) then
10:                     E' ← E_tmp
11:     return E'
```

The cycle check $\text{PathExists}(v, u, (V, E'))$ is a standard reachability query. We maintain the transitive closure of the edge set incrementally, reducing each check to a constant-time set membership test; maintaining this closure across $O(|V|^2)$ candidate edges yields $O(|V|^3)$ worst-case complexity.

An edge (u, v) is added only if no path from v to u exists in the current graph (line 7), preventing cycles, and only if the intermediate graph still satisfies Graham's bound (line 9), maintaining schedulability after every addition. The parameter p controls the density of augmentation: a higher p samples more candidate edges, producing a more constrained graph with less exploitable concurrency. The resulting intermediate DAG G' (e.g., the dashed edge E_Δ in Stage 2 of Fig. 3) can therefore be safely passed to Phase 2, where runtime fake-subtask budgets are sampled within the remaining schedulability margin.

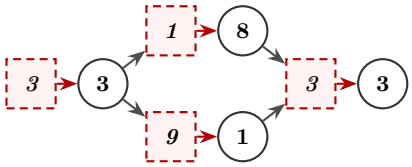


Figure 4 Effect of fake-subtask insertion on the critical path of an ADAG: adding fake subtasks (squares) can shift the longest path from one branch to another (here from top to bottom).

6.5 Runtime Fake Subtask Refresh (FRESH)

We now describe how to assign random WCETs to fake subtasks at runtime while preserving schedulability.

Directly using Eq. (5) to randomly assign fake-subtask execution time $\mathrm{vol}(V^F)$ is problematic because $\mathrm{len}(\hat{G}_i)$ depends on the assigned fake WCETs: different assignments can change the critical path from one branch to another. Fig. 4 illustrates this: the top path has length $(3+8+3)=14$ in the original DAG (ignoring the square-shaped fake subtasks), but in the ADAG the bottom path has length $(3+3+9+1+3+3)=22$ and becomes the longest. We address this circular dependency via the following lemma.

▶ **Lemma 6.6.** *If the condition in Eq.* (5) *is met, then we have*

$$\mathrm{vol}(V^F) \le m \cdot D_i - \mathrm{vol}(G_i) - (m-1)\cdot \mathrm{len}(G_i). \tag{6}$$

Proof. Follows from Eq. (5) and that $\mathrm{len}(\hat{G}_i) \ge \mathrm{len}(G_i)$. ◀

We use the right-hand side of Eq. (6) as the initial budget to randomly assign WCETs to the fake subtasks. Since this condition is necessary but not sufficient, we verify the assigned WCETs against Eq. (5). If the check fails, we iteratively scale down all fake WCETs by a factor $\sigma \in (0,1)$ until Eq. (5) holds.

To derive a suitable σ, let len_o denote the sum of WCETs of the *original* subtasks on the current longest path of $\hat{G}_i$, so that $\mathrm{len}(\hat{G}_i) = \mathrm{len}_o + (\mathrm{len}(\hat{G}_i) - \mathrm{len}_o)$. After scaling, this path's length becomes $\mathrm{len}_o + \sigma(\mathrm{len}(\hat{G}_i) - \mathrm{len}_o)$, and the total fake volume becomes $\sigma \cdot \mathrm{vol}(V^F)$. Substituting into Eq. (5) gives:

$$(m-1)\big(\mathrm{len}_o + \sigma(\mathrm{len}(\hat{G}_i) - \mathrm{len}_o)\big) + \sigma \cdot \mathrm{vol}(V^F) = m \cdot D_i - \mathrm{vol}(G_i)$$

Solving for σ yields:

$$\sigma = \frac{m \cdot D_i - \mathrm{vol}(G_i) - (m-1)\,\mathrm{len}_o}{\mathrm{vol}(V^F) + (m-1)(\mathrm{len}(\hat{G}_i) - \mathrm{len}_o)} \tag{7}$$

which is the estimated scale-down factor for all fake WCETs. However, applying σ may still not satisfy Eq. (5) because a different path now may become critical after scaling. For instance, in Fig. 4, scaling by $\sigma = 0.5$ gives the top path a total WCET of $0.5(3+1+3)+3+8+3 = 17.5$ and the bottom path $0.5(3+9+1)+3+1+3 = 13.5$. The top path becomes the longest path, and Eq. (5) is still not met. We therefore iterate – recomputing σ on the new critical path and scaling again – until the bound is satisfied. Algo. 3 formalizes this process; Fig. 3 (Stages 4a–4b) walks through a concrete example.

Algo. 3 constructs the ADAG (line 4), computes the initial budget via Eq. (6) (line 5), randomly distributes this budget among the fake subtasks, storing the result in the array C^F of fake WCETs (line 6), and iteratively scales them using σ from Eq. (7) until Eq. (5) is satisfied (lines 7–10). The loop terminates because $\sigma < 1$ in every iteration (since

Algorithm 3 Fake Subtask Refresh (FRESH).

procedure FRESH(G_i, E_Δ, D_i, m)
  Input: Original DAG $G_i = (V_i, E_i)$, extension edges E_Δ, deadline D_i, cores m.
  Output: Map $\{v \mapsto c(v) \mid v \in V^F\}$.
  $\hat{G}_i \leftarrow$ CONSTRUCT ADAG(G_i, E_Δ) ▷ Algo. 1
  $B_0 \leftarrow m \cdot D_i - \mathrm{vol}(G_i) - (m-1) \cdot \mathrm{len}(G_i)$
  $C^F \leftarrow$ DISTRIBUTERANDOMLY(B_0, V^F)
  while Eq. (5) not met **do**
    $\sigma \leftarrow$ solve Eq. (7)
    for all $v \in V^F$ **do**
      $C^F[v] \leftarrow \lfloor C^F[v] \cdot \sigma \rfloor$

Eq. (5) was not satisfied) and the floor operation ensures $\mathrm{vol}(V^F)$ strictly decreases. Since $\mathrm{vol}(V^F) \leq m \cdot D_i$ (Eq. (6)), the iteration count is bounded by $m \cdot D_i$ (pseudo-polynomial); in practice, convergence averages fewer than 3 iterations across all experiments in Section 8. The state-of-the-art single-DAG schedulability test [15] can replace Graham's bound; we use Eq. (4) for simplicity of presentation.

At each job release, both augmentation phases run again and generate another random ADAG. Dependency Augmentation (Algo. 2) randomly selects precedence edges E_Δ to restrict concurrency while preserving acyclicity and Graham feasibility. FRESH (Algo. 3) then constructs the ADAG – inserting one fake subtask v^F per original subtask v (Algo. 1) – and samples random fake-subtask budgets, scaling them if necessary to satisfy Eq. (5).

7 Expansion to Multi-Task System

The preceding sections developed CDR for a single DAG on dedicated cores. We now extend it to multi-task systems under federated scheduling [18]. The key idea is to partition tasks by utilization: each high-utilization task receives its own dedicated cores, while all low-utilization tasks share a common cluster of the remaining cores.

Consider the taskset $\mathcal{T}$ of n periodic DAG tasks to be scheduled on M cores (as defined in Section 3). Federated scheduling partitions $\mathcal{T}$ into *High-utilization tasks:* $\mathcal{T}_{\text{high}} = \{G_i \in \mathcal{T} \mid U_i > 1\}$ and *Low-utilization tasks:* $\mathcal{T}_{\text{low}} = \{G_i \in \mathcal{T} \mid U_i \leq 1\}$. Each high-utilization task $G_i \in \mathcal{T}_{\text{high}}$ is assigned $m_i = \lceil (\mathrm{vol}(G_i) - \mathrm{len}(G_i))/(D_i - \mathrm{len}(G_i)) \rceil$ dedicated cores [18] – the smallest core count that satisfies Graham's bound (Theorem 6.4) for G_i. The taskset is then schedulable if the cores remaining after these assignments suffice for the low-utilization tasks [18]:

$$M - \sum_{G_i \in \mathcal{T}_{\text{high}}} m_i \geq 2 \sum_{G_i \in \mathcal{T}_{\text{low}}} U_i. \tag{8}$$

We apply CDR to each category as follows.

For *High-utilization tasks*, each $G_i \in \mathcal{T}_{\text{high}}$ runs in isolation on its m_i dedicated cores, so the single-task CDR framework applies directly. We run FRESH (Algo. 3) on each such task independently, using the slack on its dedicated cores (see Section 6.3) to inject randomization.

For *Low-utilization tasks*, after assigning cores to high-utilization tasks, the remaining $M_{\text{remain}} = M - \sum_{G_i \in \mathcal{T}_{\text{high}}} m_i$ cores form a shared cluster for all tasks in $\mathcal{T}_{\text{low}}$. When Eq. (8) holds with strict inequality, this cluster has a capacity margin $\Delta = M_{\text{remain}} - 2\sum_{G_i \in \mathcal{T}_{\text{low}}} U_i > 0$. We exploit this margin for randomization: Δ is distributed randomly among the tasks in $\mathcal{T}_{\text{low}}$, and each task's share is then distributed among its fake subtasks. Formally, for each $G_i \in \mathcal{T}_{\text{low}}$, we increase its utilization by $U_i^F \geq 0$ subject to two constraints:

1. $U_i + U_i^F \leq 1$, so that G_i remains a low-utilization task; and
2. $\sum_{G_i \in \mathcal{T}_{\text{low}}} (U_i + U_i^F) \leq M_{\text{remain}}/2$, so that Eq. (8) remains satisfied.

The low-utilization tasks then can be scheduled on the shared cores using partitioned EDF [20] or partitioned RM [1]. As established in the threat model, cross-task attacks are possible: subtasks of one DAG may attack those of another.

8 Experiment and Evaluation

We evaluate CDR through extensive experiments. Two metrics from Section 5 guide the evaluation: *System Threat* (probability of victim–attacker temporal overlap; Eq. (1)) and *Task Distribution Entropy* (Shannon entropy of a subtask's start-time distribution; Eq. (2)). We first study how attack-model parameters affect *System Threat*. We then assess multi-task security under federated scheduling using the full CDR pipeline (dependency augmentation followed by temporal augmentation). We isolate single-task scenarios to characterize how augmentation affects an individual DAG without inter-task interactions. Finally, we report runtime overhead measurements of CDR (Section 8.5). Exploring the parameter space required approximately 200 000 core-hours on an AMD EPYC 9354 Zen4 cluster using an OpenMP/MPI implementation, driven by sweeping attack-model, augmentation, and core-count parameters across hundreds of task sets with 10 000 Monte Carlo runs each. Random numbers were generated via the Mersenne Twister [21].

8.1 Experimental Setup

We randomly generated all DAGs using the method of Melani et al. [22]. Generation starts from a root subtask and repeatedly expands each non-terminal subtask by sampling a set of successors. Each successor is designated terminal or non-terminal with a preset probability, and non-terminal successors are expanded recursively up to a maximum depth of 3.

We consider four *Schedule-Intrusive Attacker* vectors: Anterior, Posterior, Pincer, and Concurrent. Unless otherwise specified, the vulnerable-window length is set to 20% of the victim subtask's WCET. The minimum attacker execution time ($\mathcal{C}_a(\tau_{\text{vul}})$) is set to 10% of the corresponding vulnerable-window length. Section 8.2 reports how varying these parameters affects *System Threat*.

We evaluate configurations using *System Threat* (Eq. (1)) and *Task Distribution Entropy* (Eq. (2)), estimated via Monte Carlo simulation. Each run produces one randomized schedule for a full hyperperiod. *System Threat* is estimated as the fraction of total runs that contain at least one system threat event (Def. 5.3). We use 10,000 runs per DAG for *System Threat* and 5,000 runs for *Task Distribution Entropy*, yielding a relative standard error below 0.3%.

8.2 Sensitivity to Attack Parameters

We first investigate how attack parameters affect *System Threat* using 100 DAG tasks ($U_i \in [0.1, 5.0]$). We swept two parameters in 10% increments: (1) **Vulnerable window size**: from 10% to 100% of the victim subtask's WCET, and (2) **Required attacker time**: from 10% to 100% of the vulnerable window.

As Fig. 5 shows, *System Threat* increases with larger vulnerable windows and decreases with longer required attacker execution times. Based on these trends, we fix the vulnerable window at 20% and the required attacker time at 10% for all subsequent experiments.

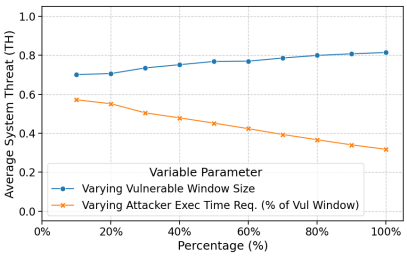


Figure 5 *System Threat* sensitivity to vulnerable window size (% of victim WCET) and required attacker execution time (% of vulnerable window).

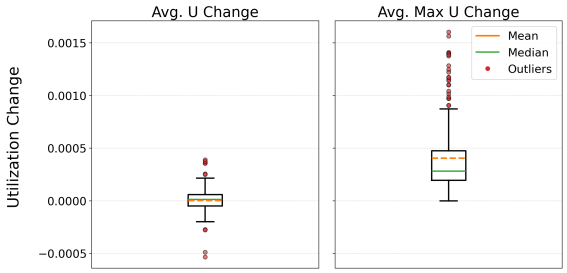


Figure 6 Utilization change from Highly Composite Numbers (HCN)-based hyperperiod scaling. Mean (orange dashed), median (green solid), and interquartile range are shown.

8.3 Evaluation of Multi-Task Security

We evaluate multi-task systems under federated scheduling. We generated 200 task sets, each containing a random number of DAGs, with total utilization up to 15.0.

The hyperperiod (the least common multiple of all task periods) can grow very large for random period sets – in ours, the largest exceeds 10^{40} – making full-hyperperiod simulation infeasible. We therefore scale each task set to a bounded hyperperiod $H^\star$ chosen from a predefined set of HCN [26]: we select the HCN closest to the original hyperperiod, then for each task τ_i replace its period with the divisor of $H^\star$ closest to T_i. Subtask WCETs are scaled by $\alpha_i = T_i'/T_i$, where T_i' is the new period assigned to task τ_i and the fractional remainders are then redistributed greedily (largest remainder first) to minimize utilization error. Fig. 6 confirms that this scaling introduces negligible utilization distortion even when compressing hyperperiods from 10^{40} down to at most 45,360.

We apply both CDR phases: *Schedulability-Constrained Edge Augmentation* (Dep. Aug.; Algo. 2) followed by FRESH. Core availability is expressed as a *core multiplier* $\kappa = M/M_{\min}$, where M is the available core count and $M_{\min}$ the minimum satisfying the federated schedulability condition (Eq. (8)); $\kappa = 1$ is minimal provisioning and larger κ indicates more spare capacity. Let vp denote the fraction of subtasks designated as vulnerable and ap the fraction of subtasks designated as attackers.

Fig. 7 shows *System Threat* reduction for varying vp and ap: threat decreases consistently across all configurations, with larger reductions when fewer subtasks are vulnerable or when the attacker fraction is small. Fig. 8 (left) plots threat reduction versus the core multiplier κ. The results exhibit a *non-monotonic* dependence on κ: for larger p (more aggressive dependency augmentation), threat reduction can decrease as κ grows from 1 toward ≈ 2 before increasing at higher κ. The right panel of Fig. 8 aggregates threat reduction over all possible attacks for varying values of p. Increasing p enforces more serialization (reducing exploitable concurrency) but also amplifies non-monotonic behavior, analyzed in Section 8.4.

Fig. 9 shows *anomalies* – cases where the ADAG exhibits higher threat than the original DAG. These (red points) constitute only about 8% of task sets and the increase in threat for each such task is relatively small (the vertical distance of each red dot from $y = x$ is small). This aligns with Nasri et al.'s [25] observation that randomization can open new attack vectors. An offline screening step that retains only the candidate ADAGs not raising threat would mitigate these regressions, a direction we leave for future work.

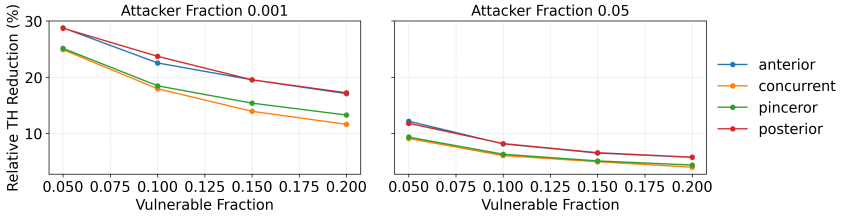


Figure 7 *System Threat* reduction as a function of the vulnerable-subtask fraction vp for different attacker-subtask fractions ap.

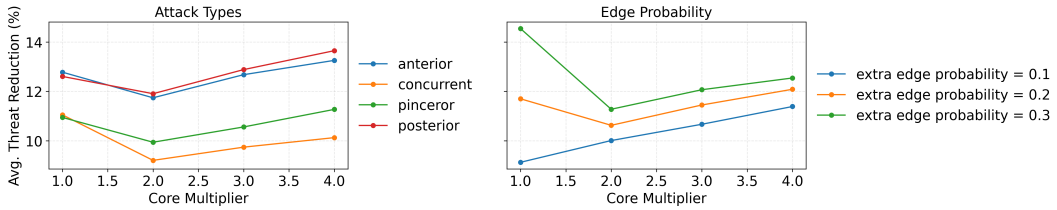


Figure 8 *System Threat* reduction as a function of the core multiplier κ: by attack type (left) and by edge probability (right).

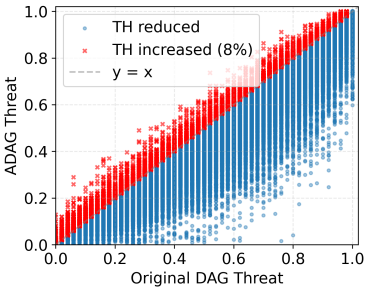


Figure 9 Original DAG threat vs. ADAG threat for each task set. Blue points (below $y{=}x$) indicate threat reduction; red points (above $y{=}x$) are anomalous cases where the ADAG increases threat.

Entropy Analysis. Beyond threat reduction, we assess how CDR affects schedule unpredictability by measuring *Task Distribution Entropy*. Since Dependency Augmentation inserts edges that serialize previously concurrent subtasks, one might expect it to reduce *Task Distribution Entropy* by constraining execution orders. To test this, Fig. 10 compares Temporal Augmentation alone with both augmentation phases combined. Both configurations show that average *Task Distribution Entropy* increases monotonically with core count, as additional slack enlarges the randomization budget. At low to moderate core counts, the combined configuration yields higher entropy than Temporal Augmentation alone. The additional edges shift subtasks to start times that temporal augmentation alone cannot reach, spreading each subtask's start-time distribution more uniformly across time slots. However, at high core counts ($\kappa = 4$), Temporal Augmentation alone surpasses the combined configuration. When resources are abundant, the extra serialization imposed by dependency edges becomes a constraint rather than a benefit: the large fake subtask budgets already provide sufficient randomization on their own, and the added edges restrict the scheduling freedom that would further increase entropy.

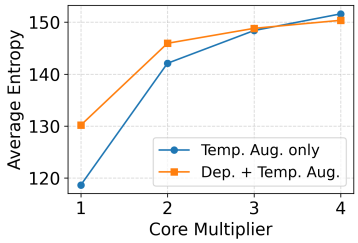


Figure 10 Average *Task Distribution Entropy* in multi-task systems versus core multiplier κ.

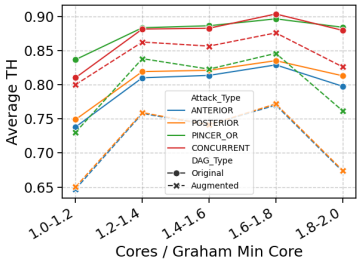


Figure 11 *System Threat* vs. κ in single-task systems (reverse-U-shaped).

8.4 Analysis of Intrinsic Security via Single-Task Scenarios

To investigate the non-monotonic behavior observed in multi-task settings, we isolate single-DAG systems, decoupling inter-task attack scenarios from the intrinsic properties of the task graph. We generated 400 DAG tasks with $U_i \in [1.5, 6.0]$ and varied the core multiplier (κ) from 1.0 to 2.0.

Fig. 11 plots *System Threat* against the core multiplier. Threat follows a reverse-U-shaped curve, highest at intermediate core counts and lowest at the extremes of the core range. Equivalently, *threat reduction* peaks when the system is either tightly or generously provisioned. This non-monotonic trend reveals that **allocating more cores does not necessarily improve security**. The counter-intuitive finding stems from two competing effects, which we term *intrinsic* and *extrinsic* security:

1. **Intrinsic security** arises from limited parallelism. Tight core budgets force more sequential execution, reducing attacker–victim overlap. Adding cores increases concurrency and opens more *vulnerable windows*, eroding this inherent protection.
2. **Extrinsic security** arises from randomization. The randomization budget grows with the number of cores m. At low core counts, even small budgets effectively shift execution windows because parallelism is limited. As m increases moderately, the loss of intrinsic security outpaces the randomization gains. At high core counts, the randomization budget becomes large enough to dominate, restoring threat reduction.

These two mechanisms explain the multi-task results (Section 8.3). Dependency edges (Fig. 8) strengthen intrinsic security by enforcing serialization within each DAG, but increased concurrency as core count grows gradually offsets this effect, reproducing the non-monotonic trend. Threat reduction is higher in single-task systems because randomization directly separates attacker–victim windows, whereas independent per-task randomization limits the ability to reduce cross-task overlap.

8.5 Runtime Overhead

Using the 400 DAGs from Section 8.4, this section presents the runtime overhead of ADAG generation, the total time spent by the NPWC scheduler (i.e., scheduling overhead) in making all the scheduling decisions in a period of an ADAG, and its comparison to that of the original DAG. We performed time measurements using the `std::chrono::steady_clock` (monotonic, nanosecond-resolution) wrapped in Resource Acquisition Is Initialization (RAII) scoped timers around each ADAG generation phase and the NPWC scheduler in our simulator. For this set of experiments, we used a mid-range desktop AMD Ryzen 5 3600 CPU with threads pinned (`OMP_PROC_BIND=close`, `OMP_PLACES=cores`). Each point in the plots of Fig. 12 represents the median for 100 trials, with the relative standard error kept below 0.4%.

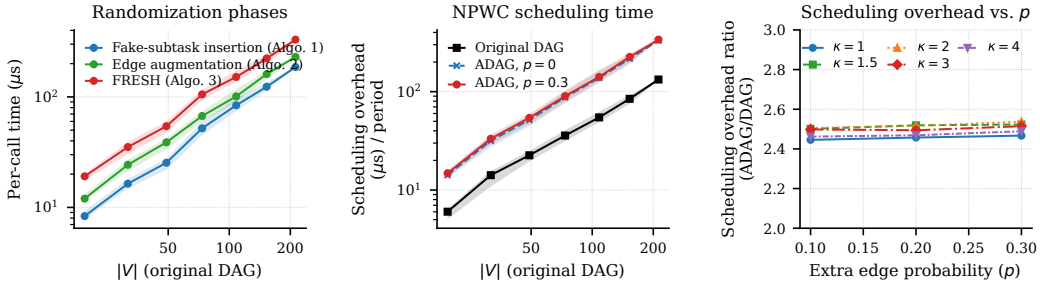


Figure 12 Left subfigure: Time taken by individual phases of ADAG generation for varying $|V|$. **Middle subfigure:** Total time taken by the NPWC scheduler (i.e., scheduling overhead) in one period for the original DAG, the ADAG with $p = 0$ (no dependency augmentation) and with $p = 0.3$ (with dependency augmentation) as a function of $|V|$. The left and middle subfigures represent the medians and the bands span the 25th–75th percentiles. **Right subfigure:** Scheduling overhead ratio (scheduling overhead of ADAG/scheduling overhead of DAG) for varying p and different κ.

We measured (the left subfigure in Fig. 12) the time taken by the three individual phases of ADAG generation: (1) fake-subtask insertion, Algo. 1, (2) Dependency Augmentation, Algo. 2, and (3) Temporal Augmentation under FRESH, Algo. 3. We set the extra edge probability $p = 0.3$ and core multiplier $\kappa = 1$. The total time by all three phases for each release of a DAG has median 232 μs with a 99th percentile below 595 μs, dominated by FRESH (median 108 μs), then fake-subtask insertion (median 55 μs), and the edge augmentation (median 69 μs at $p = 0.3$, shrinking sharply as fewer extra edges are added at smaller p). The overhead of ADAG generation thus stays in the few-hundred-microsecond range.

The total overhead of the NPWC scheduler in making all the scheduling decisions (e.g., checking predecessors' completion, deciding which ready nodes to dispatch for execution) in one period is measured (the middle sub-figure in Fig. 12) for the original DAG, for the ADAG with no dependency augmentation ($p = 0$) and also with both temporal and dependency augmentations ($p = 0.3$). The scheduling overhead of NPWC exhibits no noticeable variation with or without dependency augmentation; the curves for $p = 0$ and $p = 0.3$ in the middle subfigure nearly overlap. The difference in overhead for scheduling ADAG in comparison to the original DAG is mainly due to temporal augmentation; however, the absolute per-period total scheduling overhead stays in tens to hundreds of microseconds.

To quantify the additional scheduling overhead relative to the baseline (i.e., unaugmented DAG), we compute the ratio of the scheduling overhead of the ADAG to that of the original DAG, varying the extra-edge probability $p \in \{0.1, 0.2, 0.3\}$ across core multipliers $\kappa \in \{1, 1.5, 2, 3, 4\}$ (right subfigure of Fig. 12). The NPWC scheduler incurs approximately 2.5× higher overhead when scheduling ADAGs compared to the original DAG across the entire range of p. This increase is primarily attributable to the doubling of the number of vertices in the ADAG (i.e., total $2|V|$ nodes in an ADAG). The contribution of dependency augmentation is, as discussed above, quite minor.

CDR reduces *System Threat* in both single- and multi-task systems, at sub-millisecond per-release randomization cost and an approximately 2.5× scheduler overhead: dependency augmentation provides intrinsic security through serialization, while temporal augmentation provides extrinsic security via randomization. Resource provisioning exhibits a non-obvious trade-off – intermediate core counts may reduce randomization benefits, so systems should be either tightly or generously provisioned. In multi-task environments, per-task randomization alone does not eliminate cross-task leakage through shared microarchitectural resources; combining it with spatial isolation (e.g., cache partitioning or bandwidth throttling) would further strengthen security.

9 Discussion

Integrating into Existing Systems. CDR is a *user-space/compiler-level* transformation: the scheduler remains unchanged. At **compile time**, a source-to-source transform applies Temporal Augmentation (Algo. 1), expanding each subtask v into a fake–real pair $(v^F \to v)$ and encoding candidate dependency edges as conditional synchronization points. At **run time**, each job release selects random dependency edges E_Δ (Algo. 2) and invokes FRESH (Algo. 3) to assign random fake-subtask durations – both within the schedulability budget.

Source-to-source compilers that automatically restructure parallel programs are well established. PLuTo [7] generates tiled, OpenMP-annotated parallel code from C loop nests fully automatically. The C^3 pre-compiler [8] instruments OpenMP shared-memory programs with barriers and coordination logic for checkpointing – without modifying the underlying runtime. More directly, Ayav et al. [3] formalize fault-tolerance as automatic program transformations on hard real-time periodic tasks, inserting heartbeating and checkpointing commands while formally proving that schedulability is preserved. CDR follows the same paradigm: a compile-time structural transformation that adds a non-functional property – schedule randomization – to real-time tasks while preserving deadline guarantees.

The ADAG model naturally accommodates runtime overheads (e.g., graph traversal, FRESH execution itself). Given a known worst-case overhead δ per fake subtask, the runtime reduces its active duration to $c(v^F) - \delta$, so each fake subtask's random duration absorbs the overhead, preserving the schedulability guarantee without additional safety margins. By the same reasoning, even a non-randomized ADAG provides explicit slots for per-subtask overhead, making it a useful model for accounting for general RTS overheads.

Future Work. Two directions stand out. First, when vulnerable or attacker subtasks are known, heuristics can concentrate the fake-subtask budget on those subtasks for a more targeted defense than uniform randomization. Second, deterministic (non-random) fake-subtask assignments that provably block specific attack vectors while satisfying the schedulability bound could provide a complementary "static defense" alongside the randomized approach.

10 Conclusion

We presented CDR, a scheduler-oblivious framework that randomizes parallel real-time DAG schedules via graph transformation while preserving hard deadline guarantees. The ADAG model is provably sufficient to represent any feasible non-preemptive schedule, and two attacker-aware metrics – *System Threat* and *Task Distribution Entropy* – quantify defense against intrusive and observation-based attackers. Experiments show significant vulnerability reductions and reveal that additional cores can erode security by enabling concurrency that outpaces randomization gains, underscoring the need for security-aware resource provisioning in multicore real-time systems.

References

1 B. Andersson and J. Jonsson. The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%. In *15th Euromicro Conference on Real-Time Systems, 2003. Proceedings.*, pages 33–40, Porto, Portugal, 2003. IEEE Comput. Soc. doi:10.1109/EMRTS.2003.1212725.

2 Abdullah Al Arafat, Zhishan Guo, and Amro Awad. VR-Spy: A Side-Channel Attack on Virtual Key-Logging in VR Headsets. In *2021 IEEE Virtual Reality and 3D User Interfaces (VR)*, pages 564–572, 2021. doi:10.1109/VR50410.2021.00081.

3 Tolga Ayav, Pascal Fradet, and Alain Girault. Implementing fault-tolerance in real-time systems by automatic program transformations. In *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*, EMSOFT '06, pages 205–214, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1176887.1176917.

4 H. Baek and C. M. Kang. Scheduling randomization protocol to improve schedule entropy for multiprocessor real-time systems. *Symmetry*, 12(5):753, 2020. doi:10.3390/sym12050753.

5 S. Baroumand, A. Zaman, and L. Mihaylova. Attack detection and fault-tolerant control of interconnected cyber-physical systems against simultaneous replayed time-delay and false-data injection attacks. *Iet Control Theory and Applications*, 17(5):527–541, 2022. doi:10.1049/cth2.12393.

6 Michael Bechtel and Heechul Yun. Memory-Aware Denial-of-Service Attacks on Shared Cache in Multicore Real-Time Systems. *IEEE Transactions on Computers*, 71(9):2351–2357, 2022. doi:10.1109/TC.2021.3108044.

7 Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1375581.1375595.

8 Greg Bronevetsky, Daniel Marques, Keshav Pingali, Peter Szwed, and Martin Schulz. Application-level checkpointing for shared memory programs. *SIGPLAN Not.*, 39(11):235–247, 2004. doi:10.1145/1037187.1024421.

9 C. Chen, S. Mohan, R. Pellizzoni, and R. Bobba. On scheduler side-channels in dynamic-priority real-time systems. *CoRR*, abs/2001.06519, 2020. doi:10.48550/arxiv.2001.06519.

10 Chien-Ying Chen, Sibin Mohan, Rodolfo Pellizzoni, Rakesh B. Bobba, and Negar Kiyavash. A Novel Side-Channel in Real-Time Schedulers. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 90–102, 2019. doi:10.1109/RTAS.2019.00016.

11 Jiyang Chen, Tomasz Kloda, Ayoosh Bansal, Rohan Tabish, Chien-Ying Chen, Bo Liu, Sibin Mohan, Marco Caccamo, and Lui Sha. SchedGuard: Protecting against schedule leaks using linux containers. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 14–26, 2021. doi:10.1109/RTAS52030.2021.00010.

12 Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, 1998. doi:10.1145/277652.277725.

13 R. L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, March 1969. doi:10.1137/0117039.

14 Yi Han, Jeffrey Chan, Tansu Alpcan, and Christopher Leckie. Using Virtual Machine Allocation Policies to Defend against Co-Resident Attacks in Cloud Computing. *IEEE Transactions on Dependable and Secure Computing*, 14(1):95–108, 2017. doi:10.1109/TDSC.2015.2429132.

15 Qingqiang He, Nan Guan, Mingsong Lv, Xu Jiang, and Wanli Chang. Bounding the Response Time of DAG Tasks Using Long Paths. In *2022 IEEE Real-Time Systems Symposium (RTSS)*, pages 474–486, 2022. doi:10.1109/RTSS55097.2022.00047.

16 Nahid Juma, Jonathan Shahen, Khalid Bijon, and Mahesh Tripunitara. The Overhead from Combating Side-Channels in Cloud Systems Using VM-Scheduling. *IEEE Transactions on Dependable and Secure Computing*, 17(2):422–435, 2020. doi:10.1109/TDSC.2018.2790932.

17 Auguste Kerckhoffs. La cryptographie militaire. *J. des Sci. Militaires*, 9:161–191, 1883.

18 Jing Li, Jian Jia Chen, Kunal Agrawal, Chenyang Lu, Chris Gill, and Abusayeed Saifullah. Analysis of Federated and Global Scheduling for Parallel Real-Time Tasks. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 85–96, 2014. doi:10.1109/ECRTS.2014.23.

19 S. Liu and W. Yi. Task parameters analysis in schedule-based timing side-channel attack. *IEEE access : practical innovations, open solutions*, 8:157103–157115, 2020. doi:10.1109/access.2020.3019323.

20 J. M. López, J. L. Díaz, and D. F. García. Utilization Bounds for EDF Scheduling on Real-Time Multiprocessor Systems. *Real-Time Systems*, 28(1):39–68, 2004. doi:10.1023/B:TIME.0000033378.56741.14.

21 Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998. doi:10.1145/272991.272995.

22 Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio C. Buttazzo. Response-Time Analysis of Conditional DAG Tasks in Multiprocessor Systems. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 211–221, Lund, Sweden, July 2015. IEEE. doi:10.1109/ECRTS.2015.26.

23 Piyoosh Purushothaman Nair, Arnab Sarkar, and Santosh Biswas. Fault-Tolerant Real-Time Fair Scheduling on Multiprocessor Systems with Cold-Standby. *IEEE Transactions on Dependable and Secure Computing*, 18(4):1718–1732, 2021. doi:10.1109/TDSC.2019.2934098.

24 Girija J. Narlikar. Scheduling threads for low space requirement and good locality. In *Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '99, pages 83–95, New York, NY, USA, June 1999. Association for Computing Machinery. doi:10.1145/305619.305629.

25 Mitra Nasri, Thidapat Chantem, Gedare Bloom, and Ryan M. Gerdes. On the Pitfalls and Vulnerabilities of Schedule Randomization Against Schedule-Based Attacks. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 103–116, 2019. doi:10.1109/RTAS.2019.00017.

26 S. Ramanujan. Highly Composite Numbers. *Proceedings of the London Mathematical Society*, s2_14(1):347–409, 1915. doi:10.1112/plms/s2_14.1.347.

27 Beheshteh Raouf and Seyedamirabbas Mousavian. False data injection to conceal load-altering attacks via electric vehicles. *Sustainable Energy, Grids and Networks*, 42:101640, 2025. doi:10.1016/j.segan.2025.101640.

28 Jiankang Ren, Zheng Wang, Chi Lin, Mohammad S. Obaidat, Hongrui Xie, Haihui Zhu, Chunxiao Liu, Kaiwen Wang, and Guozhen Tan. REORDER++: Enhanced Randomized Real-Time Scheduling Strategy Against Side-Channel Attacks. *IEEE Transactions on Network Science and Engineering*, 10(6):3253–3266, 2023. doi:10.1109/TNSE.2023.3254653.

29 Maria A. Serrano, Alessandra Melani, Roberto Vargas, Andrea Marongiu, Marko Bertogna, and Eduardo Quiñones. Timing characterization of OpenMP4 tasking model. In *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 157–166, 2015. doi:10.1109/CASES.2015.7324556.

30 C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948. doi:10.1002/j.1538-7305.1948.tb01338.x.

31 Y. Song, X. Liu, Z. Li, M. Shahidehpour, and Z. Li. Intelligent data attacks against power systems using incomplete network information: A review. *Journal of Modern Power Systems and Clean Energy*, 6(4):630–641, 2018. doi:10.1007/s40565-018-0427-z.

32 David Trilla, Carles Hernandez, Jaume Abella, and Francisco 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)*, pages 1–6, 2018. doi:10.1109/DAC.2018.8465919.

33 Nils Vreman, Richard Pates, Kristin Krüger, Gerhard Fohler, and Martina Maggio. Minimizing Side-Channel Attack Vulnerability via Schedule Randomization. In *2019 IEEE 58th Conference on Decision and Control (CDC)*, pages 2928–2933, 2019. doi:10.1109/CDC40024.2019.9030144.

34 David Ward, Ireri Ibarra, and Alastair Ruddle. Threat Analysis and Risk Assessment in Automotive Cyber Security. *SAE International Journal of Passenger Cars - Electronic and Electrical Systems*, 6(2):507–513, August 2013. doi:10.4271/2013-01-1415.

35 Man-Ki Yoon, Jung-Eun Kim, Richard Bradford, and Zhong Shao. TaskShuffler++: Real-Time Schedule Randomization for Reducing Worst-Case Vulnerability to Timing Inference Attacks. doi:10.48550/arXiv.1911.07726.

36 Man-Ki Yoon, Sibin Mohan, Chien-Ying Chen, and Lui Sha. TaskShuffler: A Schedule Randomization Protocol for Obfuscation against Timing Inference Attacks in Real-Time Systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, 2016. doi:10.1109/RTAS.2016.7461362.

37 Ruochi Zhang and Parv Venkitasubramaniam. Stealthy Control Signal Attacks in Linear Quadratic Gaussian Control Systems: Detectability Reward Tradeoff. *IEEE Transactions on Information Forensics and Security*, 12(7):1555–1570, 2017. doi:10.1109/TIFS.2017.2668220.