Balancing Security and Schedulability: WCET Evaluation and Security Optimization in CPS

Zihan Li†, Marion Sudvarg†, Ching-Hsiang Chan†, Ryan Burrow‡, Nathan Burow‡,
Cailani Lemieux-Mack*, Sanjoy Baruah†, Ning Zhang†, Bryan C. Ward*
† Washington University in St. Louis ‡ MIT Lincoln Laboratory * Vanderbilt University
{tomson.li, msudvarg, c.ching-hsiang, baruah, zhang.ning}@wustl.edu,
{ryan.burrow, nathan.burow}@ll.mit.edu, {kailani.j.lemieux.mack, bryan.ward}@vanderbilt.edu

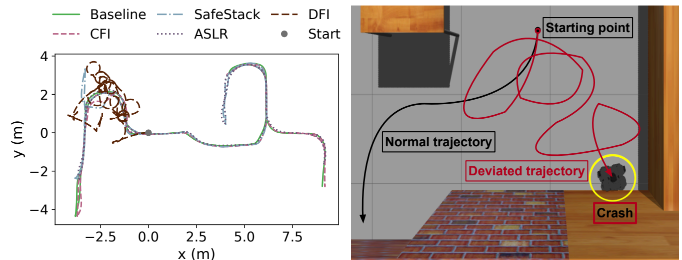
***Abstract*—As real-time cyber-physical systems (CPS) become increasingly prevalent and interconnected, ensuring their security becomes essential. Yet, integrating runtime security mechanisms into real-time systems is challenging due to non-uniform performance overheads that may differ significantly under average- and worst-case behaviors. This paper systematically examines how commonly used runtime memory-safety defenses affect worst-case task performance in real-world CPS applications. The experiments reveal that security mechanisms vary markedly in their computational impact, providing practical guidance for their deployment. They also highlight the opportunity to selectively combine defenses to enhance overall protection. The paper further presents an optimization framework, QUASAR^{RT}, and a case study on ArduPilot demonstrating how system designers can apply these findings to maximize their defense coverage while maintaining real-time schedulability. Results indicate that QUASAR^{RT} is effective at finding the most secure feasible task configurations, even under complex schedulability analysis.**

I. Introduction

As real-time and embedded systems are increasingly deployed in safety-critical, cyber-physical system (CPS) applications, such as self-driving cars [87] and medical IoT devices [32], there is a pressing need to ensure not only their timeliness but also their security to maintain safety. Indeed, recent studies indicate that potential consequences of the lack of security protections in CPS are dire. A famous pacemaker exploit could allow a remote attacker to make a victim's heart explode [10]. A vulnerable infotainment unit in the Jeep Cherokee enabled malicious CAN message injection that could stop the vehicle on the highway [60].

Gaps in Existing Understanding of Security Overhead in Real-Time CPS. Performance overhead is often one of the most important considerations when deciding whether to deploy a particular security mechanism [62, 75]. This is especially true in real-time CPS. A recent industry survey reports that real-time system developers generally rate timing predictability as more important than security [7]. This finding suggests they may be hesitant to incorporate additional security measures if doing so risks sacrificing temporal correctness.

While there have been several recent efforts to empirically understand security overheads in classical IT systems [23, 51, 62, 69, 72, 75, 79], these results are not directly applicable to real-time CPS. First, these works evaluate security mechanisms on general-purpose benchmarks with execution patterns that differ markedly from the typically recurrent and time-triggered tasks found in CPS. This periodic behavior can induce cascading timing effects that do not arise in conventional desktop or server applications. Second, since these studies target general-purpose workloads, they usually characterize only average-case execution times, and impacts on worst-case execution time (WCET) remain largely unstudied.



(a) Trajectory comparison. Data-flow integrity causes significant deviation. (b) TurtleBot crashes into the wall when using data-flow integrity.

Figure 1: TurtleBot behaviors under different security defenses.

Our Effort to Understand Real-Time Implications of Security Mechanisms. To fill this gap, we conduct an empirical study of widely used memory safety mechanisms across six real-world CPS platforms. These platforms were chosen to reflect a broad spectrum of CPS domains – including ground vehicles, aerial drones, legged robots, and autonomous driving systems – and represent realistic, widely adopted CPS software stacks used in both research [53] and industry [13, 78]. For security mechanisms, we selected control-flow integrity (CFI) [3], SafeStack [50], data-flow integrity (DFI) [24], and address space layout randomization (ASLR) [66] as representative runtime defenses because they each target distinct memory-based attack vectors and reflect a diverse range of enforcement strategies and performance characteristics.

Figure 1 underscores the importance of this study and highlights the need for a framework in which to reason about security and performance trade-offs. It illustrates the runtime behavior of a TurtleBot3 [77] – an indoor robot designed for home service – under the different security defenses we consider. Trajectories differ across mechanisms because each imposes distinct timing effects. Notably, the high overheads incurred by DFI cause the executed trajectory to depart from the planned path almost immediately (Figure 1a), and the TurtleBot eventually collides with a wall (Figure 1b).

Our Approach Towards Security Optimization in CPS.

© 2026 IEEE. Personal use of this material is permitted.
Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

These results make it clear that, although security is crucial in connected CPS, a system designer must carefully select defenses to avoid temporal behavior that compromises safety. We reason about this as a **constrained optimization problem**:

Given a CPS application, apply a set of defenses to each task so as to maximize system security while remaining schedulable.

Toward modeling and solving such a problem, this paper establishes a framework to:

1) understand how each security mechanism (and valid combinations thereof) impacts WCET, given task-dependent effects that may differ significantly from the average case;
2) establish a metric to quantify how much each defense contributes to system security; and
3) define constraints encoding feasibility conditions under the chosen scheduling policy and analysis.

1) Execution-time Overheads. To address this, we study two aspects: (i) how each defense affects average-case (ACET) and worst-case execution time (WCET) overheads, and (ii) *why* these changes occur (*e.g.,* which execution paths and memory behaviors are affected). For the former, we measure task execution times using hardware-in-the-loop (HIL) setups. We apply nonparametric bootstrapping to conduct distribution-agnostic WCET analysis over the tails of the observed distributions [58]. This provides, to a specified confidence level, an upper bound on the execution time expected to hold for a given proportion of runs. For the latter, we use run-time profiling to correlate mechanistic behavior with those measurements.

To the best of our knowledge, this is the first empirical characterization and distribution-aware analysis of timing impacts of several widely used software defenses in diverse, real-world CPS platforms. Unlike prior work [33], which uses static analysis to evaluate the WCET impact of security defenses on benchmark programs, our approach remains practical for use in real CPS software stacks.

Results show that ACET overhead and WCET expansion vary largely independently across defenses and tasks. A given mechanism can have low average cost yet a large worst-case increase, while another can raise ACET noticeably but leave the tail nearly unchanged. The same defense may also be lightweight for one task and costly to another. These findings motivate *per-task* selection and composition of defenses to balance security with real-time constraints.

2) Security Metric. To quantify the extent to which each considered defense, and combinations thereof, contribute to the security of a task, we leverage the QUASAR (Quantitative Attack Space Analysis and Reasoning) analysis tool [70]. QUASAR encodes an attack class (*e.g.,* memory-corruption attacks) as Boolean predicates over a set of capabilities necessary for an attack to succeed; these may be represented in an Attack Capability Graph (ACG). A defense is defined as a set of constraints on the ACG that disables some combination of attack capabilities. While QUASAR includes definitions for several common memory protections, we additionally encode formal expressions of SafeStack and DFI in QUASAR's modeling language. For a given ACG and combination of

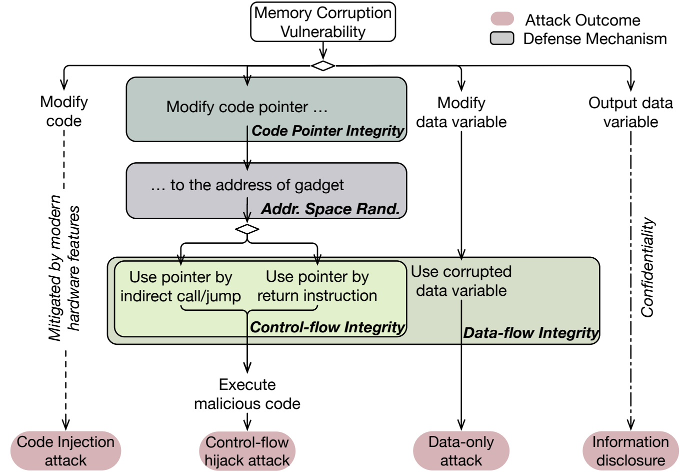


Figure 2: Attack model for memory corruption exploits.

defenses, QUASAR solves the Boolean sharp satisfiability (#SAT) problem to quantify the proportion of attack paths in the ACG blocked by those defenses. This "defense coverage" metric forms the basis of our optimization objective.

3) Optimal Defense Selection. We present an offline optimization tool, QUASAR^{RT}, that assigns combinations of security mechanisms to tasks that maximize total defense coverage while satisfying real-time schedulability constraints. It formulates and solves the constrained optimization problem as a mixed-integer linear program (MILP), supporting three scheduling policies: implicit-deadline preemptive EDF, and both preemptive and non-preemptive constrained-deadline fixed-priority (FP) scheduling. For non-preemptive FP, it also introduces binary ordering variables and associated constraints to encode an optimal priority order over tasks. Unlike prior work [33, 83], QUASAR^{RT} encodes exact analysis for several supported scheduling policies while retaining ILP tractability.

Experiments indicate that QUASAR^{RT} is effective at quickly finding an optimal assignment of security defenses to tasks, which are then instrumented offline during system design to enable online security protections. In a case study on ArduPilot [12], QUASAR^{RT} demonstrates effective targeted defense selection, successfully defending against a memory corruption attack while mitigating the control deviations that occur when high-overhead DFI checks are applied to all tasks.

II. Memory Security Threats

This section provides a detailed overview of memory-corruption-based attacks and their corresponding defense mechanisms, then categorizes the factors by which these mechanisms incur runtime overhead.

A. Memory-Based Attacks and Defenses

Threat Model. According to the CWE (Common Weakness Enumeration) ranking [28], out-of-bounds write ranked 2nd in 2024 and 1st in 2023 among the most prevalent types of attacks. Consequently, memory corruption remains one of the most persistent and exploitable software vulnerability classes [29]. Attackers can leverage flaws – such as buffer

	Data and Text	Normal Stack	ROP Stack
0xF00	"/bin/sh"	r.a.	r.a.
		epb	epb
0x205	ret	…	…
0x201	pop edx	local n	0xF00
		arg2	0x103
0x107	ret	arg1	system()
0x103	push edx	r.a.	0x201

Figure 3: In a code reuse attack, the return address is overwritten to divert control to attacker-chosen gadgets, chained together by specified return addresses on the stack. Ultimately "/bin/sh" is executed.

overflows and use-after-free errors – in a victim process to escalate privilege, bypass security checks, hijack control flow, or manipulate application logic to trigger unsafe behavior. This is particularly dangerous in real-time CPS, where software directly influences physical actions.

Multiple Potential Attack Paths. Existing works [51, 75] identify four general categories of memory-based attacks: code injection, control-flow hijacking, data-only attacks, and information disclosure. Figure 2 illustrates their unique paths, each proceeding from exploiting a low-level memory error to compromised system behavior.

In this work, we focus on control-flow hijacking and data-only attacks, as they remain highly relevant to real-time CPS platforms, where both execution integrity and data correctness are critical to safety and functionality. We also focus on runtime defenses, since these may change the timing behavior of the system. Consequently, we exclude tools like sanitizers [71] and fuzzers [2] that are meant to be used during testing.

Control-Flow Hijacking. These attacks exploit memory corruption to redirect program execution to unintended code paths. This is typically achieved by overwriting code pointers with, *e.g.,* return addresses, function pointers, or vtable entries. Modern techniques like return-oriented programming (ROP) and jump-oriented programming (JOP) allow attackers to reuse existing code within the binary to craft malicious behavior without injecting new code. There are many code reuse variants, all of which rely on *gadgets* – code snippets that perform a specific computation or operation, and which can be chained together to give an attacker control over a process.

Figure 3 illustrates how gadgets work in a code reuse attack using ROP. The left column shows excerpts from the data and text sections of the program. In particular, the string "/bin/sh" is at address 0xF00,[1] and executable bytes at addresses 0x107 and 0x205 are the ret instruction.[2] The middle column shows the stack under normal execution for the 32-bit *System V* calling convention. Assume an attacker can overwrite values on the stack due to, *e.g.,* a memory-corruption vulnerability. The attack payload used to perform a code-reuse attack and open a shell is shown in the right-hand column. By overwriting the return address with 0x201, the attacker causes system() to be popped into edx and then transfers control to 0x103; that gadget pushes edx and returns, effectively invoking system("/bin/sh").

These types of attacks are particularly concerning in CPS applications, where they can subvert task logic, bypass safety checks, or hijack control loops. CFI, SafeStack, and ASLR may be employed to mitigate this threat.

CFI [3] is a defense that detects control-flow hijacking by restricting indirect control transfers to those allowed by a program's control-flow graph (CFG). The compiler instruments such transfers (*e.g.,* indirect calls and jumps) with runtime checks; if a transfer targets an address not permitted by the CFG, the process is halted [22]. These checks and extra instructions can increase ACET and inflate WCET on worst-case paths with many monitored transfers.

SafeStack [50] is a component of Code Pointer Integrity (CPI) that mitigates stack overflows from corrupting control-flow data such as return addresses. It splits each thread's stack into *safe* and *unsafe* regions: the safe stack stores return addresses, register spills, and locals that are always accessed safely, while other objects (*e.g.,* buffers) are placed on the unsafe stack, so overflows on the unsafe stack cannot overwrite the safe one. Enforcing this separation requires compiler-inserted code to maintain a separate safe-stack pointer (typically in function prologues/epilogues), which adds instruction and memory operations and can therefore increase both ACET and WCET.

ASLR [66] is enabled by default on most modern operating systems. It is a probabilistic defense that randomizes a process's memory layout at load time, making code and data addresses less predictable and hindering an attacker's ability to locate useful gadgets or pointers. Although ASLR does not enforce correctness, it raises the practical difficulty of control-flow hijacking. Since randomization occurs during load time, ASLR typically has minimal timing impact during execution.

Data-Only Attacks. Unlike control-flow hijacking, data-only attacks maintain a program's control flow but corrupt internal variables to manipulate program behavior. These attacks target data such as status flags, configuration parameters, sensor inputs, or actuator commands – any of which, if altered, can lead to unsafe or unintended behavior in CPS applications. Because these attacks do not violate control-flow constraints, they can bypass CFI-like defenses entirely. A key defense against this class of attacks is data-flow integrity (DFI).

DFI [24] is a runtime defense that enforces data-flow correctness by ensuring each memory read originates from a legitimate write according to a statically constructed data-flow graph. This graph is built through conservative static analysis, and the program is instrumented to track and validate memory writes at runtime. If a read occurs from an unexpected source, the process is terminated. Unlike control-flow defenses, DFI protects both control-flow and non-control-flow data, offering broader coverage at the cost of higher runtime overhead.

The remaining two paths – **code injection** and **information disclosure** – are not the focus of our evaluation. *Direct* code-injection attacks are largely mitigated on modern CPS plat-

[1] Note that in practice, libc *does* actually contain this string.

[2] x86 uses a variable-length instruction set, so processors do not impose alignment constraints on instructions. Consequently, these may not be ret instructions emitted by the assembler, but bytes from the end/beginning of other instructions that happen to encode a ret [44].

forms by MMU-enforced memory protections, such as Data Execution Prevention (DEP) [59] and Write XOR Execute ($W \oplus X$) [65], which mark writable regions as non-executable and therefore prevent injected payloads from running as native code. However, these mechanisms do not fully eliminate exploitation: attackers might work around non-executable memory using code-reuse techniques (e.g., ROP/JOP) and can sometimes bypass or disable protections for specific mappings through vulnerabilities or misconfigurations. Accordingly, our study treats DEP and $W \oplus X$ as baseline platform hardening that reduces the prevalence of *direct* injection and instead focuses on the timing impact of defenses targeting control-flow and data-only attacks, which remain relevant under code-reuse attack models. Information-disclosure attacks, while critical for bypassing defenses such as ASLR, primarily serve as enablers in exploit chains. Our paper emphasizes runtime performance, so it does not evaluate leak-prevention mechanisms directly, though it considers how ASLR's memory-layout randomization may affect execution times.

B. Categorization of Security-Induced Runtime Overhead

The runtime overhead of security defenses arises from three primary factors: the frequency ω and complexity C of integrity checks, and any constant costs Δ incurred. Frequency describes how often a defense's instrumentation intervenes during job execution, while complexity captures the computational overhead of each individual intervention, *e.g.,* a simple bounds check or metadata lookup and validation. Constant costs arise from additional bookkeeping structures, metadata, or memory-layout changes that persist across executions. Thus, overhead can be formulated in terms of these factors as:

$$\text{overhead} \propto \omega \cdot C + \Delta$$

These factors manifest differently across defenses. High-frequency defenses may instrument nearly every instruction; others apply checks only to memory loads and stores, or even more specifically to pointer operations. Their fidelity and complexity also affect their overhead; more selective schemes restrict checks to only code pointers, and the most lightweight variants check only indirect branches, such as virtual or indirect function calls. Defenses that perform fewer and coarser checks tend to incur lower average overhead.

Some defenses also have significant constant costs. For example, DFI not only increases ω and C through per-access validation of loads and stores, but also introduces a large Δ due to the additional runtime metadata it maintains (*e.g.,* shadow state and tables), which increases the steady-state memory footprint. In contrast, CFI avoids runtime metadata entirely, keeping Δ low. Other mechanisms introduce overhead indirectly by restructuring the memory layout, *e.g.,* SafeStack's splitting of a program's stack. Although layout transformations may not add new instructions, they can still affect how the processor interacts with the memory hierarchy, influencing cache and TLB locality. The impact of such changes is architecture-dependent and often non-deterministic.

Table I: Details of CPS Platforms and Selected Critical Tasks

CPS App.	Hardware	Tasks
TurtleBot3 (T)	Raspberry Pi 4	MoveBase AmclNode (AMCL)
Jackal UGV (J)	Intel NUC 8	HandleLaserScan (LsrScan)
Humanoid (H)	Intel NUC 8	RobotisController (RobotisCtrl)
Quadruped Robot (Q)	Jetson AGX Orin	UnitreeGuide::junior_ctrl (Ctrl)
ArduPilot (Ar)	Raspberry Pi 4	Copter::fast_loop (FastLp) Copter::update_GPS (UpdGPS)
Autoware (Auto)	i9-12900K w/ 4070Ti Super	trajectory_follower (TrajFollow) object_validation (ObjValid)

III. Timing Impacts from Security Mechanisms

We conducted a systematic evaluation of the four representative security mechanisms from our defense taxonomy in Section II-A (ASLR, CFI, DFI, and SafeStack) to assess the quantitative impacts of their overheads on WCET, and correspondingly, their suitability for use in real-time systems.

A. Experimental Setup

Target CPS Applications. To evaluate the real-time impact of runtime defenses, we studied six real-world CPS platforms: TurtleBot3 [77], Jackal UGV [46], ROBOTIS OP3 Humanoid [68], Unitree Quadruped [78], ArduPilot [12], and Autoware [13]. These cover diverse CPS domains, including unmanned ground vehicles (UGVs), legged robots, unmanned aerial vehicles (UAVs), and autonomous driving. Each platform was run in hardware-in-the-loop (HIL) simulation with its respective scenarios, and we recorded execution times of one or two selected critical control tasks, spanning perception, localization, planning, and control loops.

Target Computing Platforms. Evaluations were conducted on systems recommended by the CPS platform vendors. In total, we used four different computing platforms, including a Raspberry Pi 4, an Intel NUC 8 with a 4-Core Intel® Core™ i7-8559U CPU and 8GB RAM, an NVIDIA Jetson AGX Orin, and an x86 system with a 16-core Intel® Core™ i9-12900K CPU with 128GB RAM and an NVIDIA 4070 Ti SUPER GPU. For most applications, we used the hardware specified in their documentation. For some, we chose the hardware closest to the recommended system requirements. For instance, one of the reference platforms listed by Autoware is the ADLINK in-vehicle computer [14] with an Intel® Core™ 13/12th Gen processor, an NVIDIA RTX 4000 SFF Ada GPU, and 128GB RAM, which is similar to our setup. Table I summarizes the CPS applications, computing platforms, and tasks evaluated.

Target Security Mechanisms. All evaluated defenses are open source. We used the released CFI and SafeStack implementations from LLVM version 13.0.1 [56], while DFI is a research prototype based on LLVM 13.0.0 [84]. To maintain fair comparisons, when evaluating each defense, we kept all compiler options (*e.g.,* optimization level) that were also enabled when measuring baseline performance, even those not related to the defense. Since ASLR is deployed by default in most modern Linux distributions, to evaluate it individually, we disabled it by changing the

randomize_va_space kernel variable for all experiments other than those that specifically tested ASLR.[3]

Measurement Methodology. The overarching objective of our experiments is to quantify the real-time performance impacts of all considered defenses. Each platform was run 1,000 times with each defense, and without any defense applied to establish a baseline. To minimize the impact from computationally expensive simulation environments (*e.g.,* Gazebo), we establish a hardware-in-the-loop (HIL) setup. A separate machine equipped with an 18-core Intel® Core™ i9-7980XE CPU, 64 GB RAM, and an NVIDIA RTX 3090 GPU runs the simulations, while the CPS platform software stacks execute on their specified computing platforms from Table I.

For each CPS platform, we used official simulation scenarios and missions provided with their release. For drones and autonomous vehicles, these involve reaching a fixed set of waypoints. For robots, missions consist of a sequence of actions, such as standing up and walking. To broaden scenario and operating-condition coverage, we also randomize the waypoint/action order at the start of each simulation, producing a different mission instance per run and enabling execution-time measurements over a wider range of mission characteristics. This randomization is applied only during mission generation (*i.e.,* before execution begins); it does not permute, reshuffle, or reorder the timing data collected from any run – the recorded timing traces preserve their original temporal order.

Timing data for all tests was measured using the Linux clock_gettime() function with the high-resolution process timer. This method was chosen over wall-clock timing functions such as time(), as it allows us to capture each task's actual computational workload by excluding non-recurring initialization and loading overheads and preemption-based idle intervals. In addition to end-to-end timing, we used the Linux *perf* subsystem to record hardware events and call stacks, which provide insight into the observed worst-case and typical execution paths. To reduce background noise, we conducted all experiments in isolation. We minimized the system's running services, *e.g.,* by disabling mail services and window managers. Moreover, unlike traditional approaches that flush caches between executions, we retained the cache state across invocations to better reflect realistic runtime behavior. This design choice aligns with common WCET analysis assumptions, where cache-related preemption and cold-start effects are modeled separately [86]. However, retaining cache state can lead to slight measurement variance across runs due to the residual impacts from prior executions. To provide a further systematic evaluation and identify the root causes of defense overheads, we applied static analysis to the codebase of the CPS applications we evaluated.

Distribution-Based Timing Analysis. Traditional static WCET analysis remains difficult to apply to real-world CPS because their software stacks and execution environments are highly dynamic (*e.g.,* due to OS scheduling and interrupts, complex middleware, and complex microarchitectural effects), which violates the simplifying assumptions needed to derive tight, trustworthy bounds [4, 57]. To quantify how each defense affects timing behavior, we estimate both average-case (ACET) and worst-case execution times (WCET) using nonparametric bootstrapping over the observed measurements. This approach enables (i) construction of high-confidence upper bounds on execution time for optimization and schedulability analysis and (ii) assessment of whether a defense induces statistically significant overhead.

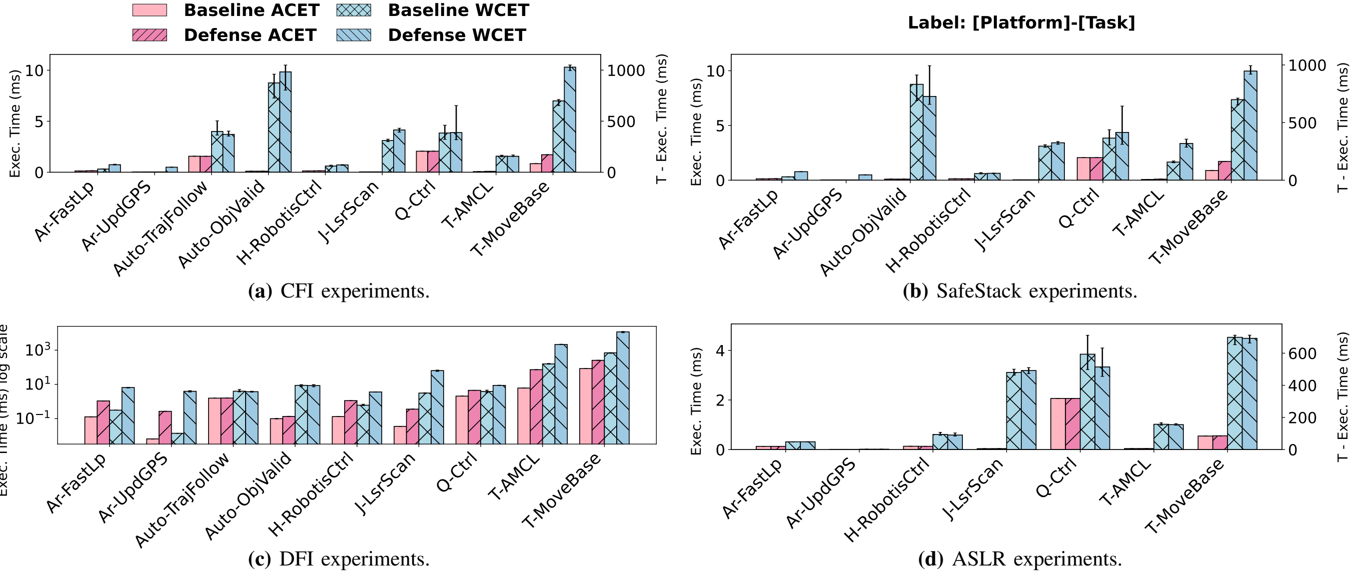
For each task/defense pair, we bound the ground-truth execution time distributions using nonparametric bootstrapping (the state-of-the-art approach from Marković et al. in [58]), which provides a sound metric by which to obtain a set of distribution-agnostic execution times for a given (low) exceedance probability to a high degree of confidence. We target the upper-tail quantile $q_{1-10^{-p}}$ (*e.g.,* $p=4$ for a $\leq 10^{-4}$ probability of exceedance) and draw $B=1000$ resamples with replacement. For each resample, we recompute the sample quantile and take the *upper* 95% confidence limit from the bootstrap distribution as a conservative WCET estimate to parameterize the schedulability constraints during optimization. We report the bootstrap median (50$^{\text{th}}$ percentile) as the point estimate for both ACET and WCET and show their confidence intervals.

B. Results and Analysis

Figure 4 presents the measured ACET and WCET statistics for baseline execution (*i.e.,* without applying any defense) and execution with defenses instrumented. Bars represent the reported bootstrap median and 95% confidence intervals for each considered defense/task combination. When WCET confidence intervals do not overlap between baseline and defensive execution, the overhead incurred by the security mechanisms is statistically significant. The abbreviations of each platform are defined in Table I.

In the analysis below, we distinguish between (i) the *typical* execution path, *i.e.,* the path most frequently taken and thus dominant in the ACET, and (ii) the *maximal* execution path, *i.e.,* that observed in runs with the largest execution times, which dominates the WCET. In general, for information-flow checking mechanisms (such as CFI and DFI), the overhead correlates with the amount of instrumentation applied to security-sensitive operations along the execution path. If these are similar along the typical and maximal paths, then the resulting ACET and WCET inflation relative to the baseline will be roughly the same. Otherwise, when the maximal path contains a higher density of such operations, it will exhibit proportionally larger WCET inflation. On the other hand, SafeStack's split-stack transformation is largely structural and deterministic. For most tasks, it yields near-zero ACET overhead and minor WCET changes, though in some cases, it incurs higher overhead when many variables and address-taken locals must be moved to the unsafe stack. ASLR introduces negligible ACET and WCET changes because randomization occurs once at load time and adds no per-access checks. Based on these distinct characteristics,

[3]Due to a bug in Autoware's ROS node initialization, we could not evaluate it with ASLR disabled. Thus, all Autoware experiments use ASLR.



(a) CFI experiments.

(b) SafeStack experiments.

(c) DFI experiments.

(d) ASLR experiments.

Figure 4: ACET vs. WCET for four types of defenses across CPS platforms. TurtleBot (T) execution times use the right y-axes of Subfigures **(a)**, **(b)**, and **(d)**. Platform and task abbreviations are listed in Table I.

Figure 5: CFI checks extend the worst-case execution paths.

we organize the observations below by mapping each to a common defense class, ranging from deterministic information-flow enforcement mechanisms to probabilistic exploit-mitigation via randomization.

Observation 1. *Though CFI often shows similar WCET expansion to ACET overhead, several tasks exhibit* ***larger*** *worst-case expansion when CFI checks align with maximal paths.*

For example, in TurtleBot *MoveBase*, WCET expands by 156.13% versus a 139.73% ACET overhead; in *Humanoid*, WCET increases by 9.63% versus 8.26% for ACET (Figure 4a). To investigate the root cause, we used the Linux *perf* subsystem to record execution samples over a fixed period. In *TurtleBot MoveBase*, the maximal path (scoreTrajectory → lineCost) accounts for 59.7% of samples in the CFI build versus 45.2% in the baseline (Figure 5). In the CFI binary, additional frames (*e.g.,* __cfi_check, __cfi_slowpath_diag) also appear frequently on this path, directly increasing the path's cost and dominance, which in turn raises WCET more than ACET. By contrast, when CFI checks execute on *typical* paths (*e.g.,* TurtleBot *AMCL*), ACET and WCET grow by similar factors. In Figure 6, CFI adds a function entry stub to most functions (with .cfi suffix) on the typical path. This explains why some tasks exhibit tightly coupled ACET/WCET growth, while others show pronounced worst-case amplification: it depends on where the CFI checks land relative to the program's maximal path. Using static analysis, we also found that in most cases (∼80%), ACET overhead and worst-case expansion are similar when CFI checks are inserted within the program (*i.e.,* via compiler-generated .cfi local entry stub). The remaining ∼20% exhibit larger WCET expansion because they additionally invoke CFI helper checks from external runtime libraries during cross-DSO (dynamic shared objects) calls, resulting in roughly 30% more such checks.

Observation 2. *DFI dominates average and worst-case costs, with WCET expansion often far exceeding ACET overhead.*

As another information-flow enforcement mechanism, DFI provides more coverage than CFI; it performs more memory operations and checks metadata at runtime. This adds instructions and extra memory traffic on typical paths, increasing cache/TLB misses. Unlike CFI, *perf* reported that most tasks have DFI checks on the typical paths. Thus, DFI typically produces the largest ACET and WCET expansions (Figure 4c). In several tasks, the worst-case cost grows far more than the mean (*e.g.,* ACET overhead reaches >10×, while WCET reaches >100× the baseline). Static analysis shows that DFI instruments ∼167% extra instructions on average, reaching >200% in some tasks. In our simulations, we observed clear functional signs of overload: with DFI enabled for all tasks, TurtleBot deviated from its planned trajectory and collided with a wall (Figure 1), and ArduPilot exhibited a significantly longer pre-arm phase and higher control deviation during flight.

Observation 3. *SafeStack generally incurs near-zero ACET overhead and small WCET expansion. Exceptions occur when*

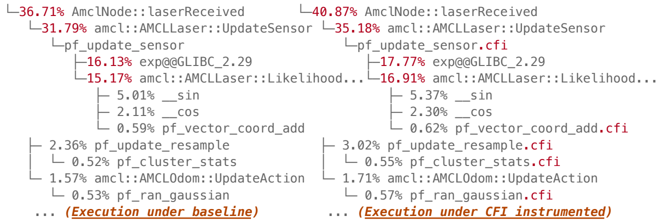


Figure 6: CFI checks on normal execution paths.

code ***heavily*** *uses large or address-taken local variables that SafeStack moves to the unsafe stack.*

SafeStack splits the stack: return addresses, register spills, and small scalars stay on a compact *safe* stack, while large or address-taken local variables move to an *unsafe* stack. For most tasks, this improves locality of control data and yields near-zero ACET overhead with small changes in WCET (Figure 4b). For example, in Autoware's *objectValidation* task, the median WCET decreases by 12.6% from baseline, consistent with the average cache-miss rates shown by *perf*: 13.86% at baseline versus 12.6% with SafeStack. For more significant outliers, such as TurtleBot *AMCL*, many temporaries are large or address-taken, so the hot working set spans two stacks; we observe a 102.5% WCET expansion for *AMCL* and 35.5% for *MoveBase*. *Perf* also reports ∼25% more retired instructions with SafeStack, consistent with extra prologue/epilogue work, unsafe-stack pointer maintenance, and additional memory traffic. Using static analysis, we also found that most cases (∼90%) have few address-taken locals, while ∼10% have heavy use of those. These effects increase L1/L2 cache and dTLB pressure and, in turn, raise ACET and widen the tail.

Observation 4. *ASLR is lightweight in both ACET and WCET, but introduces modest run-to-run variability.*

ASLR randomizes base addresses once at program start and adds no per-instruction checks. In our experiments, most tasks show no statistically significant difference in ACET or WCET expansion as the ASLR and baseline binaries are identical (Figure 4d). Although randomization occurs at page-level granularity, it can still perturb microarchitectural state across runs: (i) new virtual page numbers map to different i/dTLB sets, (ii) different physical frames chosen by the kernel change LLC set "colors," and (iii) allocator and stack behavior can shift object offsets within pages, affecting L1 set selection. As a result, ACET typically remains unchanged while tails may widen slightly due to cache/TLB/branch predictor effects. For this reason, real-time evaluations often disable ASLR to improve reproducibility and reduce non-determinism [38].

C. Takeaways

Takeaway 1. *Worst-case timing is not always proportional to average-case overhead.*

Defenses with similar ACET overheads among tasks can lead to very different WCET outcomes: if the additional instructions and memory traffic fall on the maximal path, WCET grows more than ACET; if they fall only on the typical path, ACET and WCET tend to increase by similar factors. These characteristics are hard to capture in benchmark workloads. For instance, the average performance overhead of SafeStack, as measured by Kuznetsov et al. [50], is less than 0.1% across various benchmarks. However, we observed WCET expansion up to 102%, indicating that even defenses with negligible average overhead can substantially inflate worst-case execution times. We highlight three recurring sensitivities:

- *CFI* adds checks to indirect and virtual calls. When these are on the maximal and not the typical path, WCET increases more than ACET; when they are on the typical path, CFI affects the average and worst case similarly.
- *DFI* validates many memory accesses and is most expensive when the maximal path is memory-intensive, leading to large increases in both ACET and WCET.
- *SafeStack* is cheap when the typical path uses small, non-address-taken locals, but WCET can grow more than ACET when large or address-taken locals force frequent accesses to the unsafe stack on the maximal path.

Takeaway 2. *Program structure drives different impacts.*

While worst-case timing is governed by the code executed along the maximal path, the overall timing impact of a defense is shaped by program structure. For example, in the Quadruped *UnitreeGuide* task, the typical path is largely straight-line code with direct calls, and frequently accessed locals are small and not address-taken; accordingly, CFI triggers few checks overall, and SafeStack keeps most data on the compact safe stack. The observed WCET expansions, 1.21% (CFI) and 13.54% (SafeStack), have 95% confidence intervals that overlap with the baseline, suggesting no statistically significant increase. Conversely, when a task contains few indirect transfers or address-taken locals, the impact is minimal because little instrumentation is exercised at runtime.

Takeaway 3. *The fidelity of integrity checks impacts both ACET overhead and WCET expansion.*

As checks get finer and fire more often, they add instructions and memory traffic, touch more cache lines and pages, and increase both mean and worst-case times. For instance, in the *UnitreeGuide* task for the Quadruped Robot, with few indirect calls on the typical path, CFI adds only a small cost (∼0.1% ACET, ∼1.21% WCET). In contrast, DFI validates many memory accesses against metadata (∼118% ACET, ∼127% WCET), which increases instruction count, metadata reads/writes, and cache/TLB activity. Finer-grained checking thus produces higher ACET and larger WCET, with the tail potentially higher and less predictable across runs.

Takeaway 4. *Not all defenses are universally applicable across CPS applications.*

In several tasks and platforms, certain defenses could not be enabled, or system features could not be disabled, due to software assumptions, code patterns, or toolchain/runtime constraints. For example, applying SafeStack to Autoware's *trajectoryFollower* task caused a segmentation fault at startup

due to the unsafe stack not being initialized when the thread starts. In addition, as noted in III-A, we were unable to disable ASLR for Autoware because some of its ROS nodes derive their names from an address-based signature; with ASLR disabled, multiple instances received identical names, resulting in a launch-time name collision and system failure. These cases highlight that certain parts of the CPS software stack incorporate address/layout assumptions, suggesting that defenses should be integrated via security-real-time co-design, with compatibility analysis and timing validation conducted on the target platform rather than relying on a uniform switch.

IV. QUASAR^{RT}: Optimizing Defense Coverage

Building on our observations, we next show how to use these results to provide security while respecting real-time constraints. If a system has abundant slack time, then stronger, higher-overhead defenses may be preferable to defenses that do not provide as much security. Conversely, for a system with high utilization, only low-overhead defenses may be feasible. This gives rise to an important optimization problem: how can we maximize the overall security of a real-time system while maintaining schedulability? We develop a new optimization framework, QUASAR^{RT}, to answer this question, then present experiments informed by our empirical performance results that demonstrate its effectiveness.

A. Defense Coverage Metric

Quantifying security is widely known to be a difficult and often imprecise endeavor [43]. Attackers do not comply with stochastic models, and new attack techniques are constantly being developed that undermine previously held assumptions [75]. Nonetheless, efforts at quantifying security can still be useful towards designing new defenses and encourage the adoption of stronger security postures.

QUASAR. We consider the defense-coverage metric given by the Quantitative Attack Space Analysis and Reasoning (QUASAR) tool [70]. QUASAR is based on an expert-developed model of fine-grained individual attack capabilities that are required to carry out a given class of attacks. The model is synthesized into a Boolean formula – represented by an Attack Capability Graph (ACG) – in which any solution corresponds to a valid attack. Defenses impose constraints on the ACG which can be incorporated into the Boolean formula, thereby limiting the number of solutions. While generally the problem of Boolean solution counting (#SAT) is hard, QUASAR compiles the Boolean formula into a Smoothed Deterministic Decomposable Negation Normal Form (SDDNNF) [30] using the C2D #SAT solver [31]. This transformation is efficient in practice and enables polynomial-time queries. We use QUASAR to determine the defense coverage provided by individual security mechanisms and combinations thereof, defined as the proportion of attack paths eliminated by the selected defense. QUASAR may be instructed to analyze every combination of selected mechanisms. We then prune equivalent combinations, leaving just the minimal subsets from each equivalence class.

```
dcap SafeStack assures
        Memory_Layout_Known I {Memory_Image_Known} and
        (Memory_Disclosure.Leakage_Type implies
          (Memory_Accessed.Memory_Readable I {Data.Memory_Accessed} and
           Location.Data.Memory_Accessed I {Stack.Location.Data}))
```

Figure 7: SafeStack specified with QUASAR's model language [70].

Defense Coverage of Selected Security Mechanisms. QUASAR natively synthesizes a complete ACG for memory-corruption attacks such as control-flow hijacking. Moreover, of the four defenses we consider, ASLR and CFI are already encoded; in particular, our CFI implementation is represented by its "Fine-Grained Control Flow Integrity" defense. We used QUASAR's domain-specific language to express the constraints imposed by SafeStack and DFI. Figure 7 shows the specification for SafeStack, which assures that an attacker must learn a program's memory layout via its memory (not disk) image, and that if this is gleaned via memory disclosure, then it must be accessed through stack data.

B. Constrained Optimization

We build upon the results of QUASAR and incorporate the defensive overheads from our previous experimental evaluations to develop a new defensive optimization framework, QUASAR^{RT}. For a given collection of tasks and defenses, it provides an assignment of exactly one defense strategy (*e.g.,* a single security mechanism, a valid combination, or the unprotected baseline program) to each task to maximize the sum of defense coverage across tasks (weighted by each task's relative security criticality) while guaranteeing that the system remains schedulable. In this context, security criticality indicates the importance of securing a given task relative to the others. For example, a process that handles cryptographic keys or sensitive information may be more security critical than an infotainment system. Although security and safety criticality might be linked in some CPS, these are independent concepts [8, 9]. Ultimately, the objective is to

$$\max_{\{x_{i,j}\in\{0,1\}\}} \quad \sum_{x_{i,j}} \left(x_{i,j} \cdot w_i \cdot s_{i,j}\right) \tag{1a}$$

$$\text{s.t.} \quad \forall_{\tau_i}, \quad \sum_j x_{i,j} = 1 \tag{1b}$$

$$\text{and} \quad \forall_{\tau_i}, \quad C_i = \sum_j x_{i,j} \cdot c_{i,j} \tag{1c}$$

$$\text{and} \quad \Gamma \text{ is schedulable} \tag{1d}$$

with notation defined in Table II. Although multiple defenses might be applied to a single task, Constraint (1b) restricts each task to augmentation by *exactly one* defense. This is because (i) defense coverage is not additive, as logical expressions may be shared in common among descriptions of unique defenses, and so the attack paths they block are not disjoint [70]. (ii) Overheads are also not additive; as we've shown, thorough quantification of the WCET behavior induced by each defense is necessary. And (iii) some combinations of defenses do not form valid compositions. However, as in QUASAR, one may evaluate valid combinations of security mechanisms to characterize the composition as its own *unique* defense.

Table II: Notation

Symbol	Explanation
$\tau_i = (T_i, w_i)$	A task.
$T_i \in \mathbb{N}^+$	The period of task τ_i.
$w_i \in \mathbb{N}^+$	The security criticality of task τ_i.
$d_j = (s_j)$	A defense.
$s_j \in \mathbb{Q}^{\geq 0}$	The defense's coverage, *i.e.,* the proportion of attack paths it restricts per QUASAR [70].
$c_{i,j} \in \mathbb{R}^+$	The WCET of task τ_i when defense d_j is applied.
$C_i \in \mathbb{R}^+$	The WCET of task τ_i for the selected defense.
$x_{i,j} \in \{0, 1\}$	A binary variable indicating that task τ_i has been augmented with defense d_j.

We note that by defining defense d_0 as the *base implementation* with defense coverage $s_0 = 0$, we can represent the case where a task has no defense assigned to it. $c_{i,0}$ is then the worst-case execution time of the unmodified task τ_i.

Constraint (1c) defines the worst-case execution time of each task as it arises from the selected defense. Given the resulting values C_i, Constraint (1d) ensures that the task system remains schedulable under the considered scheduling policy.

Generality. QUASAR$^{\text{RT}}$ is built in a modular fashion, taking defenses' coverage and tasks' WCETs as inputs to an optimization problem. Although it uses the ACG-coverage-based security metric defined by QUASAR, it simply treats this as a reward value, maximizing total (weighted) reward among selected security mechanisms. Therefore, it can easily be modified to use any security metric. Similarly, although we use estimated WCETs based on nonparametric bootstrapping over the tails of measured execution time distributions, any notion of WCET (including hard upper bounds derived with static analysis) is accepted. We note, however, that QUASAR$^{\text{RT}}$ is limited in its ability to handle non-independence, *i.e.,* WCETs are treated independently for purposes of response-time analysis and schedulability. An axiomatic and correlation-aware analysis of probabilistic WCET (pWCET) – *e.g.,* that of Bozhko et al. [19] or Marković et al. [58] – is left to future work.

C. Scheduling Models

Here, we describe QUASAR$^{\text{RT}}$'s instantiations of the optimization problem (1) for uniprocessor EDF and FP scheduling. These canonical policies were selected to support our case study on ArduPilot in Section IV-E. ArduPilot uses a single-threaded executor; thus, it is a modern CPS application for which uniprocessor scheduling applies.

Preemptive EDF with Implicit Deadlines. A set Γ of Liu and Layland tasks $\tau_i = (C_i, T_i)$, *i.e.,* with implicit deadlines $D_i = T_i$ are schedulable by preemptive EDF on a uniprocessor if and only if the total utilization does not exceed 1. In such systems, Constraint (1d) in QUASAR$^{\text{RT}}$'s constrained optimization problem can be expressed as

$$\sum_i \frac{C_i}{T_i} = \sum_{x_{i,j}} \frac{x_{i,j} \cdot c_{i,j}}{T_i} \leq 1 \tag{2}$$

QUASAR$^{\text{RT}}$ encodes this optimization problem as a mixed-integer linear program (MILP) over the binary variables $x_{i,j}$.

Preemptive FP Scheduling. A fixed-priority, preemptive task system Γ is schedulable [47] if and only if, for each $\tau_i \in \Gamma$, there exists some value of $t \leq D_i$ for which:

$$t \geq C_i + \sum_{k<i} \left\lceil \frac{t}{T_k} \right\rceil \cdot C_k \tag{3}$$

where tasks are indexed by decreasing priority, *e.g.,* in deadline-monotonic (DM) order. The smallest value of t satisfying this relationship is the task's *response time*; finding such a value of t for each task is referred to as *response-time analysis* (RTA). Thus, Constraint (1d) may be expressed as:

$$\forall_{\tau_i}, \quad \exists t_i < D_i \tag{4a}$$

$$\text{s.t.} \quad t_i \geq \sum_j x_{i,j} \cdot c_{i,j} + \sum_j \sum_{k<i} \left\lceil \frac{t_i}{T_k} \right\rceil \cdot x_{k,j} \cdot c_{k,j} \tag{4b}$$

The resulting optimization problem can be represented as an MILP by extending the representation of RTA given by Baruah and Ekberg in [15]. There, they introduce integer variables $Z_{i,k}$ to represent each term $\left\lceil \frac{t_i}{T_k} \right\rceil$. To enforce this intended interpretation, constraints of the form

$$Z_{i,k} > \left\lceil \frac{t_i}{T_k} \right\rceil$$

are added. Because each $Z_{i,k}$ variable is an integer, this respects the ceiling operator that appears in Equation (3).

The addition of the $x_{k,j}$ terms in Constraint (4b) naïvely give rise to a quadratic program. However, because the $x_{k,j}$ terms are binary variables, the constraint may be linearized. We define a large constant value M and for each triple (i, j, k) for $k < i$, we add an integer variable $z_{i,j,k}$ constrained as:

$$z_{i,j,k} > 0 \quad \text{and} \quad z_{i,j,k} \geq c_{k,j} \cdot \frac{t_i}{T_k} + M \cdot (x_{k,j} - 1) \tag{5}$$

This way, if $x_{k,j}$ takes the value 1, the term $M \cdot (x_{k,j} - 1)$ will evaluate to 0 and so $z_{i,j,k}$ will be forced to take the value $\left\lceil \frac{t_i}{T_k} \right\rceil$. If $x_{k,j}$ is instead 0, the term $M \cdot (x_{k,j} - 1)$ will take the value $-M$. If M is chosen to be sufficiently large, the expression $c_{k,j} \cdot \frac{t_i}{T_k} + M \cdot (x_{k,j} - 1)$ evaluates to a negative value, and thus, $z_{i,j,k}$ will be forced to 0.

Non-Preemptive FP Scheduling. A growing number of CPS applications have task systems that are scheduled non-preemptively. For example, the ArduPilot [12] platform uses a single-threaded executor with a task dispatching loop that synchronously calls a function corresponding to the next job scheduled for execution, which then returns to the loop on completion. The other CPS applications evaluated in this work run atop ROS or ROS2; in a recent paper, Teper et al. demonstrate the possibility of using the ROS2 events executor in a manner that is compatible with priority-based, non-preemptive scheduling [76].

The schedulability condition for non-preemptive FP task systems is similar to the one for preemptive systems (4b),

except that for each task τ_i, one must add a blocking term to capture the worst-case interference from a job of a lower-priority task already executing before the release of the τ_i [88]:

$$\forall_{\tau_i},\ \exists t_i \leq D_i \text{ s.t. } t_i \geq C_i + \sum_{k<i} \left\lceil \frac{t_i}{T_k} \right\rceil \cdot C_k + \max_{k>i}\{C_k\} \quad (6)$$

This can be augmented with defense-selection variables $x_{i,j}$ similarly to Constraint (4b). To represent the blocking term $\max_{k>i}\{C_k\}$, we introduce a continuous variable B_i. For every triple (i, j, k) for $k > i$, we add a linear constraint of the form:

$$B_i \geq x_{k,j} \cdot c_{k,j} \quad (7)$$

Priority Selection. Due to blocking terms, DM priority assignments are not generally optimal [26]. We note that the $k < i$ and $k > i$ indexing conditions on the $\sum$ and max terms of Constraint (6) represent those tasks τ_k with higher and lower priorities than τ_i, respectively. We can therefore augment the MILP with an additional variable $h_{i,k}$ for every pair of tasks (τ_i, τ_k) with the interpretation that it takes the value 1 if τ_i is assigned a higher priority than τ_k, and 0 otherwise. Then Constraints (6) and (7) may be stated instead as:

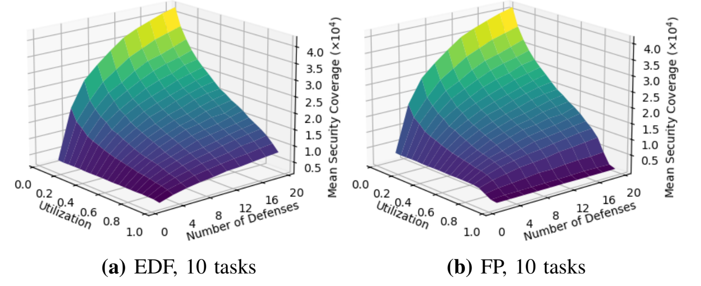
$$t_i \geq C_i + \sum_j \sum_{k \neq i} \left\lceil \frac{t_i}{T_k} \right\rceil h_{k,i} \cdot x_{k,j} \cdot c_{k,j} + \max_{j,k \neq i}\{h_{i,k} \cdot x_{k,j} \cdot c_{k,j}\}$$

To enforce antisymmetry over the total ordering of priority – exactly one of $h_{i,k}$ and $h_{k,i}$ may be 1 – we add constraints of the form $h_{i,k} + h_{k,i} = 1$ for every distinct pair (τ_i, τ_k). To enforce transitivity – that if τ_i is higher priority than τ_j, which is higher priority than τ_k, then τ_i must have a higher priority than τ_k – we add constraints of the form $h_{i,j} + h_{j,k} - 1 \leq h_{i,k}$.

Because the variables $h_{k,i}$ and $x_{k,j}$ are binary, their product forms a logical AND ($\wedge$), so linearizing is straightforward: for each triple $(i, j, k), i \neq k$, we introduce a binary variable $y_{i,j,k}$ constrained as $y_{i,j,k} \geq h_{k,i} + x_{k,j} - 1$. Linearizing the resulting terms $\left\lceil \frac{t_i}{T_k} \right\rceil \cdot y_{i,j,k}$ can be done similarly as before.

Discussion The analysis presented here serves as a starting point for other scheduling policies, including those intended for multiprocessors. The linearization technique in Equation (5) for the binary variables $x_{i,j}$ representing defense selection allows QUASAR$^{\text{RT}}$ to be applied to any ILP-tractable schedulability analysis. For example, the utilization-based conditions for global EDF [37, Theorem 5] $\left(\sum_{\tau_i} U_i \leq m - (m-1) \cdot \max_{\tau_i}\{U_i\}\right)$ and global RM [17, Theorem 5] $\left(\sum_{\tau_i} U_i \leq \frac{m}{2}\left(1 - \max_{\tau_i}\{U_i\}\right) + \max_{\tau_i \in \Gamma}\{U_i\}\right)$ can be represented as linear constraints on task execution times and defense selection. The MILP in [73] for partitioned EDF with variable task execution times is directly transferable, and can be extended easily to partitioned RM.

However, QUASAR$^{\text{RT}}$ is not easily extended to scheduling policies for which analysis might not be ILP-tractable, *e.g.,* to EDF with constrained deadlines, which is known to be co-NP-complete [35] even under bounded utilization. Exact and approximate approaches to optimal defense selection under such scheduling policies remain directions for future work.



(a) EDF, 10 tasks **(b)** FP, 10 tasks

Figure 8: Mean weighted defense coverage achieved by QUASAR$^{\text{RT}}$.

D. Evaluation

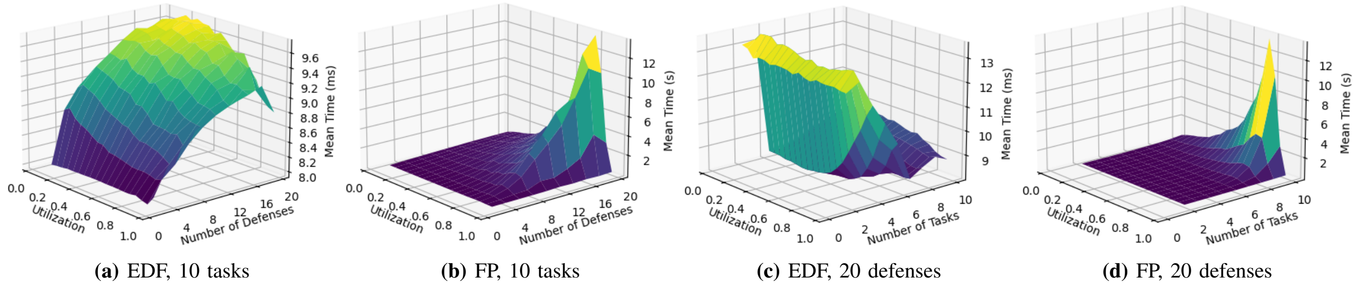
Here we evaluate QUASAR$^{\text{RT}}$ with preemptively scheduled synthetic tasks. Our case study in Section IV-E considers non-preemptive FP scheduling with optimal priority assignment.

Implementation. The QUASAR$^{\text{RT}}$ framework is written in C++ and provides a small library of functions allowing a user to specify or randomly generate sets of tasks and defenses parameterized according to Table II. It implements wrappers around the Gurobi Optimizer [39] to construct and solve the MILPs corresponding to implicit-deadline preemptive EDF and preemptive and non-preemptive FP scheduling.

Experimental Setup. To evaluate QUASAR$^{\text{RT}}$, we generate sets Γ of tasks τ_i of size n from 2–10, and with total base utilization demands (*i.e.,* with no defenses applied) $U_{\text{SUM}} = \sum_{\tau_i} \frac{C_i}{T_i}$ in the range 0.05–0.95 in steps of 0.05. We additionally generate sets Ψ of defenses d_j of size m from 2–20. Using the observations from our measurements, we split the defenses evenly between two classes. *Light* defenses increase task execution times by up to 1.2× according to the observed maximum WCET inflation measured for CFI, SafeStack, and ASLR. *Heavy* defenses increase task execution times by up to 100×, reflecting the large inflation measured for DFI (see Figure 4c). Since the selection of a heavy defense can only be justified if it enables a significant increase in defense coverage over light defenses, light defenses are randomly assigned a defense coverage $s_j \in \mathbb{N}^+$ uniformly in 1–100, while heavy defense coverage is selected from 101–1000. We note that we use an integer for defense coverage (the *number* of attack paths blocked) rather than a rational number (the *proportion* of paths) for numerical stability and efficiency in the solver.

For each combination of values (n, m, U_{SUM}) considered, we generate 1000 unique sets of tasks and defenses. Total utilization is split among tasks in a non-biased random fashion using the UUniSort technique [18]. Task periods are sampled uniformly from the range 10–100, corresponding to the "uni-moderate" distribution used in [1, 20]; base execution times are then derived as $C_{i,0} = U_i \cdot T_i$. Each task is also randomly assigned a security criticality value w_i uniformly from 1–10. For each task/defense pair (τ_i, d_j), the WCET $c_{i,j}$ of task τ_i with defense d_j applied is determined by uniformly selecting a value y in $[1, z]$, where $z = 1.2$ for light defenses and $z = 100$ for heavy defenses, then taken as $c_{i,j} = y \cdot c_{i,0}$.

We compile QUASAR$^{\text{RT}}$ using gcc version 11.5.0 and Gurobi version 10.0.3. We measure execution times using a single thread on a server with an AMD EPYC 9754 CPU



(a) EDF, 10 tasks **(b)** FP, 10 tasks **(c)** EDF, 20 defenses **(d)** FP, 20 defenses

Figure 9: Mean execution times of QUASAR$^{\text{RT}}$ optimization.

and 128GB of RAM running Linux 5.14.0 with Simultaneous Multithreading and CPU throttling disabled.

Results. Figure 8 shows the average defense coverage achieved by QUASAR$^{\text{RT}}$ across different system configurations, with unschedulable task sets treated as having coverage 0. As expected, a smaller baseline utilization provides more flexibility to apply security mechanisms, yielding higher coverage. As utilization increases, coverage falls off more quickly for FP scheduling; this makes sense since EDF generally admits higher total utilization. We also observe that coverage increases with the number of available defenses m; this suggests that broader options provide more flexibility to a system designer, providing them more space in which to select an optimal yet schedulable combination of defenses.

Figure 9 plots the mean execution times incurred by QUASAR$^{\text{RT}}$ to construct and solve the optimization problem for each task set. EDF is solved much more quickly ($<$14 ms on average) than FP scheduling; this makes sense since EDF schedulability is encoded as a simple utilization bound instead of an integer-based representation of RTA. Nonetheless, FP optimization is reasonably efficient, completing in under 20 seconds on average for 10 tasks and 20 defenses, and much faster for smaller values of m and n.

E. Case Study on ArduPilot

Experimental Setup. To evaluate QUASAR$^{\text{RT}}$'s effectiveness in a real-world CPS application, we conducted a case study on ArduPilot in HIL simulation. We sent *MAVlink* messages from the simulation machine to assign missions and issue commands to the drone, including arm throttle (enable the motor), take off (automatic climb to a target altitude), and switching to mode auto (autonomous mission mode). We then recorded the control behavior from mission start until the drone reached its first waypoint. We collected traces under four configurations: normal execution and an attack scenario evaluated with no defenses, with all defense mechanisms applied, and with the optimal defense assignment computed by QUASAR$^{\text{RT}}$.

Attack Emulation. We launched a memory-corruption attack on ArduPilot by targeting a vulnerability in its user interface components based on the real-world CVE-2022-28711 [63]. It is initiated at a random time during flight to avoid synchronization artifacts. The attack injects a payload message to change actuator control parameters, altering the throttle channel. After ArduPilot enters mode auto and starts the mission, it no longer needs additional user input. Consequently, the control tasks continue to run for the duration of the mission even if the user interface program is aborted by CFI or DFI upon attack detection. In this case, the mission completes successfully and the attack is mitigated.

Execution Times and Defenses. We measured the execution times of ArduPilot's 38 tasks under the baseline; with all combinations of ASLR, SafeStack, and CFI; and with DFI alone. As described in Section III-A, we used nonparametric bootstrapping over the observed distributions to upper-bound the execution time for each task/defense combination. Since many tasks tolerate occasional deadline misses in practice, we target a $\leq$1% exceedance probability, taking the upper bound of the 95% confidence interval as the characterized WCET.

Using QUASAR, we quantify the coverage of each considered defense. SafeStack provides the lowest coverage (~22%) while DFI provides the highest (~94.4%). Even the combination of the other three security mechanisms only achieves a defense coverage of ~85.6%, which is why we did not consider combining them with DFI.

Security Criticality. In CPS, tasks differ substantially in function, authority, and exposure. While security and safety criticality can be linked, ultimately, the quantification of security criticality is left to the system designer. In general, defining security criticality is a challenging, system-specific problem; the optimization framework is intentionally flexible, allowing different criticality policies to be encoded as needed.

As a proof of concept for ArduPilot, we present a metric that uses static analysis to estimate a task's attack surface. Specifically, we count the number of residual attack capabilities – *i.e.,* the fine-grained capabilities in QUASAR's memory-corruption ACG that are not inherently eliminated in the baseline program – that are present in each task. We restrict our search to the 12 capabilities identified as high impact in [70]. This includes the presence of vtables, stack canary, ROP gadgets, non-executable memory, position-independent executable, etc. The residual count is then normalized by the relative number of indirect calls and return addresses present in the task's binary; the result is taken as the task's security criticality.

Results. Figure 10 compares the control deviations due to each configuration over the flight duration. Under normal conditions, control deviations average only 1.8 *centimeters*. When the attack is launched, the average deviation increases to 21.37 *meters*. In response to this significant deviation, the position estimator task triggers the fail-safe, and the copter initiates an emergency landing around the 1600$^{\text{th}}$ control loop.

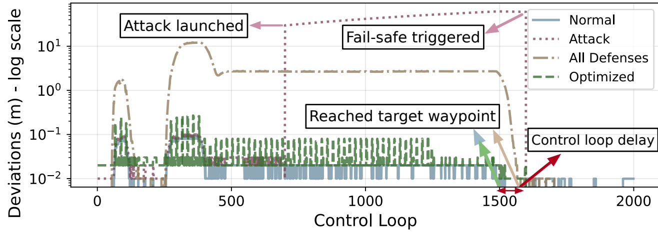


Figure 10: ArduPilot control behaviors.

With all defenses applied, the user interface components abort as soon as the attack is detected. As a result, the crafted payload is not passed to the downstream control tasks, and the copter is able to continue the mission without its control parameters being altered. However, the substantial overhead of this full-defense configuration causes several jobs to miss their deadlines. For instance, the response time of the main control loop fast_loop exceeds its deadline by ∼1.89× on average. The resulting control deviations remain high, with an average of 2.63 m, causing the copter to reach its desired waypoint approximately 100 control loops later than under normal conditions (as indicated by the red arrow in Figure 10).

Using the WCET, defense coverage, and security criticality parameters that we obtained, we ran QUASAR^{RT} on the 36 tasks dispatched in non-preemptive, FP fashion by ArduPilot's single-threaded executor. QUASAR^{RT} jointly assigns defenses and priorities to each task, selecting the combination of ASLR, SafeStack, and CFI for 12 tasks and DFI for the remaining 26. With this optimized defensive configuration, average control deviation is reduced to 2.3 cm, the upper bound on execution time expected to hold for 99% of runs of the main control loop is about 0.79 ms compared to its 2.5 ms period (*i.e.,* <⅓ utilization), and the attack is still mitigated by aborting the vulnerable user interface components. In this scenario, the copter reaches its desired waypoint at the correct time.

V. Related Work

This work primarily focuses on evaluating the applicability of runtime defenses – originally designed for general-purpose computer systems – to real-time systems, and on automatically synthesizing effective combinations of these defense strategies. Both directions have received limited attention in prior work. Accordingly, the remainder of this section reviews existing research on security in embedded and real-time systems.

Software Security for Embedded Systems: Many recent works address various security aspects in embedded systems, including CFI [11, 34, 85, 90] and DFI [84] enforcement, attestation [5, 6, 74, 82], hot-patching [42, 64], memory isolation [27, 48], and system availability [16, 54, 55, 81]. These works aim to address either overhead issues [5, 11, 34, 74, 84, 90] or the lack of hardware features [27, 48] in embedded systems. However, none of them explicitly consider the real-time implications. In contrast, this work distinguishes itself by addressing security and schedulability in real-time systems by quantifying WCET impacts.

Real-Time Security: Most related works in this area focus on information leakage through scheduling side channels [25, 61, 67], and propose corresponding mitigations [49, 80, 89]. However, their primary concern is confidentiality, which is orthogonal to software integrity, the focus of this work. Other works have also leveraged real-time properties to enhance memory security defenses [21, 36, 41, 83, 84]. These approaches take advantage of the slack between actual execution time and the WCET to adaptively generate defense strategies within WCET constraints. However, they typically target a single attack vector or focus on a specific security property. In contrast, this work investigates multiple defense mechanisms across different security properties and enables their strategic combination to achieve more effective and optimized protection.

Security Optimization: This work matures that of [45, 52]. To our knowledge, only two other prior works provide MILP-based optimization frameworks to maximize security under schedulability constraints. In [33], Di Leonardi et al. characterize WCETs in Mälardalen benchmark programs [40] using static analysis and quantifying security by enumerating vulnerabilities in CVE entries. Though they support fixed-priority, preemptive scheduling, their constraints are sufficient, not exact. In [83], Wang et al. encode exact schedulability constraints for fixed-priority, preemptive scheduling, but their security metric is specific to context-sensitive pointer integrity, limiting applicability to a single type of defense. In contrast, we propose a measurement- and profiling-based methodology toward quantifying overheads based on observed execution time distributions in dynamic, real-world CPS platforms where static WCET analysis remains impractical. The QUASAR^{RT} framework encodes exact schedulability constraints while retaining ILP tractability and supports non-preemptive scheduling with optimal priority ordering. Furthermore, it integrates QUASAR for automated security quantification for feasible combinations of defenses according to coverage in an ACG.

VI. Conclusions and Future Work

This paper empirically characterizes the real-time impact of diverse memory safety mechanisms and studies their effect on worst-case timing behavior. It distills design insights for selecting appropriate defenses and presents an optimization framework, QUASAR^{RT}, that automatically synthesizes combined strategies to maximize security while meeting real-time constraints. It uses QUASAR [70] to quantify security of each valid combination of defenses according to its coverage of an Attack Capability Graph (ACG). We evaluated QUASAR^{RT} with a case study in ArduPilot, demonstrating its feasibility in choosing defense assignments that mitigate attacks while preserving control performance.

As future work, we will extend QUASAR^{RT} to multiprocessor and other scheduling policies with ILP-tractable analysis. We will also consider fine-grained security, e.g., at the basic block level [33]. Our work focuses on defenses that enforce global invariants, which simplifies security analysis since partial instrumentation can leave exploitable gaps. However, as future work we will explore the tradeoffs involved with placement of high-overhead defenses along average-case (but not worst-case) program execution paths.

Acknowledgment

We thank the shepherd and reviewers for their valuable feedback. This work was partially supported by the NASA (80NSSC21K1741, 80NSSC24M0070), NSF (CNS-2154930, CNS-2238635), ARO (W911NF-24-1-0155), ONR (N000142412663), and a WashU OVCR seed grant.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited. This material is based upon work supported by the Under Secretary of War for Research and Engineering under Air Force Contract No. FA8702-15-D-0001 or FA8702-25-D-B002. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of War for Research and Engineering.

© 2026 Massachusetts Institute of Technology. Delivered to the U.S. Government with Unlimited Rights, as defined in DFARS Part 252.227-7013 or 7014 (Feb 2014). Notwithstanding any copyright notice, U.S. Government rights in this work are defined by DFARS 252.227-7013 or DFARS 252.227-7014 as detailed above. Use of this work other than as specifically authorized by the U.S. Government may violate any copyrights that exist in this work.

References

[1] SchedCAT: Schedulability test collection and toolkit. https://github.com/brandenburg/schedcat, 2020.

[2] AFL. https://github.com/google/AFL, 2021.

[3] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security*, CCS, 2005.

[4] J. Abella, C. Hernandez, E. Quiñones, F. J. Cazorla, P. R. Conmy, M. Azkarate-askasua, J. Perez, E. Mezzetti, and T. Vardanega. WCET analysis methods: Pitfalls and challenges on their trustworthiness. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, 2015.

[5] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik. C-FLAT: control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 743–754, 2016.

[6] T. Abera, R. Bahmani, F. Brasser, A. Ibrahim, A.-R. Sadeghi, and M. Schunter. DIAT: Data integrity attestation for resilient collaboration of autonomous systems. In *NDSS*, 2019.

[7] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. I. Davis. A comprehensive survey of industry practice in real-time systems. *Real-Time Systems*, 58(3):358–398, 2022.

[8] A. Al Arafat, S. Vaidhun, B. C. Ward, and Z. Guo. A secure resilient real-time recovery model, scheduler, and analysis. In *Proc. 9th Workshop on Mixed Criticality Systems (WMC), RTSS*, 2022.

[9] A. Al Arafat, K. Wilson, S. Vaidhun Bhaskar, B. Ward, and Z. Guo. Resilient scheduling of real-time cyber-physical systems against memory-corruptions. In *Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2025.

[10] W. Alexander. Barnaby Jack could hack your pacemaker and make your heart explode, June 2013.

[11] N. S. Almakhdhub, A. A. Clements, S. Bagchi, and M. Payer. μRAI: Securing embedded systems with return address integrity. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26*, 2020.

[12] ArduPilot. https://ardupilot.org/. Accessed: 2025-05-11.

[13] Autoware foundation. https://autowarefoundation.github.io/autoware-documentation/main/. Accessed: 2025-05-11.

[14] Autoware foundation, reference hw. https://autowarefoundation.github.io/autoware-documentation/main/reference-hw/ad-computers/. Accessed: 2025-09-11.

[15] S. Baruah and P. Ekberg. An ILP representation of response time analysis. Short note available from https://research.engineering.wustl.edu/~baruah/Submitted/2021-ILP-RTA.pdf, 2021.

[16] S. Baruah, P. Ekberg, M. Hosseinzadeh, A. Li, B. Ward, and N. Zhang. Who's afraid of butterflies? a close examination of the butterfly attack. In *2023 IEEE Real-Time Systems Symposium (RTSS)*, pages 53–63. IEEE, 2023.

[17] M. Bertogna, M. Cirinei, and G. Lipari. New schedulability tests for real-time task sets scheduled by deadline monotonic on multiprocessors. In J. H. Anderson, G. Prencipe, and R. Wattenhofer, editors, *Principles of Distributed Systems, 9th International Conference, OPODIS 2005, Pisa, Italy, December 12-14, 2005, Revised Selected Papers*, volume 3974 of *Lecture Notes in Computer Science*, pages 306–321. Springer, Springer, 2005.

[18] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Syst.*, 30(1–2):129–154, May 2005.

[19] S. Bozhko, F. Marković, G. von der Brüggen, and B. B. Brandenburg. What really is pWCET? a rigorous axiomatic proposal. In *2023 IEEE Real-Time Systems Symposium (RTSS)*, pages 13–26. IEEE, 2023.

[20] B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, University of North Carolina, Chapel Hill, NC, 2011.

[21] N. Burow, R. Burrow, R. Khazan, H. Shrobe, and B. C. Ward. Moving target defense considerations in real-time safety-and mission-critical systems. In *Proceedings of the 7th ACM Workshop on Moving Target Defense*, pages 81–89, 2020.

[22] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. Control-flow integrity: Precision, security, and performance. *ACM Comput. Surv.*, 50(1), Apr. 2017.

[23] N. Burow, X. Zhang, and M. Payer. SoK: Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 985–999. IEEE, 2019.

[24] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 147–160, 2006.

[25] C.-Y. Chen, S. Mohan, R. Pellizzoni, R. B. Bobba, and N. Kiyavash. A novel side-channel in real-time schedulers. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 90–102, 2019.

[26] J.-J. Chen, G. von der Brüggen, W.-H. Huang, and R. I. Davis. On the pitfalls of resource augmentation factors and utilization bounds in real-time scheduling. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, pages 9–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017.

[27] 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)*, pages 65–82, 2018.

[28] 2024 CWE top 25 most dangerous software weaknesses. https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html. Accessed: 2025-09-11.

[29] CVEdetails. https://www.cvedetails.com/vulnerabilities-by-types.php. Accessed: 2025-05-11.

[30] A. Darwiche. Decomposable negation normal form. *J. ACM*, 48(4):608–647, July 2001.

[31] A. Darwiche. A compiler for deterministic, decomposable negation normal form. In *AAAI/IAAI*, pages 627–634, 2002.

[32] Dexcom G7 Continuous Glucose Monitoring (CGM) System. https://www.dexcom.com/g7-overview. Accessed: 2025-05-11.

[33] S. Di Leonardi, F. Aromolo, P. Fara, G. Serra, D. Casini, A. Biondi, and G. Buttazzo. Maximizing the security level of real-time software while preserving temporal constraints. *IEEE Access*, 11:35591–35607, 2023.

[34] Y. Du, Z. Shen, K. Dharsee, J. Zhou, R. J. Walls, and J. Criswell. Holistic control-flow protection on real-time embedded systems with Kage. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2281–2298, 2022.

[35] P. Ekberg and W. Yi. Uniprocessor feasibility of sporadic tasks remains conp-complete under bounded utilization. In *2015 IEEE Real-Time Systems Symposium*, pages 87–95, 2015.

[36] J. Fellmuth, T. Göthel, and S. Glesner. Instruction caches in static WCET analysis of artificially diversified software. In *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[37] J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems*, 25(2):187–205, Sep 2003.

[38] Z. Guo, K. Yang, F. Yao, and A. Awad. Inter-task cache interference aware partitioned real-time scheduling. In *Proceedings of the 35th annual ACM symposium on applied computing*, pages 218–226, 2020.

[39] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2024.

[40] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. In B. Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *Open Access Series in Informatics (OASIcs)*, pages 136–146, Dagstuhl, Germany, 2010. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[41] M. Hasan, S. Mohan, R. B. Bobba, and R. Pellizzoni. Exploring opportunistic execution for integrating security into legacy hard real-time systems. In *2016 IEEE Real-Time Systems Symposium*, pages 123–134, 2016.

[42] Y. He, Z. Zou, K. Sun, Z. Liu, K. Xu, Q. Wang, C. Shen, Z. Wang, and Q. Li. RapidPatch: Firmware hotpatching for real-time embedded devices. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2225–2242, 2022.

[43] C. Herley and P. C. Van Oorschot. SoK: Science, security and the elusive goal of security as a scientific pursuit. In *2017 IEEE symposium on security and privacy (SP)*, pages 99–120. IEEE, 2017.

[44] A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz. Microgadgets: size does matter in turing-complete return-oriented programming. In *Proceedings of the 6th USENIX conference on Offensive Technologies*, pages 7–7. USENIX Association, 2012.

[45] A. Horne. *Optimizing Memory-Corruption Security Defenses for Real-Time Systems*. PhD thesis, Massachusetts Institute of Technology, 2022.

[46] Jackal UGV. https://clearpathrobotics.com/jackal-small-unmanned-ground-vehicle/. Accessed: 2025-05-11.

[47] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, 01 1986.

[48] A. Khan, D. Xu, and D. J. Tian. Ec: Embedded systems compartmentalization via intra-kernel isolation. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2990–3007. IEEE, 2023.

[49] K. Krüger, M. Volp, and G. Fohler. Vulnerability analysis and mitigation of directed timing inference based attacks on time-triggered systems. In *Euromicro Conference on Real-Time Systems*, 2018.

[50] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2014.

[51] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated software diversity. In *35th IEEE Symposium on Security and Privacy*, S&P, 2014.

[52] C. Lemieux-Mack, K. Leach, N. Zhang, S. Baruah, and

B. C. Ward. Optimizing runtime security in real-time embedded systems. In *Proc. of Workshop on Optimization for Embedded and Real-time Systems (OPERA)*, 2024.

[53] A. Li, J. Wang, S. Baruah, B. Sinopoli, and N. Zhang. An empirical study of performance interference: Timing violation patterns and impacts. In *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 320–333. IEEE, 2024.

[54] A. Li, J. Wang, and N. Zhang. Software availability protection in cyber-physical systems. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 1807–1825, 2025.

[55] A. Li and N. Zhang. Data-flow availability: Achieving timing assurance in autonomous systems. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 445–463, 2024.

[56] LLVM released version 13.0.1. https://github.com/llvm/llvm-project/tree/release/13.x. Accessed: 2025-05-11.

[57] M. Lv, N. Guan, Y. Zhang, Q. Deng, G. Yu, and J. Zhang. A survey of WCET analysis of real-time operating systems. In *2009 International Conference on Embedded Software and Systems*, pages 65–72, 2009.

[58] F. Marković, G. von der Brüggen, M. Günzel, J.-J. Chen, and B. B. Brandenburg. A distribution-agnostic and correlation-aware analysis of periodic tasks. In *2024 IEEE Real-Time Systems Symposium (RTSS)*, pages 215–228. IEEE, 2024.

[59] Microsoft. A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003. Online, September 2006.

[60] C. Miller and C. Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015(S 91):1–91, 2015.

[61] S. Mohan, M. K. Yoon, R. Pellizzoni, and R. Bobba. Real-time systems security through scheduler constraints. In *26th Euromicro Conference on Real-Time Systems*, pages 129–140, 2014.

[62] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–258, 2009.

[63] NVD - CVE-2022-28711. https://nvd.nist.gov/vuln/detail/CVE-2022-28711. Accessed: 2025-09-11.

[64] C. Niesler, S. Surminski, and L. Davi. HERA: Hotpatching of embedded real-time applications. In *NDSS*, 2021.

[65] OpenBSD. Openbsd 3.3, 2003.

[66] PaX. PaX address space layout randomization, 2003.

[67] R. Pellizzoni, N. Paryab, M.-K. Yoon, S. Bak, S. Mohan, and R. B. Bobba. A generalized model for preventing information leakage in hard real-time systems. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 271–282, 2015.

[68] ROBOTIS OP3 Humanoid. https://emanual.robotis.com/docs/en/platform/op3/introduction/. Accessed: 2025-05-11.

[69] A. Schelfhout, A. Jacobs, J. Vinck, and S. Volckaert. Diagnosing and neutralizing address-sensitive behavior in multi-variant execution systems. In *Proceedings of the 18th European Workshop on Systems Security*, pages 1–10, 2025.

[70] R. Skowyra, S. R. Gomez, D. Bigelow, J. Landry, and H. Okhravi. QUASAR: Quantitative attack space analysis and reasoning. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 68–78, 2017.

[71] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz. SoK: sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1275–1295. IEEE, 2019.

[72] S. Song, J. Zhang, and M. Yan. Oreo: Protecting ASLR against microarchitectural attacks. In *32nd Annual Network and Distributed System Security Symposium, NDSS*, pages 24–28, 2025.

[73] M. Sudvarg, D. Wang, J. Buhler, and C. Gill. Subtask-level elastic scheduling. In *2024 IEEE Real-Time Systems Symposium (RTSS)*, pages 388–401. IEEE, 2024.

[74] Z. Sun, B. Feng, L. Lu, and S. Jha. OAT: Attesting operation integrity of embedded devices. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1433–1449. IEEE, 2020.

[75] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In *Proc. of IEEE Symposium on Security and Privacy*, 2013.

[76] H. Teper, O. Bell, M. Günzel, C. Gill, and J.-J. Chen. Reconciling ROS 2 with classical real-time scheduling of periodic tasks. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2025)*. IEEE, 2025.

[77] Turtlebot3. https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/. Accessed: 2025-05-11.

[78] Unitree Quadruped Robot Go1. https://www.unitree.com/go1. Accessed: 2025-05-11.

[79] J. Vinck, A. Jacobs, A. Voulimeneas, and S. Volckaert. Divide and conquer: Introducing partial multi-variant execution. In *2025 IEEE 10th European Symposium on Security and Privacy (EuroS&P)*, pages 1049–1066. IEEE, 2025.

[80] N. Vreman, R. Pates, K. Krüger, G. Fohler, and M. Maggio. Minimizing side-channel attack vulnerability via schedule randomization. In *2019 IEEE 58th Conference on Decision and Control*, pages 2928–2933, 2019.

[81] J. Wang, A. Li, H. Li, C. Lu, and N. Zhang. Rt-tee: Real-time system availability for cyber-physical systems using arm trustzone. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 352–369. IEEE, 2022.

[82] J. Wang, Y. Wang, A. Li, Y. Xiao, R. Zhang, W. Lou, Y. T. Hou, and N. Zhang. {ARI}: Attestation of real-time mission execution integrity. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 2761–2778, 2023.

[83] Y. Wang, C. Lemieux-Mack, T. Chantem, S. Baruah, N. Zhang, and B. C. Ward. Partial context-sensitive pointer integrity for real-time embedded systems. In *2024 IEEE Real-Time Systems Symposium (RTSS)*, pages 415–426. IEEE, 2024.

[84] Y. Wang, A. Li, J. Wang, S. Baruah, and N. Zhang. Opportunistic data flow integrity for real-time cyber-physical systems using worst case execution time reservation. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 6615–6632, 2024.

[85] Y. Wang, C. L. Mack, X. Tan, N. Zhang, Z. Zhao, S. Baruah, and B. C. Ward. Insectacide: Debugger-based holistic asynchronous cfi for embedded system. In *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 360–372. IEEE, 2024.

[86] B. C. Ward, A. Thekkilakattil, and J. H. Anderson. Optimizing preemption-overhead accounting in multiprocessor real-time systems. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, pages 235–243, 2014.

[87] Waymo One: Autonomous rides in San Francisco. https://waymo.com/waymo-one-san-francisco/. Accessed: 2025-05-11.

[88] G. Yao, G. Buttazzo, and M. Bertogna. Feasibility analysis under fixed priority scheduling with fixed preemption points. In *2010 IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 71–80, 2010.

[89] 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 *2016 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 1–12, 2016.

[90] J. Zhou, Y. Du, Z. Shen, L. Ma, J. Criswell, and R. J. Walls. Silhouette: Efficient protected shadow stacks for embedded systems. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1219–1236, 2020.