An Empirical Study of Performance Interference: Timing Violation Patterns and Impacts

Ao Li, Jinwen Wang, Sanjoy Baruah, Bruno Sinopoli, Ning Zhang
Washington University in St. Louis
{ao, jinwen.wang, baruah, bsinopoli, zhang.ning}@wustl.edu

***Abstract*—Multi-core platforms are becoming increasingly prevalent in cyber-physical systems such as automobiles and robots. However, contention for shared resources makes it challenging to guarantee timing predictability. Existing studies have primarily focused on characterizing the extent to which such interference can induce delays (usually from an adversarial perspective). Unfortunately, less is understood on the physical impacts of these timing delays in different cyber-physical platforms. In this paper, we fill this gap by providing an empirical examination of the end-to-end effects of performance interference on real-world applications. We analyze the root causes of harmful interference and summarize potential implementation pitfalls.**

To automate this process, we introduce TimeTrap, a tool that analyzes performance interference in autonomous systems through the lens of control outcome. To understand the extent to which timing interference may cause control deviations, TimeTrap has to strategically leverage different magnitudes of resource contention to trigger targeted deadline miss patterns. Through this exercise, we found that a naïve approach that maximizes task latency via performance interference may fail to trigger worst-case outcomes (i.e. physical damages) due to built-in fail-safe mechanisms. As a result, delays have to be induced in a stealthy manner to avoid triggering fail-safes. To achieve this, TimeTrap first employs a system that actively injects fine-grained delays into the target software, adjusting the duration based on measured feedback. Second, TimeTrap leverages predictability in CPS execution patterns and resource usage to automatically tune its aggressor workloads, matching these patterns to achieve targeted interference and execution delays in a victim. We evaluate TimeTrap on two physical-world platforms and six platforms in a hardware-in-the-loop simulation environment, including robotic arms, UGVs, UAVs, self-driving cars, and humanoid robots. These studies demonstrate that an interference-based attack surface exists in different stages of the CPS pipeline, from perception to planning and control.

I. Introduction

Multi-core architectures are highly effective at providing increased computational resources under constraints on size, weight, and power (SWaP) in embedded platforms. However, when deployed in real-time cyber-physical systems (CPS) where timing predictability is essential, performance interference from contention for shared resources among concurrent tasks is increasingly a concern. Such interference may lead to delays in data propagation along the processing pipeline in modern CPS platforms, consequently degrading performance.

Existing Literature. Understanding the impacts of performance interference in CPS requires a comprehensive analysis of both the sources and characteristics of interference, as well as its resulting impact on control outcomes. This demand is also highlighted by a recent industrial challenge presented by Arm [1]. Existing studies follow three distinct lines of research: *(i)* The first exclusively examines the extent of worst-case performance interference [2]–[12] without considering the subsequent implications on system and software behavior. *(ii)* The second primarily focuses on analyzing the propagation of delays throughout the execution pipeline [13]–[19]. Such works approach the problem from the scheduling perspective, yet often do not consider the impact on control outcomes. *(iii)* The third abstractly models the control degradation resulting from various deadline miss patterns in an individual task [20]–[23]. However, these models do not consider the broader view of cascading effects among multiple tasks in the system. To analyze the impact of performance interference through a holistic view that interconnects software properties, real-time properties, and control properties [24], our work empirically studies the manifestation of performance interference in different CPS platforms to illuminate the implications for physical safety risks in the real world.

Systematizing Timing Issues in Real-World CPS Software. To develop an appropriate abstraction over timing issues, we analyzed 241 real-world timing-related bugs from eight prominent CPS applications. We discovered that these bugs often arise from the inadequate specification or enforcement of timing semantics. We also found that not all timing bugs result in adverse control outcomes in the physical world. In this paper, we use *timing issues* to refer to the timing bugs that produce adverse physical outcomes. Dissecting these real-world timing issues has provided us with new insights to formulate the problem of performance interference impact as *temporal displacements* in key state variables of the CPS process (more details about the bug systematization can be found in Section II).

TimeTrap — Technical Challenges and Solutions. Informed by our observations, we introduce ***TimeTrap***, a tool that aims to characterize the worst-case end-to-end impacts of performance interference by adversarially inducing interference with the goal of maximizing control deviation (i.e. difference in physical outcome). However, naïvely exhausting system resources is often ineffective due to built-in safety checks, as we discuss in Section II. Consequently, it is imperative to discern which task(s) should be delayed in order to effect significant control deviation. Determining how much delay the attacker should cause is also challenging, since too little

delay is insufficient to breach safety standards, yet too much may trigger fail-safe mechanisms. To address this, TimeTrap borrows methodology from software fault injection [25] by actively injecting delays into the target system to discover exploitable execution timing patterns. To further automate this process, we have also developed program analysis and instrumentation tools to automatically determine the appropriate delay injection location(s) and magnitude.

Even with such patterns identified, TimeTrap needs to further verify that the intended interference pattern is achievable. There are numerous ways to cause performance interference ([4], [8], [26]–[32]), each with unique triggering mechanisms and interference characterization. Furthermore, the effectiveness of the interference also depends heavily on the resource utilization pattern of the victim thread. TimeTrap takes advantage of the fact that different task execution phases often exhibit distinct sensitivity to each performance interference vector. It parameterizes the attack strategy and uses it to control interference intensity. By strategically generating aggressor workloads targeted at specific vectors, TimeTrap induces the intended delay during the target execution phases of the specified task(s).

Experiments and Results. Using TimeTrap, we were able to attack different CPS pipeline components, including perception, localization, planning, and control. We evaluated TimeTrap on eight autonomous systems (including autonomous vehicles, drones, humanoid robots) in a hardware-in-the-loop (HiTL) simulation environment. We also tested it on two real physical systems: the OpenManipulator robotic arm [33] and the Turtlebot3 indoor robot [34]. We show that naïve Denial-of-Service (DoS) attacks with high CPU bandwidth and priority do not necessarily cause harmful physical effects, whereas TimeTrap can create workloads that trigger specific timing patterns to achieve these outcomes.

In summary, this work[1] makes the following contributions:

- It provides a systematization of timing bugs from multiple real-world autonomous systems, covering root causes, manifestations, fix strategies, and other characteristics.
- It empirically examines several real-world CPS applications to understand how performance interference can trigger bugs and assesses the impacts of these timing issues on control performance. It dissects the root cause of control degradation and formulates it as a problem of *temporal displacement* in a subset of critical control data consumed by safety-critical computations.
- It introduces TimeTrap, an automatic tool framework for discovering timing issues that cause significant control degradation and for constructing adversarial aggressor workloads to trigger them. TimeTrap is evaluated on eight autonomous systems: two in the real world and six in HiTL simulation. TimeTrap generates aggressor workloads that cause significant control deviations in various CPS modules, including perception, localization, planning, and control.

[1]Materials can be accessed from the GitHub repository at https://github.com/WUSTL-CSPL/TimeTrap and the project's website at https://timetrap-rtas.github.io/.

II. Systematization of Real-World Issues and Temporal Displacement

In this section, we first utilize ORB-SLAM as an example to dissect real-world timing issues, and then we formulate these timing issues to *temporal displacement*, followed with a more comprehensive analysis of timing issues that developers have encountered in real-world CPS software.

A. Dissecting the Timing Issues – Temporal Displacement

To introduce the concept of temporal displacement, we employ ORB-SLAM [35], [36] as an illustrative example. ORB-SLAM is a localization pipeline that sees broad use in self-driving applications [37]. It employs a feature extractor to process an incoming video stream and output landmark descriptions. It matches them with previous descriptions saved in its map database to estimate vehicle position. Instead of a computationally intensive exhaustive search, ORB-SLAM uses an IMU-based motion model to narrow the search scope. At this stage, both IMU sensor data and camera image frames are combined. ORB-SLAM also periodically updates the map to account for changes in its surroundings. Figure 1 shows ORB-SLAM's three primary data trackers, each handling data from a different source.

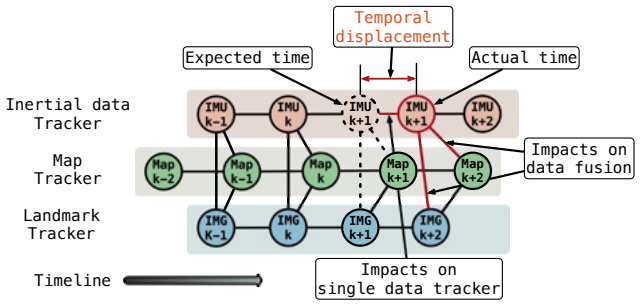


Fig. 1: Dissection of temporal displacement.

Figure 1 also illustrates an example of data delay. Inertial data ($\texttt{IMU}_{k+1}$), expected to be available at the instant indicated by the dashed circle, is delayed due to an adversarial task. This delay impacts both its data tracker and the data fusion process. Within the `Inertial Data Tracker`, delayed inertial data amplifies the uncertainty of motion prediction. More critically, this delay can cause the `Landmark Tracker` to utilize temporally misaligned inertial data. This misalignment subsequently results in a failure to match the current landmark descriptions against the map database, causing the algorithm to lose track of its position. This data misuse can be conceptually considered as *Temporal Displacement*, $\delta = T_{\text{actual}} - T_{\text{expected}}$. Intuitively, it is the difference between the actual and the expected timing of the update. In fact, temporal displacements always exist, but built-in system mechanisms are engineered to minimize them.

Examples of Timing Issues in Software. Although many autonomous systems already employ mechanisms to check and handle abnormal timing, this approach does not always ensure accurate timing.

```
1:  while (true) {
2:      // Browser the queue where IMU data are sorted by timestamps.
3:      imu = ImuTracker.DataQueue.front();
4:      if (imu->Timstamp < PreviousImage->TimeStamp) {
5:          ImuTracker.DataQueue.pop_front(); // Drop the out-dated IMU data.
6:       } else if (imu->Timstamp < CurrentImage->TimeStamp) {
7:           res.push_back(*imu); // Keep the data between two frames.
8:           ImuTracker.DataQueue.pop_front();
9:       } else {
10:          break;}} // Reach the end
```

(a) Timestamp checking of IMU data between two image frames

```
/* Thread A (Landmark Tracker) */
1:  elapsedTime = now() - lastUpdateTime;
2:  if (elapsedTime > 0.5) {
3:       points = CreatNewMapPoints();
4:       MapTracker.InsertMapPoints(points);
5:      MapTracker.NeedBA = true;       // Should update map now
6:  } End of if


/* Thread B (Map Tracker) */
1:  while (true) {
2:      if (this->NeedBA) {
3:          BAOptimization(); // Update the map
4:    } }; End of while
```

(b) Timing policy specified in userspace application.

Fig. 2: Simplified snippets in ORB-SLAM2.

A straightforward solution to prevent temporal displacement is to check the data age and act according to its freshness. However, this often requires sophisticated condition checks due to the complex nature of timing behavior. Many programming languages lack timing semantics, leading to incomplete expression of the programmer's intent. Figure 2(a) shows code from ORB-SLAM2 [35] that fetches IMU data between two images, checking that the timestamps fall between those of the images to avoid temporal misalignment. Given that the IMU's sampling rate, R_i, is typically greater than the camera's, R_c, there should ideally be R_i/R_c IMU samples between two image frames. However, the code fails to detect missing IMU samples. Even if the retrieved IMU data has valid timestamps, data loss can still lead to erroneous results.

As depicted in Figure 2(b), ORB-SLAM2's *Landmark Tracker* code checks if the time since the last map update exceeds 0.5 seconds. If so, it sets a flag to trigger a map update. Ideally, given sufficient resources, the `BAOptimization()` function, which runs in its own thread, is immediately invoked. However, high system overhead may prevent this function from timely execution because the CPU scheduler is unaware of the intrinsic timing requirements of the application (such as those illustrated in Figure 2(b)).

B. Temporal Displacement in Real-world Applications

To explore temporal displacement and characterize its patterns in real-world software implementations, we examined timing-related issues and pull requests in the GitHub projects of popular (by the number of forks and stars) CPS applications. We covered a wide variety of modules including perception, planning, control, etc. The 8 projects studied are Autoware.Auto [38], MoveIt [39], Google Cartographer [40], Baidu Apollo [41], ORB-SLAM2/3 [42], [43], ROS-Planning Navigation [44], and ROS2 rcl middleware [45]. We manually inspected the discussions of each issue and pull request, excluding any falsely reported issues. The threats to the validity of our study include the representativeness of the application we have chosen and our examination methodology.

Projects	*Autoware [38]*	*MoveIt [39]*	*Cartographer [40]*	*Apollo [41]*	*ORB-SLAM2/3 [42], [43]*	*Navigation [44]*	*ROS2 rcl [45]*
LoC	101k	143k	60k	571k	69k	120k	149k
Category	Full	Control*	Loca.	Full	Loca.	Control*	Mw.
Total #	1243	2740	1953	15251	1064	3352	2066
Timing #	17	27	43	55	12	22	65
Type I	3	4	10	12	8	2	5
Type II	5	11	16	23	-	6	2
Type III	2	9	10	12	6	1	37

Full=Full stack, including perception, localization, planning and control;
* Control here includes mission planning, motion planning, and controller;
Loc. = Localization; Sw. = Software; Mw. = Middleware.

TABLE I: Analysis of Timing-Related Issues Across Various GitHub Projects

Prevalence of Timing Issues. The statistics are reported in Table I. Each project is seen to suffer from dozens of timing-related issues in development. The timing issues constitute approximately 0.5% to 2% of the overall bugs. It is worth noting that the number of issues does not directly indicate a project's quality, but is heavily influenced by its popularity and the developers' activity in the target GitHub repository. By analyzing these 241 timing issues and reproducing some of them, we detail our findings in the following discussion.

Current Practice for Timing Constraints. Timing constraints are programmed diversely in practice. We classify them into two categories: *control-flow-centric* and *data-flow-centric*. Control-flow-centric timing constraints are defined from the perspective of the program's execution, such as through scheduling models (e.g., deadlines and periods) and locking protocols, which are among the most conventional methods. In the data-flow-centric paradigm, timing information is coupled with data and then utilized to determine software behaviors. A common example involves messages in ROS, which are tagged with timestamps. Developers leverage these timestamps to check the freshness or temporal alignment of data, thus adjusting software behavior accordingly. In userspace applications (except for ROS2 rcl [45]), the data-flow-centric paradigm is preferred, with 69.3% of issues being related to it, because (1) it does not require modifications to the underlying scheduling infrastructure, and (2) it better captures the dependency between tasks, helping developers to reason about the correct timing constraints within the context of complex task chains.

Root Causes. We identify three timing issue root causes:

- `Type I`. Timing constraints are missing or implicitly assumed without proper enforcement, where any variations on task execution timings can potentially introduce temporal

displacement. This accounts for 23.9% of the issues.

- `Type II`. Timing constraints are specified, but they are incomplete (34.2%); temporal displacement is permitted only for a subset of the data or within a certain threshold.
- `Type III`. Specified timing constraints are completely defined but not effectively enforced, which account for 41.8%. These are often caused by flaws in system executors (e.g., schedulers) and improper coordination among them.

As data-flow-centric timing constraints are more prevalent in software (69.3%), many `Type I` issues are largely caused by the absence of timing information (e.g., timestamps) on data. Although ROS offers a built-in timestamp assignment interface in its API, developers often fail to propagate these timestamps throughout the lengthy processing pipeline. This issue is exacerbated when multiple sensor data streams are fused within a single component.

`Type II` issues prevail due to the challenge of accurately expressing timing semantics across different scenarios. This category includes vulnerabilities like imprecise value ranges or TOCTOU ('time-of-check-to-time-of-use') problems (e.g., Apollo#14461[2]), where timestamp verification occurs prematurely. Sometimes, these issues arise when timestamps are not expressed with sufficient granularity (Cartographer#600[3]). Exploiting these issues necessitates inducing precise delays during specific execution stages.

`Type III` issues are commonly found in the ROS2 middleware [45] due to flawed implementations in schedulers. Most of these issues stem from race conditions (e.g., ROS2-rcl#2012[4]) and often involve various scheduling entities, such as timers (e.g., ROS2-rclpy#1016[5]). Additionally, timing-related utilities are also error-prone. For example, the issues caused by incompatible timing formats between different clock systems (e.g., ROS2-rcl#947[6]) are common.

Manifestation. The triggering conditions of timing issues we collected exhibit characteristics similar to conventional concurrency bugs [46], making them challenging to distinguish since both are triggered by unexpected timing. Our focus is on the issues related to algorithm-level semantics and control performance, given that the symptoms of concurrency bugs have been well-studied in prior work. Generally, timing issues do not necessarily lead to delays in control outputs because actuation commands are usually sent by a separate task that may be unaffected by the task experiencing abnormal timings. However, timing issues often lead to data misuse, predominantly arising from the use of stale data or, occasionally, premature data. This stale data can lead to desynchronization when the program consumes data from disparate sensor modalities or components. Furthermore, it can cause data jitter if updates occur at inconsistent intervals (e.g., Cartographer#242[7]). Desynchronization is common (more than 30%) due to the complexities in reasoning about multi-threaded concurrent execution. The scope of most issues involves two or three threads (both intra- and inter-process), where the delays in a subset of threads trigger unexpected timing.

Most of these issues yield false computational results, which subsequently undermine the physical stability of the control system. Some of them (< 5%) cause the software to become unresponsive, while slightly more (< 10%) even lead to software crashes. Desynchronization and jitter are more subtle and typically result in adverse control degradation. In contrast, end-to-end latency is less of a concern because programmers are typically more aware of the data's age within a sequential context and thus often adopt compensation strategies for such delays. When reproducing some timing issues, it is found that while timing issues cause erroneous intermediate results, most robotic software is robust enough to filter out such occasional faults; an adverse outcome (task output) requires consecutive abnormal timings to manifest.

Insight: Discovering timing issues requires inducing delays in only a *subset* of the threads, leading to an abnormal execution order. Such delays should be injected consecutively to propagate the erroneous results to the final control outcome.

Mitigation Strategies. Among the analyzed issues, 141 (58.5%) have been fixed in their codebase. In general, we found that timing issues can be fixed by either enforcing the correct execution timing or changing the software's handling of the abnormal timing. The former is typically achieved by adding or modifying synchronization primitives (e.g., mutexes) in the software, while it is rarely achieved by modifying the underlying enforcement infrastructure, such as schedulers and timers. However, synchronization primitives are not a silver bullet. Adding locks can introduce new problems, including increased real-time overhead and potential inversion in lock ordering (rcl#1121). Regarding the latter, most issues (over 80%) were fixed in a data-flow-centric manner. The mitigation strategy involves integrating more timing information into the data, typically by adding timestamps to data structures, and then adopting check conditions and handling mechanisms for abnormal timing cases. The most prevalent mechanisms for handling task misses are to *(i)* abort the task instances (e.g., callback in ROS) and then trigger operational-safe or fail-safe mechanisms (e.g., setting the velocity to zero); or *(ii)* use predictive models or algorithms to approximate the correct data. Such handling mechanisms introduce challenges to TimeTrap, which are discussed in Section IV.

Insight: Among the target software we investigated, besides enforcing timing constraints (e.g., deadlines or periods) as conventional scheduling problems, it is also common for developers to implement detectors for abnormal timing and case-specific handlers to avoid adverse control outcomes.

III. System Model

In this work, we target real-time systems with both *event-triggered* (e.g., data-driven) and *time-triggered* (e.g., periodic) execution models. These mechanisms are natively supported by the systems that we evaluate in Section VI. We consider

[2]https://github.com/ApolloAuto/apollo/pull/14461/files
[3]https://github.com/cartographer-project/cartographer/issues/600
[4]https://github.com/ros2/rclcpp/issues/2012
[5]https://github.com/ros2/rclpy/issues/1016
[6]https://github.com/ros2/rcl/issues/947
[7]https://github.com/cartographer-project/cartographer/issues/242

adversarial behavior under the following conditions. *(i)* The aggressor workload executes on the same computing platform as the target application. *(ii)* The aggressor is restricted to execute at a lower priority than all other tasks on the system. It lacks the ability to directly modify scheduling priorities or CPU affinities, including its own. *(iii)* The aggressor workload executes at a non-privileged level – it does not have access to the address space of any other task.

IV. TimeTrap Design

Overview. TimeTrap is a tool that aims to discover timing issues and generate aggressor workloads to confirm the possibility of triggering such timing issues. Its workflow is generalized into three steps, as shown in Figure 3.

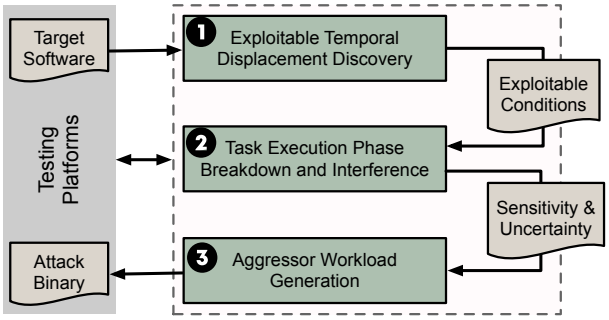


Fig. 3: The workflow of TimeTrap.

Step 1 - Exploitable Temporal Displacement Discovery. (Section IV-A) It is clear that not all temporal displacements can trigger timing bugs, and that not all timing bugs lead to severe control performance degradation. Building on top of existing literature that features discovering concurrency bugs [47], TimeTrap injects delays into program execution and then observes the control behavior and immediate variables in the testing. The delay patterns that have evident physical impacts are regarded as exploitable timing bugs.

Step 2 - Task Execution Phase Breakdown and Interference. (Section IV-B) This step involves the profiling of resource contention sensitivity. To cause the desired timing delay in specific tasks, it is essential to have an in-depth understanding of the target workloads' resource usage, as well as the potential impact of the aggressor workload. To facilitate finer control over the delay, we break down program execution into phases that display unique resource usage profiles. Based on these execution phases, individual attack primitives (contention channels) are then measured to understand their potential impact on the temporal properties of the system.

Step 3 - Aggressor Workload Generation. (Section IV-C) The attack strategy is developed based on exploitable temporal displacement and sensitivity analysis, formulating the generation as an optimization problem. The objective is to maximize expected execution timings while minimizing side effects on unrelated co-running tasks.

A. Discovering Exploitable Conditions

Challenge. To cause temporal displacement, an attacker must selectively delay certain threads with minimal impact on others. Simply delaying all tasks often leads to an end-to-end delay (input-output latency), which is typically handled by built-in mechanisms, such as aborting the task or algorithmically predicting data (refer to Section II). TimeTrap seeks to introduce specific data desynchronization or jitters, bypassing these countermeasures.

Execution timing control via delay injection. To discover such exploitable conditions, TimeTrap follows the principles of software fault injection [25], injecting delays during execution to generate diverse timing execution patterns. Concretely, it injects delays in different program locations to drive the program toward producing erroneous control results. There are two key design questions to answer: *(i)* which code locations are eligible for delay injection? and *(ii)* how much delay should be injected to trigger adverse control outcomes?

Where to inject delays? There are three alternative methods for injecting delays: (1) injecting delays at the OS level by manipulating scheduling; (2) injecting delays in the runtime library by intercepting library calls; and (3) directly injecting delays into the source code through compile-time transformation. The first two methods do not require the availability of source code. However, they also come with drawbacks. Injecting delays in scheduling loses a lot of semantic context, such as the data dependencies between different code locations. Injecting delays in the library can potentially maintain such relations, since it can provide semantic information about the sender and receiver. However, this approach is specific to the implementation of the target software and cannot scale to cases where shared data are not implemented via APIs, such as variables used for communication.

As such, TimeTrap injects delay in source code. At compile-time, it performs static analysis to identify all the code regions that access shared data with other tasks. This shared data can be intra-process memory or inter-process messages, exhibiting different granularity across various target software. For intra-process shared variable analysis, it employs points-to analysis [48]; for inter-process communication analysis, it utilizes string matching. This analysis is straightforward and it reports the code location without checking their contexts. We will refer to these locations as delay injection points. Each delay injection point is paired with other locations interacting with the shared data, with at least one point in a pair involving a write operation.

Since most timing issues often lead to erroneous results, TimeTrap conducts A/B testing to infer the impacts of delays. Specifically, it sets up the software on the target testing platform and replays the same data for two different runs, one with delay and one without. It then compares the computational results of these two executions; the results could be intermediate or final control outcomes, which can be logged through instrumentation or built-in logging mechanisms. The extent of the difference between these results indicates the impact of

delays. For each pair of injection points, it selects one to inject a delay at runtime. There is a clear trade-off between injecting a single delay and multiple delays throughout a run. Delays are injected at multiple points in one run if the intermediate results can be observed to distinguish the impact of the delay at each injection point.

How much should threads be delayed? As discussed in Section II-B, erroneous results only manifest after consecutive abnormal timings. Thus, traditional delay injection methods for discovering concurrency bugs, which are one-time shots [47], are not suitable for these scenarios. Instead, TimeTrap continuously injects the same execution timing patterns throughout one mission execution. The timing patterns are represented by three factors: the code location, whether the type is delay or jitter, and the extent of delay or jitter. An alternative representation is the weakly-hard task model [49], [50], which is unsuitable for our scenario for two reasons. First, this model only describes the number of deadline hits or misses but ignores the extent of the delay. Our preliminary study found that many applications have built-in check conditions on the data freshness. Second, the shared data is not necessarily updated at the end of the deadline, which means a deadline miss does not necessarily impact the data consumption of dependent tasks.

To obtain an effective range of delay, we progressively inject delays from large to small at the same program location. We then observe the intermediate results and the final control outcome after each change. If the final control outcome shows fail-safe operations, the extent of delays will be decreased to avoid triggering it. If the intermediate results show significant changes but the control outcome does not, this indicates that the downstream tasks are robust to such timing issues. In this case, or after exploring all extents of delays within a predefined range, the injection point is removed from the candidate list. Once an execution timing pattern is demonstrated to cause an adverse physical outcome, that pattern is recorded as exploitable and used in subsequent steps.

B. Resource-Oriented Execution Profiling

Challenge. Triggering an expected timing execution pattern through performance interference is non-trivial. Typically, multiple threads are co-running on the target platform. To induce the expected timing, it is necessary to interfere with only a subset of tasks while keeping the remaining ones untouched. Additionally, the delay should be manipulated within a specific range. TimeTrap exploits the diversity of resource usage patterns and the sensitivity of different interference channels to achieve targeted delays.

Resource-focused task execution phases. Different tasks often have diverse demands on resources. For example, motion planning tasks often solve cache-intensive optimization problems, whereas drivers that read from sensor devices rely more on I/O resources. In addition, the resource usage pattern of individual tasks also varies throughout its execution. Inspired by previous work [51], we first precisely profile a workload's resource usage over time, and then divide each task into a sequence of *execution phases* where similar resources are intensively used in each phase. Depending on the desired granularity, the phase may vary from thousands to tens of thousands of instructions. Then, we individually characterize each phase's sensitivity to individual resource contention channels. This is achieved by scheduling the target phase of the victim task on a specific core and placing interference tasks on other cores. By measuring the delays in the victim tasks' phases, we determine the effects of interference.

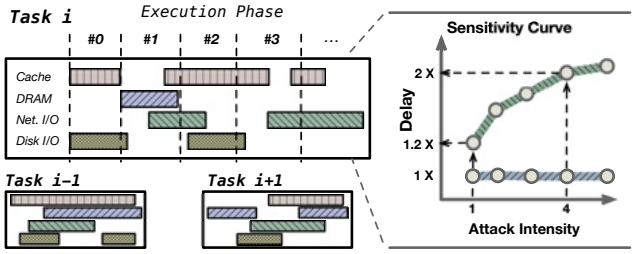


Fig. 4: Profiling resource contention sensitivity for tasks.

This fine-grained analysis benefits aggressor workload generation in three aspects: First, this analysis concentrates our focus on the most resource-demanding phases of the task. Launching attacks at these phases can be far more effective, allowing us to make a conscious trade-off between conserving the CPU budget, and triggering the delay at the appropriate phase. Second, separating each resource channel also helps to find the attack strategy that delays some specific threads without affecting the rest. Third, this fine-grained analysis facilitates TimeTrap to target only one resource channel at each phase, which reduces the search space while finding the optimal attack strategy.

Sensitivity analysis of interference channels. In TimeTrap, the aggressor workloads have to create an appropriate delay that corrupts the algorithmic results without triggering the fail-safe. To this end, we leverage the insight that a performance interference attack can be parameterized, thus allowing control over its delay extent by changing the attack parameters [9]. We first profile task delays under different levels of contention, with each level representing one set of attack parameters. As illustrated in Figure 4, for each resource channel, we incrementally increase the contention intensity and record the corresponding task delay. The attack intensity is controlled by tuning the parameters, such as memory access stride in memory-related contention. The result is a set of delays at varying discretized attack intensities. These recorded delays are then interpolated to produce a sensitivity curve [52]. This curve is utilized during the generation of an attack strategy (aggressor workloads) to predict the delay caused by any contention intensity (Section IV-C). Furthermore, we analyze sensitivity to different task alignments, yielding different collective resource usage profiles.

A limitation is that the relations between delay and attack parameters are non-linear, and the effects of different channels

are sometimes interdependent. Using the sensitivity curve to predict delay may introduce errors. One solution is to increase the granularity of sensitivity analysis to mitigate such errors.

C. Generation of Aggressor Workloads

Given the exploitable execution timing patterns and interference sensitivity, generating aggressor workloads is modeled as an optimization problem. Consider K tasks running on the target autonomous system, denoted as $\mathrm{T} = \{\tau_1, \tau_2, ..., \tau_K\}$. Among them, τ_v represents the victim task. Each task is characterized by a set of execution phases $\{p_1, p_2, ..., p_n\}$. The objective is to design an attack strategy $A_{p_i} = \{R_{p_i}, I_{p_i}\}$ for each phase p_i, with a focus on optimizing both the resource channel R and the attack intensity I. The influence of strategy A on the system is quantified by the delay $D = \{t_1, t_2, ..., t_K\}$ experienced by each task. This delay can be inferred using sensitivity profiles of the tasks, as determined in the second step (Section IV-B). We establish a mapping function S that relates the attack strategy A and the system state p_{sys} (representing the system-level execution phase) to the probability of time delay for each task $d_{\tau_i}, \tau_i \in \mathrm{T}$:

$$S(A, p_{sys}) \rightarrow D, D = \{d_{\tau_1}, d_{\tau_2}, ..., d_{\tau_K}\}, \tag{1}$$

where d_{τ_i} is the vector for task τ_i containing the delays under different probabilities. The goal is to maximize the time delay for the victim task d_{τ_v} while minimizing the delay in other tasks. By selectively delaying different invocations of the task, this mapping strategy can also introduce jitter. Simultaneously, d_{τ_v} has to lie within a proper range (obtained in Section IV-A) so as to avoid triggering fail-safe mechanisms. Therefore, we select the attack strategy that maximizes the following objective:

$$\begin{aligned} A^* &= \arg\max_A \sum_{i=1}^{K} (d_{\tau_v} - d_{\tau_i})^2, \\ s.t.\ & d_{\tau_v}^{\min} \leq d_{\tau_v} < d_{\tau_v}^{\max}; \quad d_{\tau_i} < d_{\tau_i}^{\max}, i \neq v. \end{aligned} \tag{2}$$

While the presented equations assume only one task needs to be delayed, the actual number varies with the platform. For complex systems like self-driving cars (e.g., Apollo [41] and Autoware [53]) with many tasks, up to three might need to be delayed, whereas drones typically require just one.

V. TimeTrap Implementation

In this section, we detail the implementation on Linux-based systems, such as OpenManipulator X and Turtlebot 3. Although diverse systems necessitate tailored attacks targeting specific resources and execution phases, the approach to constructing attacks on other platforms remains consistent.

Discovering Exploitable Conditions. The delay injection component of TimeTrap is realized by a compile-time instrumenter and a runtime system. The instrumenter is implemented as a compiler pass in LLVM-13 [54]. For inter-process communication, the target software predominantly uses ROS; we identify functions such as `publish()` as the delay injection points. For more fine-grained shared variables, we integrate the points-to analysis tool, SVF [48], into the pass and use it to identify the shared variables between threads. After identifying the injection points, the pass inserts the delay function before each one. The sleep duration is then dynamically set by the runtime system to mitigate natural variation and achieve more deterministic execution patterns. The delay function is transformed from the POSIX `usleep()` such that a delay injection point can wait until a specific time point. To analyze the computational results, a set of Python scripts has been developed to calculate the deviation in result output from its expected values for each task, serving as a gauge of control performance. Informed by the feedback from these outputs, the delay injection runtime system readjusts the duration of delay or removes the candidate pairs.

Performance Interference Sensitivity Analysis. Resource usage profiling is based on the Linux `perf` subsystem, which monitors performance events (e.g., cache misses) via hardware performance counters. The number of related hardware events (e.g., cache misses) indicates the intensity of using a specific resource. Given the limited number of counters, to minimize errors introduced by the multiplexing of performance counters [55], we perform and align multiple runs that each capture an equal number of hardware events to the number of counters.

To diversify the range of attack primitives, we implemented eight resource contention primitives based on the project PolyRhythm [9]. These include architectural channels such as *cache*, *row buffer*, *TLB*, and *memory bus*, as well as operating system-based channels like *network I/O*, *disk I/O*, *file system*, and *thread spawn*. By fine-tuning the respective parameters, we can achieve various attack intensities for each attack primitive.

VI. Evaluation

In this section, we *(i)* present three case studies of timing issues identified by TimeTrap, with two cases evaluated in real-world robots (Section VI-A); *(ii)* demonstrate that TimeTrap achieves effective attacks by exploring different factors (Section VI-B); and *(iii)* show how TimeTrap applies generally to different CPS platforms (Section VI-C).

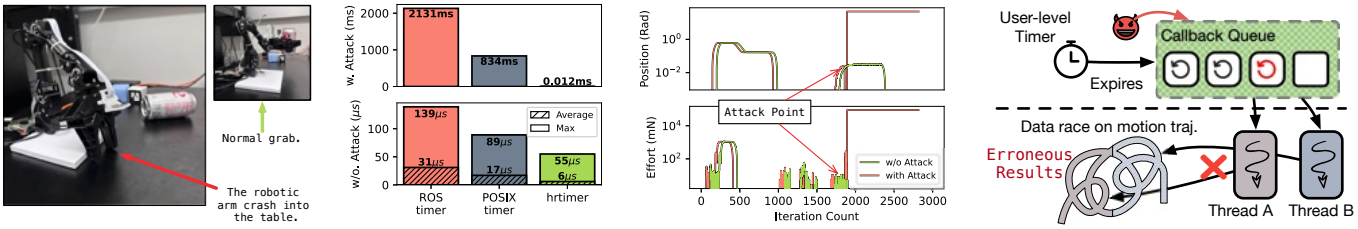
A. Case Studies

This section covers case studies on three platforms, with five more available on the website[8].

Case Study I on Robotic Arm. We first experiment on an open-source robotic arm, OpenManipulator X [33], as shown in Figure 5(a). The software stack we employ is the industrial-grade MoveIt [39], which is widely adopted in both industry [56] (e.g., NASA [57] and Intel [58]) and academia [59]–[61]. The robotic arm is integrated with a Raspberry Pi 3B for workload computation and a Pi Camera M2 for sensory input. In our attack scenarios, it is tasked with sensing and retrieving a soft drink can.

Attack Method. We schedule the MoveIt control tasks using priority 80 under the Linux SCHED_RR scheduling class.

[8] https://timetrap-rtas.github.io/



(a) Experimental environments. (b) Jitters on different timers. (c) Control digression. (d) Cause analysis.

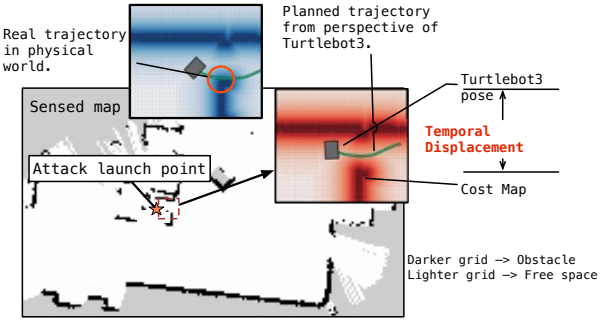
Fig. 5: Case study on a robotic arm.

TimeTrap is assigned priority 20. Utilizing the attack synthesis approach, TimeTrap identifies that the robotic arm exhibits anomalous behavior if there is a disparity in the release time of a victim task. To intentionally cause jitters in the victim, the aggressor workloads are tuned to contend for cache resources just before the scheduled release times of victim tasks.

Attack Result. The robotic arm is programmed to perform the following sequence: Initially, it identifies the can and approaches it. It then grips the can and transports it to the designated location. Under standard circumstances, the robotic arm can reliably grasp the can and accurately position it at the destination. However, when influenced by TimeTrap, the arm deviates from its pre-determined motion path and collides with the table, causing the can to fall (as shown in Figure 5(a)). Throughout this action, the intermediate control variables — including planned position and effort (torque) — spike to irregular levels. These trends are depicted in Figure 5(c). For clearer visualization, we have intentionally offset the plot representing the normal state slightly on the horizontal axis.

Root Cause Analysis — User-level Crafted Timer. Upon further investigation of the source code, the abnormal intermediate values are caused by a data race. The mission consists of multiple stages, with the robot following a subset of waypoints in each stage. Each stage activates a callback (thread) that generates a motion path. This path is calculated by an optimization solver that takes a subset of waypoints as input. However, if two tasks simultaneously generate motion trajectories, the solver processes inputs from two different subsets of waypoints, leading to incorrect results. In the implementation, each thread is triggered by a timer set at intervals of several seconds. Given that the computation itself only takes tens of milliseconds, data races normally do not happen. However, a key vulnerability lies in the userspace implementation of the ROS timer used by MoveIt, which makes it susceptible to disruption.

Figure 5(d) depicts the ROS timer mechanism: on delivery of a timeout signal, ROS places an entry into a queue for delivery to the next available thread. Figure 5(b) plots the amount of jitter that we produced with the attack. The kernel-level timer (`hrtimer`) is robust against TimeTrap due to its fast path in the kernel. However, due to the POSIX timer's userspace semantics, its release can be delayed up to 800 ms due to interference in the active task. The ROS timer is even worse, experiencing jitters extending beyond 2 seconds under attack. This manipulation by TimeTrap can cause delays that result in the simultaneous triggering of two subtasks (Threads A and B in the figure), causing errors. To conclude, though userspace timers are flexible and easy to implement, TimeTrap highlights their drawbacks in enforcing timing constraints.



(a) The two zoomed rectangles contain the generated costmap and the path. The path under temporal displacement of 0.76s is in red while the path in the real physical world is in blue.

```
// check that the observation buffers for the costmap are current,
// we don't want to drive blind
 1:    if(!controller_costmap_ros_->isCurrent()){
 2:        ROS_WARN("[%s]:Sensor data is out of date, we're not going to allow
commanding of the base for safety", ros::this_node::getName().c_str());
 3:        publishZeroVelocity();
 4:        return false;
 5:    }
```

(b) A code snippet of fail-safe mechanism.

Fig. 6: Case study on Turtlebot3.

Case Study II on Turtlebot3. This case study focuses on the Turtlebot 3 Burger, an indoor robot designed for home service, as seen in Figure 9. Equipped with an LDS-01 LiDAR and IMU, it uses Adaptive Monte-Carlo Localization (AMCL) and path planning in ROS navigation software [44] as the software stack. During tests in an office, the Turtlebot sends continuous location and task updates to a Ground Control Station (GCS) set on another PC. The GCS periodically sends trajectory data, guiding the Turtlebot's movement.

Attack Method. Similar to the earlier case study, the software stack integrates with Linux's real-time RR scheduler. An adversarial task, which holds a lower priority than all victim tasks, is also added. After the attack synthesis process, the

identified victim task emerges as the thread responsible for creating the map essential for motion planning.

Attack Result. As the Turtlebot navigates the room, it continually maps its surroundings, as the sensed map illustrated in Figure 6(a). For each control loop, the Turtlebot obtains a point cloud, matching these points with a previously established map database to deduce its current position. This point cloud also aids in the creation of a costmap, filled with obstacle information for the path planning module. TimeTrap can cause the path planning module to produce a wrong path based on a temporally mismatched costmap, thereby crashing the robot.

Root Cause Analysis — Inappropriate Timestamp Validation Checks. The Turtlebot's planning module consists of two planners: a global planner for mission trajectories and a local planner for real-time paths. The local planner's input includes the pose and a costmap detailing both static and dynamic obstacles in the current environment. For a viable path, both the pose and costmap must temporally align to correctly position obstacles. Yet, since these inputs update in separate threads, they are vulnerable to temporal displacement. TimeTrap causes Turtlebot to plan a path using a stale costmap that coincides with obstacles. Although mechanisms are in place to check the freshness of the costmap during the update of the global planner, as shown in the code snippet in Figure 6(b), there is no check for the temporal alignment between the costmap and the current pose when the software is using them together. With TimeTrap active, the Turtlebot generates its route based on the outdated costmap, shown in red in Figure 6(a). While the robot perceives its path as clear, it collides with an actual wall depicted in the blue costmap in the figure.

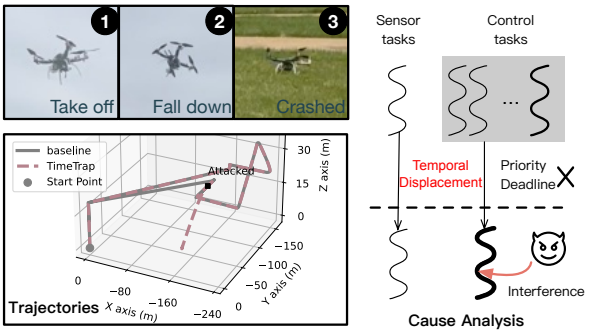


Fig. 7: Case study on Ardupilot drone.

Case Study III on Drone. We evaluated TimeTrap on a self-built quadcopter powered by a Raspberry Pi 3 Model B and Navio2 board. We deployed the widely-used Ardupilot [62] Copter-4.0 software. We sent *Mavlink* messages from another host PC to assign missions to the drone. As shown in the top-left of Figure 7, TimeTrap caused the drone to crash. To collect more detailed data, we also test the drone in simulation. Ardupilot supports only software-in-the-loop (SiTL) simulation, where physics simulation and control tasks are executed within the same cyclic task loop. In this setting, delays do not affect control performance, as they are translated into the SiTL's end-to-end delays. For proof-of-concept, we modified the sensor data emulation to run as a separate task, aligning with Ardupilot's normal flight mode, where sensors are fetched by distinct tasks.

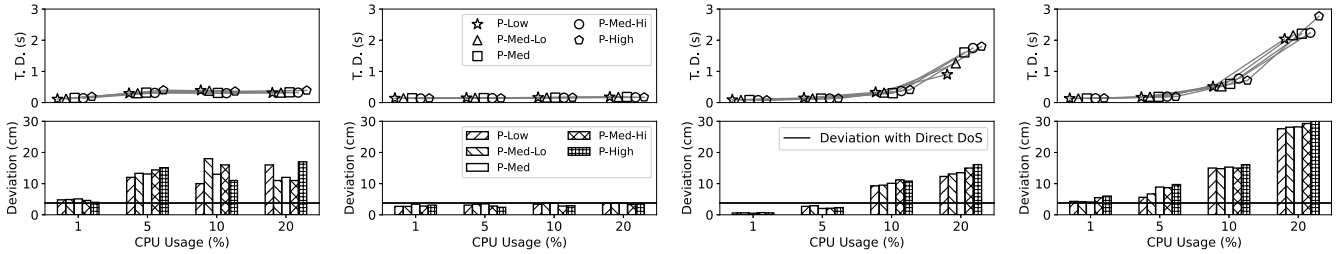
Attack Method and Result. Tasks in Ardupilot fall into two categories: input/output (I/O) and control. I/O tasks handle sensor drivers, control outputs, and remote messages, each having its own high-priority thread with a priority of 80. In contrast, control tasks, which process sensor data and compute control outputs, are grouped into a single low-priority thread with a priority of 60. A built-in user-level scheduler manages their execution. The adversarial task thread has the lowest priority, set at 20. We designated 11 waypoints for the drone's mission. At one specific waypoint, where the drone was set to make a sharp turn, we activated TimeTrap. Until that point, the drone maintained a steady speed. However, once TimeTrap was launched, the drone became unstable and crashed. its deviations from reference velocity are shown in Figure 7.

Root Cause Analysis — User-level Scheduler. The underlying issue stems from Ardupilot's reliance on user-level scheduling. In this setup, all control-related tasks are bundled into one thread, while sensor operations run on separate threads. Although user-level scheduling is efficient and versatile, it fails to provide solid guarantees of timing constraints. In essence, the kernel-level scheduling remains oblivious to priorities set at the user level. Furthermore, when there is a switch in the active thread at the kernel level, the user-level scheduling does not register this change. Consequently, the control loop task becomes unsynchronized with the sensor data loops, severely impairing the control's efficacy. As depicted in Figure 7, TimeTrap-induced data desynchronization caused the drone's velocity to exhibit substantial oscillations. It is crucial to understand that TimeTrap is not a rudimentary DoS attack that indiscriminately stalls all threads. If that were the case, the impact on control would be minimal, as the software is designed to withstand frequency reductions.

B. Effectiveness of TimeTrap

The experiments in this section aim to *(i)* showcase the effectiveness of the attack strategy and attack synthesis process in TimeTrap, and *(ii)* explore the factors that influence TimeTrap, including the resource channels, real-time execution priorities, and budgets of the attacking process.

Direct DoS is not Effective. To underscore the necessity of the attack strategy in TimeTrap, we demonstrate that a direct DoS attack, which exhausts cache resources [4], does not significantly impact the end-to-end control performance of either the Turtlebot or the robotic arm. We place a TurtleBot into the WashU Mini-City, a 3000 sq. ft. 1:8 scale city replica with realistic streets, buildings, cars, and pedestrians. In this scenario (shown in Figure 9), the adversarial task, allocated 80% CPU bandwidth and assigned a highest priority of 40, markedly delayed other tasks. However, rather than crashing into the building, the controller detected the excessive delay in input sensor data and activated fail-safe measures, halting



(a) Attack on disk I/O. (b) Attack on network I/O. (c) Attack on cache. (d) Attack on multiple resources.

Fig. 8: The effectiveness of the attack (control deviation) with regard to different attack vectors, priorities, and budgets. The four figures share the same legend. T. D. = temporal displacement.

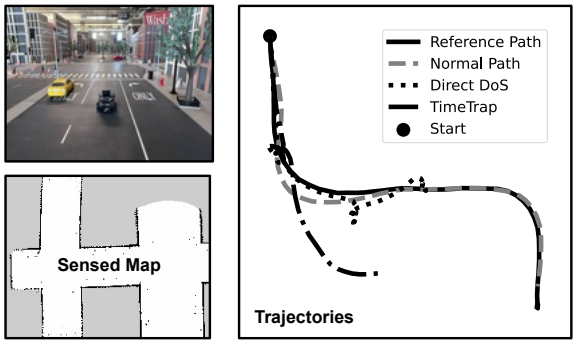


Fig. 9: Direct DoS attack vs TimeTrap on a real Turtlebot3.

the vehicle. There are 7 fail-safe conditions implemented in Turtlebot3 and Figure 6(b) shows one of them. Similarly, a direct DoS attack on the robotic arm resulted in only a minor control deviation of 3.78 cm, highlighted by the horizontal line in Figure 8. In contrast, an attack by TimeTrap could cause a deviation of up to 30.1 cm, as illustrated in Figure 8(d). This discrepancy arises because a direct DoS attack does not compromise the timer's functionality, the vulnerability explored in the previous case study. We also examined another type of direct DoS attack that does not deplete all resources, assigning the adversarial task a high priority of 40 but a low CPU bandwidth of 20%. The Turtlebot3 experienced a slowdown due to a decreased rate of control output messages but maintained its path. The robotic arm, however, exhibited only minimal temporal displacement (0.23 s) and control deviation (1.23 cm).

Impact Factors of TimeTrap. We assessed the effects of attack primitives on various shared resources, including cache, network, and disk I/O. Each attack primitive optimized a single shared resource. These individual attacks were then compared to an attack targeting a combination of resources. Additionally, we analyzed the influence of task priorities and CPU budgets. We assigned the adversarial task five levels of CPU bandwidths, ranging from 1% to 20%, and five levels of priorities: low, medium-low, medium, medium-high, and high. Here, "low" priority means the adversarial task's priority is lower than all other tasks. "Medium-low" matches the lowest existing priority in the system, while the other levels ascend accordingly. The evaluation is on the robotic arm with a Raspberry Pi 3B as the computing unit. Results are reported in Figure 8.

The attacks on disk I/O and cache are more powerful than those on network due to the system's minimal data transfers via the network. Robots utilizing SD card storage are particularly vulnerable to disk I/O attacks (Figure 8(a)), while SSD-based systems are less affected. Cache attack potency is closely tied to CPU budgets: more CPU time allows greater cache eviction. In contrast, network (Figure 8(b)) and I/O attacks (Figure 8(a)) demand less CPU time since they're managed by dedicated hardware controllers. However, such attacks yield limited temporal displacements. Attacks spanning multiple resources (as in Figure 8(d)) merge the strengths of individual ones. Lastly, as the CPU bandwidth increases, the significance of task priority becomes more obvious.

TABLE II: TimeTrap on Static and Dynamic Scenarios

		Scene	Sensing Overhead (%)	Temporal Disp. (s)	Control Dev. (cm)	Succ. Rate
Robotic Arm	TimeTrap	Static	0.18	1.6	22.1	72%
		Dyna.	0.22	1.9	25.7	86%
	Direct DoS	-	-	5.3	1.78	0%
Turtlebot3	TimeTrap	Static	0.19	0.93	13.2	61%
		Dyna.	0.32	0.81	14.5	81%
	Direct DoS	-	-	3.5	6.3	24%

Result of Direct DoS on static and dynamic scenarios are merged.

TimeTrap on Various Physical Scenarios. We established ten mission tasks, with half in dynamic scenarios. Specifically, the robotic arm had two 3.5 cm cubes as randomly placed target objects, while the Turtlebot navigated ten randomly generated indoor trajectories with diverse obstacles. Each task underwent 10 evaluations. Results in Table II show that the CPU usage during the sensing phase (sensing overhead) is minimal, and the attack success rates exceed 60% in both static and dynamic settings. Dynamic scenarios generally yielded higher success rates due to their stricter timeliness requirements.

We also tested the Direct DoS attack, where the adversarial task had a higher priority and consumed 80% of the CPU

budget. Its success rates were low across scenarios, as detailed in Table II. Despite causing substantial delays, Direct DoS could not notably deviate control behaviors.

C. Generalization of TimeTrap

Besides the three case studies, we tested eight cyber-physical systems: OpenManipulator X [33], Turtlebot3 [34], Jackal UGV [65], Ardupilot [62], PX4 Autopilot [63], ROBOTIS OP3 [64], Autoware.Auto [66], and Baidu Apollo [41] in a hardware-in-the-loop setup. Table III details the computing platforms, software stacks, scheduling mechanisms, and vulnerable modules identified. The evaluation is conducted in real-world scenarios or official simulations to ensure the practicality of TimeTrap.

Scheduling Mechanism Setup – Among the platforms tested, PX4 [63] utilizes an RTOS (Nuttx), while the others run on Linux. Autoware and Apollo also integrate ROS2 and Cyber-RT as their primary real-time middleware schedulers. For the platforms that default to Linux's Completely Fair Scheduling (CFS), we integrate their tasks into the Round Robin (RR) and Earliest Deadline First (EDF) real-time schedulers in Linux with the `PREEMPT_RT` patch. For RR, we assigned the target tasks a priority of 60 (with the highest being 99) and the adversarial task a priority of 20. We assigned the periods of tasks according to their input. For EDF, we aligned the deadlines with sensor input frequencies, ensuring no input drops.

Contention in Single-core Platform is less Effective. While single-core platforms also have shared stateful resources between tasks that can be maliciously polluted by adversarial tasks, the delay effects are limited. PX4, which uses the single-core platform Pixhawk (Arm Cortex-M7), did not crash by resource contention. However, we have identified inefficiency in priority inheritance in PX4, which can be triggered remotely. This case study is presented in Section VII.

Timing Issues Triggering Conditions. Our experimental results show that direct DoS attacks often do not guarantee a crash. (One exception is the drone where the naïve attack can also succeed due to unique aerial properties.) Their are two primary reasons. First, the naïve DoS attack, as described, requires excessive CPU budgets to cause temporal displacement. Second, significant delays on critical components can trigger the fail-safe mechanism in autonomous systems. Conversely, we discovered that even minor timing in TimeTrap can potentially cause devastating control degradation if the attack is launched at carefully selected moments. Additionally, we found that most of the timing issues discussed in Section II cannot be intentionally triggered in a predictable manner through performance interference. However, they still pose a problem, as they can be triggered by timing variability from workloads [67], system or hardware faults [68]. TimeTrap's temporal displacement discovery stage is capable of identifying these issues.

Various Modules in the Control Pipeline are Vulnerable. The proposed attack framework enabled us to identify vulnerable states in different realistic scenarios. The number of issues we identified are reported in Table III. We observe that the localization module is more prone to timing issues because it generally involves more threads, making it harder for programmers to reason about correctness in concurrent execution. We have examined the software patterns and scheduling mechanisms that make each platform susceptible to such attacks.

Built-in scheduling mechanisms offer limited mitigation on multi-core platforms. One reason is that middleware, which typically adopts data-driven execution models such as ROS2 and Cyber-RT, does not incorporate resource partitioning by design. Another reason is that many existing real-time scheduling mechanisms, such as those in RTOS Nuttx, only consider CPU usage during scheduling. Resource contention or blocking can occur across various stacks, such as the network or I/O stacks, with each single resource channel potentially leading to temporal displacements [9].

Static Timing Policy is Not Adequate. Specifying appropriate timing policies in complex autonomous systems is a challenge. In the platforms we investigated, policies primarily encompass static priorities and hardcoded thresholds for checking data freshness, which may be overly large in some scenarios. Given that a task's execution timings can vary significantly [69] due to multiple factors, such as input [70], autonomous systems interacting with dynamic environments necessitate adaptive, dynamic timing policies [69], [71].

VII. Limitations and Discussion

Attack Outcome Depends on Physical Environment. Attacks on CPS uniquely manifest in kinetic effects. The consequences of such attacks are often tied to the physical state of the system. For instance, a DoS attack on a stationary drone would lead to minimal real-world repercussions. Similarly, injecting false data into a power grid would be inconsequential if the generation is physically switched off. While TimeTrap's outcomes are heavily influenced by the system's physical state, our research indicates that it can consistently compromise the targeted CPS. The time taken for system destabilization can vary, however, sometimes requiring several seconds.

Remote Execution of TimeTrap. In addition to investigating performance interference and attacks on voltage configurations [72], we also explore methods to influence software timing in non-local environments, such as networked systems. We were able to send specially-crafted messages from a ground station to a drone running PX4 that cause priority inversion and delay task execution. In PX4, logging is handled asynchronously by a low priority (`SCHED_PRIORITY_DEFAULT - 40`) thread to avoid interference with critical tasks, since it consumes heavy IO bandwidth. The logging rate is further restricted using a variable `_log_interval` to avoid short, heavy data bursts.

However, PX4 handles user messages via the Mavlink protocol from the ground station in a high priority thread. User messages may request logging data, and the handler (`MavlinkLogHandler()`) also executes at the same high

TABLE III: TimeTrap on Various Autonomous Systems

Name	Type	Software Stack	Official Platforms	Evaluation Platforms	Tested Schedulings	# Issues in Vul. Modules				Direct DoS [4]	TimeTrap
						Percep.	Loc.	Plan.	Con.		
Turtlebot3†	UGV	Navigation [44]	RaspberryPi 4b	RaspberryPi 4b	{CFS,RR,EDF}+ROS		2	2			✓
Jackal		Google Cartographer [40]	Intel i3-4330TE	Intel i3-8100	{CFS,RR,EDF}+ROS		3				✓
Ardupilot	Drone	Ardupilot Sw. [62]	Navio2 (RPi3)	Navio2 (RPi 3b)	Built-in Scheduler				1	✓	✓
PX4		PX4 Autopilot [63]	Pixhawk	Pixhawk	Nuttx				1		✓*
Apollo	AV	Apollo full-stack [41]	Neousys 6108GC	Ryzen7 1700X	CFS+CyberRT	1	2	1			✓
Autoware		Autoware full-stack [38]	8-cores CPU	Ryzen7 1700X	CFS+ROS2	1	3	1	1		✓
OP3	Hum.	OP3 controllers [64]	Intel i3 dual core	Jetson Nano	{CFS,RR,EDF}+ROS				2		✓
OpenM. X†	Arm	MoveIt [39]	RPi 3b	RPi 3b	{CFS,RR,EDF}+ROS			1	1		✓

* is attacked remotely; † denotes the platforms evaluated in real world; The RAM size is also set as the same with official requirements; Percep = Perception; Loc. = Localization; Plan. = Path Planning; Con. = Controller.

priority. TimeTrap can send repeated log requests to the drone, causing it to deviate from its mission path or crash. The problem here is that the Mavlink handler retains its high priority when invoking functions; this causes the logging functionality to incorrectly inherit the high priority, leaving open a vulnerability to performance interference. To mitigate this, PX4 should specify a separate priority for running the function `MavlinkLogHandler()`, e.g., by propagating priorities with requests [73].

While most of the timing violations presented in this paper assume that a local process senses and induces interference, this shows that it is also possible to find a remotely exposed API that can invoke the interference workload.

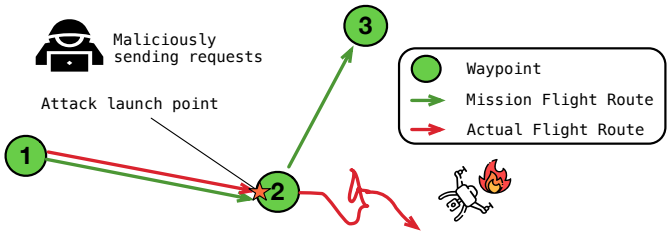


Fig. 10: Remote malicious requesting on PX4.

VIII. Related Work

Denial-of-Service Attacks via Performance Interference. DoS attacks have been extensively studied on various shared resources over the past two decades, with the primary goal being to characterize the maximum interference (slowdown) on different attack surfaces. These surfaces include individual channels such as cache [4], [12], [26], [27], [74], [75], row buffer [27], memory bus [30], network stack [28], I/O [29], and GPU [10]. Some works also consider combinations of different channels to achieve better interference [8], [9], [11], [76]. TimeTrap builds upon these prior works, taking a further step by synthesizing appropriate aggressor workloads to trigger timing issues, rather than maximizing interference.

Timing Security in Real-time Software. Given that timing is crucial for autonomous systems, several studies have been dedicated to investigating timing-related flaws and vulnerabilities. Degradation of controllers induced by input/output jitters is well studied in [77]–[79]. Synchronization attacks on smart grid [80] spoof the timestamps in GPS packets to desynchronize the phasor measurement units. The butterfly attack [81] assumes the existence of a software feature (vulnerability) that can be triggered using malicious input to increase the control jitters, thereby destabilizing the system. Mitigation methods for this issue are also explored in [24], [82]. These works primarily focus on the interplay between timing and the control model, while overlooking the software implementation. In this work, we go a step further by analyzing timing problems induced by the internal software implementation. A similar effort is from [83], which only analyzes the control performance of an AR application. Conversely, TimeTrap provides a more comprehensive study of control performance across the full stack of autonomous systems.

IX. Conclusion

This paper empirically studies how DoS attacks, facilitated by performance interference from shared resources, can trigger harmful task miss patterns, subsequently leading to control degradation in CPS platforms. We analyzed the results on several autonomous system platforms and formulated the cause as a problem of temporal displacement in critical data. To automate the analysis, we introduce TimeTrap, a tool specifically designed to analyze the end-to-end impact of performance interference on target CPS platforms. TimeTrap is capable of identifying potentially harmful task execution patterns resulting from software implementation flaws and synthesizing aggressor workloads to trigger these specific patterns. We conducted experiments on eight well-known autonomous systems in both real-world and simulated environments, and found that TimeTrap is applicable to different stages of the control pipeline across these platforms.

Acknowledgment

We thank the reviewers for their valuable feedback. This work was partially supported by the NSF (CNS-1916926, CNS-1932530, CNS-2038995, CNS-2154930, CNS-2141256, CNS-2238635, CPS-2229290), and ARO (W911NF2010141), Washington University in St. Louis, and Intel.

REFERENCES

[1] M. Andreozzi, G. Gabrielli, B. Venu, and G. Travaglini, "Industrial challenge 2022: a high-performance real-time case study on arm," in *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*, 2022.

[2] K. Nagar and Y. Srikant, "Precise shared cache analysis using optimal interference placement," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 125–134, IEEE, 2014.

[3] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele, "Worst case delay analysis for memory interference in multicore systems," in *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pp. 741–746, IEEE, 2010.

[4] M. Bechtel and H. Yun, "Denial-of-service attacks on shared cache in multicore: Analysis and prevention," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 357–367, IEEE, 2019.

[5] J. Giesen, P. Benedicte, E. Mezzetti, J. Abella, and F. J. Cazorla, "Modeling contention interference in crossbar-based systems via sequence-aware pairing (seap)," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 253–266, IEEE, 2020.

[6] M. Hassan and R. Pellizzoni, "Bounding dram interference in cots heterogeneous mpsocs for mixed criticality systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2323–2336, 2018.

[7] D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo, "A holistic memory contention analysis for parallel real-time tasks under partitioned scheduling," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 239–252, IEEE, 2020.

[8] D. Iorga, T. Sorensen, J. Wickerson, and A. F. Donaldson, "Slow and steady: Measuring and tuning multicore interference," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 200–212, IEEE, 2020.

[9] A. Li, M. Sudvarg, H. Liu, Z. Yu, C. Gill, and N. Zhang, "Polyrhythm: Adaptive tuning of a multi-channel attack template for timing interference," in *2022 IEEE Real-Time Systems Symposium (RTSS)*, pp. 225–239, IEEE, 2022.

[10] T. Yandrofski, J. Chen, N. Otterness, J. H. Anderson, and F. D. Smith, "Making powerful enemies on nvidia gpus," in *2022 IEEE Real-Time Systems Symposium (RTSS)*, pp. 383–395, IEEE, 2022.

[11] N. Singh, K. Renganathan, C. Rebeiro, J. Jose, and R. Mader, "Kryptonite: Worst-case program interference estimation on multi-core embedded systems," *ACM Transactions on Embedded Computing Systems*, vol. 22, no. 5s, pp. 1–23, 2023.

[12] M. Bechtel and H. Yun, "Cache bank-aware denial-of-service attacks on multicore arm processors," in *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 198–208, IEEE, 2023.

[13] N. Feiertag, K. Richter, J. Nordlander, and J. Jonsson, "A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics," in *IEEE Real-Time Systems Symposium: 30/11/2009-03/12/2009*, IEEE Communications Society, 2009.

[14] P. Gohari, M. Nasri, and J. Voeten, "Data-age analysis for multi-rate task chains under timing uncertainty," in *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, pp. 24–35, 2022.

[15] P. Pazzaglia and M. Maggio, "Characterizing the effect of deadline misses on time-triggered task chains," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, pp. 3957–3968, 2022.

[16] A. Davare, Q. Zhu, M. Di Natale, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli, "Period optimization for hard real-time distributed automotive systems," in *Proceedings of the 44th annual Design Automation Conference*, pp. 278–283, 2007.

[17] Y. Zhao, V. Gala, and H. Zeng, "A unified framework for period and priority optimization in distributed hard real-time systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2188–2199, 2018.

[18] T. Klaus, M. Becker, W. Schröder-Preikschat, and P. Ulbrich, "Constrained data-age with job-level dependencies: How to reconcile tight bounds and overheads," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 66–79, IEEE, 2021.

[19] M. Günzel, K.-H. Chen, N. Ueter, G. von der Brüggen, M. Dürr, and J.-J. Chen, "Timing analysis of asynchronized distributed cause-effect chains," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 40–52, IEEE, 2021.

[20] N. Vreman, A. Cervin, and M. Maggio, "Stability and performance analysis of control systems subject to bursts of deadline misses," in *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[21] A. Gujarati, M. Nasri, and B. B. Brandenburg, "Quantifying the resiliency of fail-operational real-time networked control systems," in *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[22] M. Maggio, A. Hamann, E. Mayer-John, and D. Ziegenbein, "Control-system stability under consecutive deadline misses constraints," in *32nd euromicro conference on real-time systems (ECRTS 2020)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[23] N. Vreman and M. Maggio, "Stochastic analysis of control systems subject to communication and computation faults," *ACM Transactions on Embedded Computing Systems*, vol. 22, no. 5s, pp. 1–25, 2023.

[24] 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)*, pp. 53–63, IEEE, 2023.

[25] J. M. Voas and G. McGraw, *Software fault injection: inoculating programs against errors*. John Wiley & Sons, Inc., 1997.

[26] D. H. Woo and H. Lee, "Analyzing performance vulnerability due to resource denial of service attack on chip multiprocessors," in *Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2007.

[27] T. M. O. Mutlu, "Memory performance attacks: Denial of memory service in multi-core systems," in *USENIX security*, 2007.

[28] H. S. Bedi and S. Shiva, "Securing cloud infrastructure against co-resident dos attacks using game theoretic defense mechanisms," in *ICACCI*, 2012.

[29] Z. Yang, H. Fang, Y. Wu, C. Li, B. Zhao, and H. H. Huang, "Understanding the effects of hypervisor i/o scheduling for virtual machine performance interference," in *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, pp. 34–41, IEEE, 2012.

[30] M. S. Inci, T. Eisenbarth, and B. Sunar, "Hit by the bus: Qos degradation attack on android," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pp. 716–727, 2017.

[31] T. Zhang, Y. Zhang, and R. B. Lee, "Dos attacks on your memory in cloud," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pp. 253–265, 2017.

[32] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. M. Swift, "Resource-freeing attacks: improve your cloud performance (at your neighbor's expense)," in *Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 281–292, 2012.

[33] ROBOTIS, "Openmanipulator x." https://emanual.robotis.com/docs/en/platform/openmanipulator_x/overview/. Accessed: 2022-9-07.

[34] "Turtlebot3." https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/. Accessed: 2022-01-10.

[35] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardos, "Orb-slam: a versatile and accurate monocular slam system," *IEEE transactions on robotics*, vol. 31, no. 5, pp. 1147–1163, 2015.

[36] C. Campos, R. Elvira, J. J. G. Rodríguez, J. M. Montiel, and J. D. Tardós, "Orb-slam3: An accurate open-source library for visual, visual–inertial, and multimap slam," *IEEE Transactions on Robotics*, vol. 37, no. 6, pp. 1874–1890, 2021.

[37] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *2012 IEEE conference on computer vision and pattern recognition*, pp. 3354–3361, IEEE, 2012.

[38] "Autoware auto gitlab." https://gitlab.com/autowarefoundation/autoware.auto. Accessed: 2022-12-12.

[39] "Moveit." https://moveit.ros.org/. Accessed: 2022-10-23.

[40] Google, "Google cartographer." https://google-cartographer.readthedocs.io. Accessed: 2022-03-29.

[41] "Baidu apollo open-source self-driving project." https://apollo.auto/platform/hardware.html. Accessed: 2023-08-15.

[42] "Orb-slam2 github." https://github.com/raulmur/ORB_SLAM2. Accessed: 2022-01-07.

[43] "Orb-slam3 github." https://github.com/UZ-SLAMLab/ORB_SLAM3. Accessed: 2022-01-07.

[44] "Ros navigation github." https://github.com/ros-planning/navigation. Accessed: 2022-01-01.

[45] "Ros2 rcl rclcpp rclpy libraries." https://github.com/orgs/ros2/repositories. Accessed: 2022-11-19.

[46] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pp. 329–339, 2008.

[47] G. Li, S. Lu, M. Musuvathi, S. Nath, and R. Padhye, "Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 162–180, 2019.

[48] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th international conference on compiler construction*, pp. 265–266, 2016.

[49] G. Bernat, A. Burns, and A. Liamosi, "Weakly hard real-time systems," *IEEE transactions on Computers*, vol. 50, no. 4, pp. 308–321, 2001.

[50] N. Vreman, R. Pates, and M. Maggio, "Weaklyhard. jl: Scalable analysis of weakly-hard constraints," in *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 228–240, IEEE, 2022.

[51] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, (New York, NY, USA), p. 45–57, Association for Computing Machinery, 2002.

[52] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, pp. 248–259, 2011.

[53] "Autonomous valet parking demo 2020." https://www.autoware.org/post/autonomous-valet-parking-2020. Accessed: 2022-12-12.

[54] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. CGO 2004.*, pp. 75–86, IEEE, 2004.

[55] S. S. Banerjee, S. Jha, Z. Kalbarczyk, and R. K. Iyer, "Bayesperf: minimizing performance monitoring errors using bayesian statistics," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 832–844, 2021.

[56] "Ros-industrial applications of moveit." https://moveit.ros.org/robots/. Accessed: 2022-10-23.

[57] "Picknik robotics wins space force, nasa contracts." https://www.therobotreport.com/picknik-robotics-wins-space-force-nasa-contracts/. Accessed: 2022-10-23.

[58] X. Shi, L. Cao, D. Wang, L. Liu, G. You, S. Liu, and C. Wang, "Hero: Accelerating autonomous robotic tasks with fpga," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 7766–7772, IEEE, 2018.

[59] C. Wang, J. Bingham, and M. Tomizuka, "Trajectory splitting: A distributed formulation for collision avoiding trajectory optimization," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 8113–8120, IEEE, 2021.

[60] S. Kohn, A. Blank, D. Puljiz, L. Zenkel, O. Bieber, B. Hein, and J. Franke, "Towards a real-time environment reconstruction for vr-based teleoperation through model segmentation," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 1–9, IEEE, 2018.

[61] M.-Y. Wang, A. A. Kogkas, A. Darzi, and G. P. Mylonas, "Free-view, 3d gaze-guided, assistive robotic system for activities of daily living," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 2355–2361, IEEE, 2018.

[62] "Ardupilot project." https://ardupilot.org/. Accessed: 2022-10-02.

[63] L. Meier, D. Honegger, and M. Pollefeys, "Px4: A node-based multithreaded open source robotics framework for deeply embedded platforms," in *2015 IEEE international conference on robotics and automation (ICRA)*, pp. 6235–6240, IEEE, 2015.

[64] "Robotis op3." https://emanual.robotis.com/docs/en/platform/op3/introduction/. Accessed: 2021-08-15.

[65] "Jackal ugv." https://clearpathrobotics.com/jackal-small-unmanned-ground-vehicle/. Accessed: 2021-07-30.

[66] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi, "Autoware on board: Enabling autonomous vehicles with embedded systems," in *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pp. 287–296, IEEE, 2018.

[67] M. Alcon, H. Tabani, L. Kosmidis, E. Mezzetti, J. Abella, and F. J. Cazorla, "Timing of autonomous driving software: Problem analysis and prospects for future solutions," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 267–280, IEEE, 2020.

[68] P. R. Nikiema, A. Kritikakou, M. Traiola, and O. Sentieys, "Impact of transient faults on timing behavior and mitigation with near-zero wcet overhead," in *35th Euromicro Conference on Real-Time Systems (ECRTS 2023)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

[69] I. Gog, S. Kalra, P. Schafhalter, J. E. Gonzalez, and I. Stoica, "D3: a dynamic deadline-driven approach for building autonomous vehicles," in *Proceedings of the Seventeenth European Conference on Computer Systems*, pp. 453–471, 2022.

[70] 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 *USENIX Security Symposium*, 2024.

[71] M. Sudvarg, A. Li, D. Wang, S. Baruah, J. Buhler, C. Gill, N. Zhang, and P. Ekberg, "Elastic scheduling for harmonic task systems," in *2024 Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE, 2024.

[72] 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)*, pp. 352–369, IEEE, 2022.

[73] M. Sudvarg and C. Gill, "A concurrency framework for priority-aware intercomponent requests in camkes on sel4," in *2022 IEEE 28th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 1–10, IEEE, 2022.

[74] D. Grunwald and S. Ghiasi, "Microarchitectural denial of service: Insuring microarchitectural fairness," in *MICRO*, IEEE, 2002.

[75] J. Hasan, A. Jalote, T. Vijaykumar, and C. E. Brodley, "Heat stroke: power-density-based denial of service in smt," in *11th International Symposium on High-Performance Computer Architecture*, pp. 166–177, IEEE, 2005.

[76] F. Zhou, M. Goel, P. Desnoyers, and R. Sundaram, "Scheduler vulnerabilities and coordinated attacks in cloud computing," *Journal of Computer Security*, vol. 21, no. 4, pp. 533–559, 2013.

[77] B. Wittenmark, J. Nilsson, and M. Torngren, "Timing problems in real-time control systems," in *Proceedings of 1995 American Control Conference-ACC'95*, vol. 3, pp. 2000–2004, IEEE, 1995.

[78] J. Nilsson, B. Bernhardsson, and B. Wittenmark, "Stochastic analysis and control of real-time systems with random time delays," *Automatica*, vol. 34, no. 1, pp. 57–64, 1998.

[79] A. Cervin, "Stability and worst-case performance analysis of sampled-data control systems with input and output jitter," in *ACC*, IEEE, 2012.

[80] Z. Zhang, S. Gong, A. D. Dimitrovski, and H. Li, "Time synchronization attack in smart grid: Impact and analysis," *IEEE Transactions on Smart Grid*, vol. 4, no. 1, pp. 87–98, 2013.

[81] R. Mahfouzi, A. Aminifar, S. Samii, M. Payer, P. Eles, and Z. Peng, "Butterfly attack: Adversarial manipulation of temporal properties of cyber-physical systems," in *2019 IEEE Real-Time Systems Symposium (RTSS)*, pp. 93–106, IEEE, 2019.

[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)*, pp. 2761–2778, 2023.

[83] M. Bechtel and H. Yun, "Analysis and mitigation of shared resource contention on heterogeneous multicore: An industrial case study," *arXiv preprint arXiv:2304.13110*, 2023.

[84] G. Rong *et al.*, "Lgsvl simulator: A high fidelity simulator for autonomous driving," in *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*, pp. 1–6, IEEE, 2020.

[85] P. Biber and W. Straßer, "The normal distributions transform: A new approach to laser scan matching," in *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)(Cat. No. 03CH37453)*, vol. 3, pp. 2743–2748, IEEE, 2003.

[86] W. Hess *et al.*, "Real-time loop closure in 2d lidar slam," in *ICRA*, IEEE, 2016.

[87] R. C. Coulter, "Implementation of the pure pursuit path tracking algorithm," tech. rep., Carnegie-Mellon UNIV Pittsburgh PA Robotics INST, 1992.

ADDITIONAL CASE STUDIES

This section presents extra case studies, which are evaluated using hardware-in-the-loop simulation, to demonstrate how TimeTrap can cause temporal displacements and how they instigate control deviation.

Case Study IV on Autoware.Auto. Autoware.Auto [66] is an open-source autonomous driving project, which is widely used for research and development. It contains the full-stack software packages required for self-driving cars, including detection, localization, planning, control, etc, and supports different application scenarios. Our experiments were conducted on Autoware.Auto 1.0.0 version. We run the entire software system on AMD Ryzen 7 1700X with 32 GB RAM in compliance with the official recommended platform (8 cores and 32GB). The simulated vehicle model is Lexus 2016 RX Hybrid and the scenario used is the officially released Autonomous Valet Parking [53]. For simulation, we run LGSVL [84] on another PC with a GPU of Geforce RTX 2070S. The communication between the platform being tested and the simulator is via Ethernet.

Attack Result. Figure 11 (b) shows the traveled path of the vehicle under test (VUT) on the map. TimeTrap was launched at the point marked by a star and it was periodically contending for shared resources to cause temporal displacements. On the remaining traveled path, as shown in Figure 11 (d), there exist two segments that incurred obvious temporal displacements by TimeTrap tinted in red. From the first segment, the VUT was driving straight with no obvious control deviation. However, as shown in the second segment, the VUT completely lost control and hit the adjacent building.

Cause Analysis. The reason behind this is the malfunction of the localization module, causing the vehicle to falsely estimate its position. In Autoware.Auto, localization uses the Normal Distributions Transform [85] (NDT) matching algorithm to estimate the pose of the vehicle. Since the optimization problem in NDT is modeled as a maximum a posteriori (MAP) problem in the software, an initial guess value is required to accelerate the solver. We use two code snippets in Figure 11(a) to demonstrate the vulnerability. As shown in Figure 11(a), the estimation of pose (defined as `pose_out`) relies on the initial value, which is defined as `initial_guess`. As to `initial_guess`, its value is inferred upon looking up (line 65) or extrapolating (line 71) the *transformation tree* in the system (in Code 11(a)). This inference is based on the assumption that the transformation graph `tf_graph` is in sync with `target_frame`. As the inference is time-related, a delay on the update of `tf_graph` cause result in a temporal displacement on `target_frame`, producing erroneous results in the critical variable `initial_guess`. Furthermore, the incorrect `initial_guess` causes the solver `register_measurement()` to fall into a local minimum, resulting in erroneous localization result `pose_out`.

By comparing the errors of `pose_out` in two segments that have large temporal displacements, in Figure11(d), it can be observed that although `initial_guess` had errors to the same extent in these two periods, the control was less affected in the first segment. This is because the VUT was driving straight during the first period and more common features persist in two adjacent frames of the point cloud, allowing the NDT to still converge albeit incorrectly `initial_guess`. In Figure 11(c), it can be observed that the localization result never deviated from the reference path in the first segment. On the contrary, the localization results are erroneous in the second segment, where the results were jumping arbitrarily. This indicates that the attack outcome depends on the physical state of the victim. If the vehicle was in a non-vulnerable state, attackers need more CPU budget or higher priority to cause substantial impacts on control performance.

Different Simulation Worlds. We also conducted experiments on different scenarios and the experimental results are reported in Table IV.

TABLE IV: TimeTrap Attack on Different Scenarios

Map	Scene Charac.	Sensing Overhead (%)	Temporal Disp. (s)	Control Dev. (m)	**Success-rate(%)**
Parking Lot	S	0.17%	2.0	3.6	72%
	D	0.21%	2.3	3.9	56%
Borregas Ave.	S	0.18%	2.2	1.8	77%
	D	0.36%	2.3	1.7	61%
GoMent.	S	0.17%	2.3	2.4	65%
	D	0.25%	1.9	2.2	52%
San Fran.	S	0.19%	2.7	4.8	74%
	D	0.34%	2.3	5.4	58%
Shalun	S	0.17%	2.4	2.1	64%
	D	0.23%	2.9	2.6	51%

S: Static Scenario; D: Dynamic Scenario.

From the results, we can observe that the sensing had larger errors in dynamic scenarios. This is because the randomly generated traffic participants can cause uncertain events such as slowdowns or stops. Those events make the sensing more challenging for TimeTrap. Moreover, to handle the unexpected events, our adversarial task took more time to filter the disturbance. Overall, the sensing part of TimeTrap achieved effective performance. The maximum overhead in the experiments is 0.36% CPU usage. As to the control deviation, the VUT running on *Borregas Avenue* was the less affected because its map of *Borregas Avenue* is relatively simpler. Even for the least control deviation $1.7m$, TimeTrap was still able to lead the vehicle to hit the curb.

Case Study V on Humanoid ROBOTIS OP3. ROBOTIS OP3 [64] is a miniature humanoid robot, which consists of 20 movable joints each equipped with a sensor module and a control module. The sensor module perceives the states of the robot (such as the velocity at any joint). The control module is designed based on a PID controller to generate the torque at each joint. Under the cooperation of the two modules, ROBOTIS OP3 can finish some complicated interactive tasks such as shaking hands, squatting down, standing up walking,

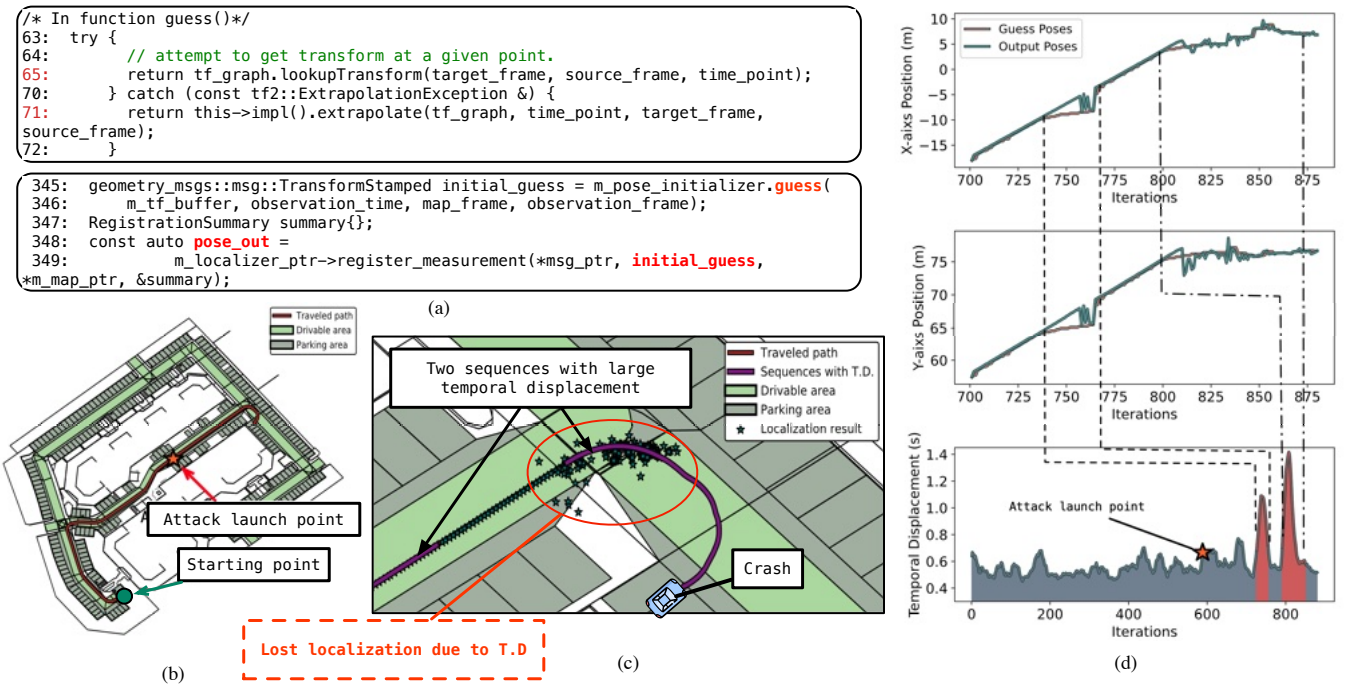
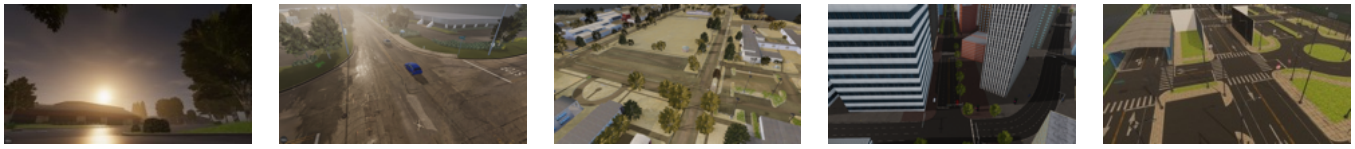


Fig. 11: Case study of Autoware.Auto. (a) Code snippets where temporal displacements happened. (b) Overall map and traveled path. (c) Zoomed traveled path and sequences with high temporal displacements that are in purple. (d) The values of two critical variables, `initial_guess` and `pose_out`, and their relations with temporal displacements.



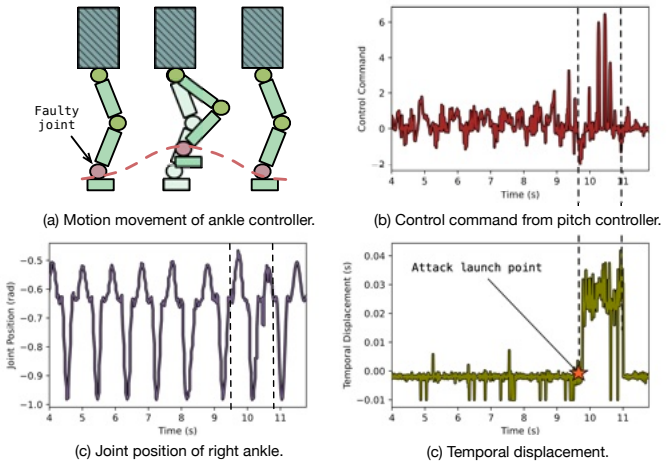
(a) Parking Lot (b) Borregas Avenue (c) GoMentum Station (d) San Francisco (e) Shalun

Fig. 12: Digital twins for testing Autoware.Auto in hardware-in-the-loop simulation environments. All those environments are simulated via modeling the real-world.

etc. We deploy the software stack of OP3 in Nvidia Jetson Nano. The simulation of OP3 is running on another PC in Gazebo. We generate a series of different tasks in advance and send them to OP3 one by one. OP3 receives the task and executes the action demanded. If it is already in the process of executing a task, any request will be ignored. During the testing, TimeTrap was randomly launched at different tasks.

Attack Result. In most cases, our attack cannot affect the stability of OP3. However, for some specific actions, such as walking, TimeTrap can effectively corrupt it, making the robot fall. Figure 13(a) shows the movement of joints in OP3. We observe that the movements of joints are highly similar and periodic with a specific time interval. However, some joints start to produce abnormal movements once TimeTrap is launched. This minor error breaches the coordination between joints, causing the robot to fall down. Once the CPU budget is beyond 5%, adversary can cause an attack outcome as shown in Figure 13.

Cause Analysis. At the control module of each joint in OP3, there are two concatenated controllers. The first one is the action controller that computes the target position a joint should reach in each control loop. The second one is the effort controller that receives the target position from the action controller and then computes the effort the servo should perform to reach the target position. Besides the input from the action controller, the effort controllers also take the sensor input of the Inertial Measurement Units (IMUs) in each joint to estimate the current position. Since they take inputs from two different sources: higher-level controllers and inertial sensors, the temporal displacement may happen within the effort controllers. From Figure 13(d), we can observe that the temporal displacement was relatively stable (around 0 ms) when it is not under attack. In this case, different joints coordinate well and move steadily. However, once the attack is launched, the temporal displacement could increase to 30ms to a point that the command calculated by the effort controller starts to oscillate (The interval marked by the dotted line in



(a) Motion movement of ankle controller. (b) Control command from pitch controller.

(c) Joint position of right ankle. (c) Temporal displacement.

Fig. 13: Case study on right ankle pitch controller of OP3. The period under attacking is fenced using dotted lines.

Figure 13(d)). As a result, the position of the right ankle joint turns out to be abnormal, which breaks the coordination between different joints such that the robot fell down.

Case Study VI on Jackal UGV. Jackal UGV [65] is an unmanned guided vehicle that can operate in both indoor and outdoor environments. We set up the simulation world in Gazebo and run the software stack on Intel i3-8100. Simulated hardware includes the four-wheel vehicle itself and a SICK LMS111 laser. The software stack is composed of a localization module (google Cartographer [40], [86]) and a control module is implemented based on the pure pursuit algorithm [87]. We pre-defined several long mission paths in an office environment (shown in Figure 14). The Jackal UGV navigates to track the reference path.

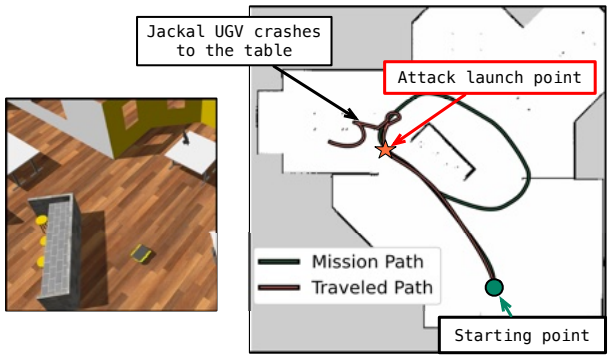


Fig. 14: Case study on Jackal UGV.

Attack Result and Cause Analysis. From Figure 14, we can observe that Jackal UGV failed to track the path and hits the table once TimeTrap is launched. In this case study, the crash was caused by erroneous location results. In Jack UGV's localization module, Cartographer, there is a sporadic task running in the background, responsible for optimizing accumulated drifts during the navigation. The optimization is supposed to release in every ten laser scans processed by the system. Since the optimization aims to mitigate the accumulated drifts, its finish will lead to a shift of localization value. The direct consequence is the vehicle's position will shift for a short distance. Such shift is negligible and within the tolerance of the controller if Cartographer can keep the task running within the implicitly (no deadline specified for this task) expected frequency. However, TimeTrap breaks the implicit deadline by constantly delaying the optimization task. Consequently, the task cannot finish in time and its workload keeps accumulating which increases the execution time in turn. After that, the vehicle's position shifts more significantly such that the controller can no longer keep tracking the reference path, leading to the crash on the table.