Who's Afraid of Butterflies? A Close Examination of the Butterfly Attack

Sanjoy Baruah*, Pontus Ekberg†, Mehdi Hosseinzadeh‡, Ao Li*, Bryan Ward§, and Ning Zhang*
*Washington University in St Louis. Email: {baruah, ao, zhang.ning}@wustl.edu
†Uppsala University. Email: pontus.ekberg@it.uu.se
‡Washington State University. Email: mehdi.hosseinzadeh@wsu.edu
§Vanderbilt University. Email: bryan.ward@vanderbilt.edu

***Abstract*—The Butterfly Attack, introduced in an RTSS 2019 paper, was billed as a new kind of timing attack against control loops in cyber-physical systems. We conduct a close inspection of the Butterfly Attack in order to identify the root vulnerability that it exploits, and show that an appropriate application of real-time scheduling theory provides an effective countermeasure. We propose improved defenses against this and similar attacks by drawing upon techniques from real-time scheduling theory, control theory, and systems implementation, that are both provably secure and are able to make efficient use of computing resources.**

I. Introduction

Mahfouzi et al. [1] had introduced the Butterfly Attack in a prior RTSS, billing it as "*a new attack scenario against cyber-physical systems that carefully exploits the sensitivity of control applications with respect to the implementation on the underlying execution platforms.*" The principle behind the Butterfly Attack (which we detail in Section II) is clever and conceptually very elegant; its effectiveness was demonstrated in [1] via a pair of case studies that were conducted in simulation. We have conducted a close investigation of the Butterfly Attack with the objective of understanding its significance and its applicability, and in order to develop effective mitigations — this manuscript reports our findings.

The Butterfly Attack [1]. Let us first provide a brief and superficial description of the Butterfly Attack; please see Section II for a more detailed discussion. It is assumed that a safety-critical control loop is implemented as a periodic task [2]; this task comprises part of a system that is modeled as a collection of independent periodic tasks and scheduled for execution upon a shared preemptive processor using the Rate-Monotonic (RM) fixed-priority scheduling algorithm [2]. This periodic task implementing the safety-critical control loop is the *target task* for the Butterfly Attack. There is a different *entry task* through which the attack is launched, that has higher RM scheduling priority than the target task, but is only responsible for less critical operations and therefore not as well protected as the target task. A malicious adversary launches the Butterfly Attack by interfering with the execution of the entry task so that it changes its timing characteristics in an apparently harmless manner: having it trigger successive invocations (release successive "jobs") farther than its period parameter apart. This change in the timing characteristics of the entry task causes in turn a change in the timing characteristics of the target task in a manner that had not been anticipated in the analysis that was conducted prior to run-time, and causes the control loop associated with the entry task to become unstable; since this control loop is safety-critical this compromises the safety of the system.

Protection against Butterfly Attacks. We will discuss this in greater detail in Section II-C, but briefly stated, Butterfly Attacks are successful against systems that are modeled using the *periodic* task model for the purposes of pre-runtime stability analysis. An effective preventative measure against Butterfly Attacks is straightforward: rather than using the periodic task model during stability analysis, instead model the system using the *sporadic* task model.

Improving the implementation. Although the straightforward fix discussed in the preceding paragraph does indeed provide adequate protection against Butterfly Attacks, it could potentially yield system implementations that make inefficient use of platform computing resources. (This equivalently means that it may be infeasible to implement the system in a secure manner upon the computing resources that are available.) We will see that efficiency of the implementation can be significantly enhanced via innovations in real-time scheduling theory and in control theory. Additionally, accurate measurement of system overheads, and the development of novel techniques for minimizing such overheads, enables us to best exploit these innovations and allows for a rational trade-off between security and implementation efficiency.

The big picture take-away message of our investigation of the Butterfly Attack has been that *the concurrent consideration of real-time, control, and systems security issues is needed in order to obtain correct and resource-efficient implementations of secure CPS's.* We have accordingly assembled a team of investigators with expertise in scheduling, control, and systems security in order to examine the Butterfly Attack from three perspectives –Real-Time Scheduling, Control Theory, and Systems Implementation– to better understand and to propose defenses against this and similar attacks that are both provably secure and are able to make efficient use of platform computing resources. Our **technical contributions** fall into three broad categories. In *real-time scheduling,* we devise a scheduling-theoretic countermeasure to Butterfly Attacks by essentially replacing the periodic task model that is used for pre-runtime stability analysis with the sporadic task model, and propose two techniques to improve the resource-efficiency of the secure system implementation (equivalently, make it more likely that the secure system will be deemed feasible to implement upon the available computing platform). From the perspective of *control theory,* we provide simulation-based evidence to advocate in favor of considering double-sided jitter (rather than single-sided jitter – these terms are explained in Section IV); for double-sided jitter, we develop algorithms for choosing controller parameters (latency and jitter) in a manner that makes it more likely that stability will be maintained. We also experimentally (again via simulation) demonstrate how hard it is to successfully launch a Butterfly Attack, by showing that merely changing the timing characteristics of the target task once is rarely sufficient for an attack to succeed: a sustained attack that forces the timing characteristics of the target task to change in very specific ways over a extended interval of time, is needed for the attack to have any real effect. Regarding our efforts in *computer systems security*, we have attempted to replicate the case studies that were conducted in [1] only in simulation, upon actual (i.e., physical) systems, but were largely unsuccessful in replicating the successes achieved in [1]; here we report on the key challenges in realizing Butterfly Attacks on physical platforms.

Organization. The remainder of this paper is organized in the following manner. We describe the Butterfly Attack [1] in Section II. In Section III, we describe how we have applied scheduling theory to strengthen systems against Butterfly Attacks, while simultaneously enhancing the efficiency of implementation. In Section IV we report on our investigations in applying control theory for similar purposes. In Section V we seek to identify the reasons behind our failure to replicate the simulation-based successful attacks of [1] in practice upon physical systems. We conclude in Section VI by providing some context to the research efforts reported here.

II. The Butterfly Attack

In this section we describe the Butterfly Attack as it is detailed in [1], providing explanations from control theory and real-time scheduling as needed.

A. The Target: A Critical Control Loop

Since the Butterfly Attack is based on destabilizing a control loop, let us start out by taking a closer look at control loops. A control loop seeks to optimize the *performance* of a plant, whilst maintaining its *stability*. To achieve this, it repeatedly executes a "sense-compute-apply" cycle –see Figure 1– by carrying out the following steps.

1) Selecting a *period* (usually denoted 'h' in the control literature) at which to repeat the sense-compute-apply cycle.

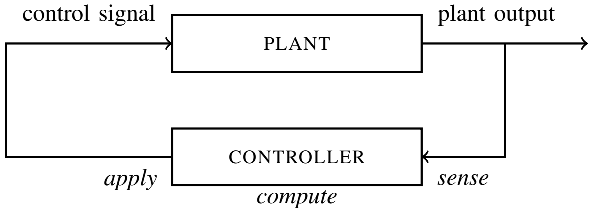


Fig. 1. A generic control loop: the sense-compute-apply cycle

2) For each $k \in \mathbb{N}$, reading in ("sensing") the plant output at some time-instant within the interval
$$[kh - J_{\text{in}}, kh + J_{\text{in}}]$$
Here J_{in} is called the *sampling jitter* (or *input jitter*) of the implementation. It is assumed in [1] that successive sensing operations happen exactly the same duration –the period– apart (i.e., *it is assumed in [1] that sampling jitter* $= 0$). We will continue with this assumption in this paper.
3) Using the sensed plant output to compute the control signal, and subsequently applying it to the plant; the duration of time that elapses between sensing and the subsequent application of the computed control signal is assumed to always lie within the interval
$$[L - J, L + J] \qquad (1)$$
for some $L, J \in \mathbb{R}_+$. Here L is called the *latency* and J the *output jitter* (or when input jitter is assumed to equal zero as is the case here, simply *jitter*) of the control loop implementation.
It is assumed in [1] that $J + L \leq h$ and $J - L \geq 0$; we will continue with this assumption in this paper.

The **stability region** of a control loop (see Figure 2 for an example) is a region on the jitter-versus-latency plane that defines the latency-jitter value combinations for which the control loop is *stable*. Specifically, the latency region specifies for each possible latency value (plotted on the x axis), the range of jitter values (plotted on the y axis) for which it can be guaranteed that the stability of the control loop is not compromised.

In [1], the stability regions for controllers are determined by using a methodology developed for this purpose that was reported by Cervin [3]; specifically, by making use of the "Jitter Margin" toolbox that is discussed in [3]. (The plot of Figure 2 above is generated by the Jitter Margin toolbox.).

B. Modeling Timing Behavior

Mahfouzi et al. [1], model the timing behavior of a safety-critical cyber-physical system as a collection $\mathbb{T}$ of independent periodic tasks [2] $\tau_1, \tau_2, \ldots, \tau_n$ that is scheduled upon a shared preemptive processor using Fixed-Priority (FP) scheduling with

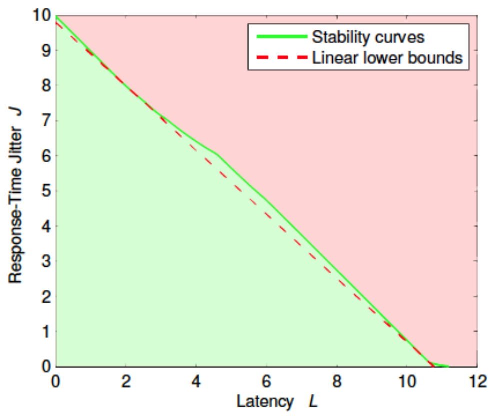


Fig. 2. Illustrating the stability region – this figure is taken from [1]

rate-monotonic priority assignment.[1] The control loop that is the target of attack is thus modeled as one of the periodic tasks, and is referred to as the *target task* (or *controller task*). Each $\tau_i \in \mathbb{T}$ is characterized by a period h_i, and a lower bound c_i^b and upper bound c_i^w on its actual execution duration each time it is executed. Task priorities are assigned according to the rate-monotonic rule: tasks with smaller periods have greater scheduling priority.

C. How the Butterfly Attack Is Launched

The Butterfly Attack is defined as follows in [1]:

Butterfly Attack (from [1, Definition 2])

"Indirect manipulation of temporal properties of a less critical task τ_j to modify the schedule of a more critical task τ_i such that the application Λ_i (τ_i relates to application Λ_i) is out of its expected behaviour is referred to as Butterfly attack."

We seek to emphasize a couple of points from this definition:

1) The use of the term "**critical** tasks": The task model in [1] assigns a criticality level to each task in the notation and spirit of Vestal [4], but does not make further use of the concept of mixed criticality (as is detailed in, e.g., the survey by Burns and Davis [5] and the hundreds of references cited there). As best as we can understand, it seems that for [1] a (more) critical task is one that can have a severe impact on plant/ system safety and is therefore designed to be specially secure from external attacks (whereas the remaining –less- or non-critical– tasks are not necessarily secure from external attacks).

2) The reference to "**expected** behaviour" in the definition merits discussion. It appears that the expected behavior of the application Λ_i alluded to in the definition above is any of the range of behaviors that Λ_i may have, given the behaviors of τ_i that are predicted by pre-run-time analysis.

As can be seen from the definition above, the **threat model** in the Butterfly Attack is as follows: A different task (the "less critical task τ_j" mentioned in the definition above, also referred to elsewhere in [1] as the *entry* or *adversary* task) has its timing characteristics changed "for the better" — the execution duration of some of its jobs is reduced to below its best-case execution time parameter value, or it releases jobs further apart than its period.

Pre-run-time analysis of the system as described in [1] consists of the following steps.

1) The worst-case response time R_i^w and best-case response time R_i^b of τ_i are computed.
2) The latency L_i of the controller task is set equal to R_i^b, and the jitter J_i to $\left(R_i^w - R_i^b\right)$. This is equivalent to assuming that the controller has latency equal to R_i^b, and any delay beyond this latency is considered to be jitter.
3) It is then checked whether this pair of (L_i, J_i) values lies within the stability region of the jitter-versus-latency plane of the controller being modeled by task τ_i; if so, task τ_i is declared to be safe from attacks that may compromise its stability.

We now discuss why such pre-runtime analysis is vulnerable to the Butterfly Attack. Recall that the threat model has two dimensions: (i) the entry task τ_j may reduce its execution duration (this dimension is dealt with by computing the minimum response time of τ_i's job in the synchronous arrival sequence); or (ii) it may increase the duration between successive job arrivals. This second dimension to the threat model in essence implies that the periodic task model is not the appropriate choice for modeling the timing behavior of the recurrent tasks in the system[2]. Indeed, it is not hard to see that the response time of τ_i's job is minimized if all the remaining tasks delay the release of their jobs by an arbitrarily large duration; in this event, τ_i's job executes without contention and its response time is therefore simply its own execution duration, which is, as per the model adopted in [1], lower-bounded by c_i^b. Hence a *safe* lower bound on R_i^b is given by task τ_i's own best-case execution time parameter c_i^b; in the absence of additional information about the remaining tasks, this is also the tightest lower bound one is able to obtain.

III. Applying Real-Time Scheduling Theory

We now specify a series of adaptations to modify and generalize the pre-runtime analysis of a system modeled as in [1] (and

[1]It is pointed out in [1] that the Butterfly Attack may be adapted to other algorithms/ workload models; however, such generalizations are not really explored in further detail.

[2]This is because FP-schedulability of periodic task systems is not *sustainable* [6]–[8] with respect to the period parameter: a periodic task system that is schedulable may cease to be so if the period parameter of one or more of the tasks in it is increased.

described in Section II above), that together serve to (i) render the system safe from Butterfly Attacks; and (ii) enhance the efficiency of the implementation. The modifications put forth in this section are all primarily based on the application, in a security-cognizant and control-aware manner, of principles from real-time scheduling theory.

Our **first** (and obvious) **adaptation** is to consider the task system $\mathbb{T}$ to be a *sporadic* task system rather than a periodic one, and hence replace the computation of R_i^b [1, Eqn (6)] with the safe upper bound that we had discussed in Section II-C above:

$$R_i^b \leftarrow c_i^b$$

This provides an adequate countermeasure to Butterfly Attacks: no manipulation of any number of entry tasks will cause the target task to operate outside its stability region.

A consequence. Observe that this bound is almost always significantly smaller than the value that would be computed by [1, Eqn (6)], and hence we end up with a smaller value of L_i and a larger value of J_i. As a general rule, the boundary of the stability region in the jitter-versus-latency plot (such as Figure 2) has a slope > -1: i.e., going leftwards by an amount Δ results in an increase in the permitted jitter that is $< \Delta$. Hence it is quite possible that a control loop that was deemed to be within the stability region under the pre-runtime analysis described in [1] will now be seen to not lie within the stability region after all, and hence systems that were previously declared to be stable will now be considered potentially unstable.

Under the assumption that at least some of the tasks (e.g., the critical ones) are not vulnerable to the attacker being able to delay the release of their jobs,[3] our **second adaptation** seeks to compute an improved (i.e., a larger) value for R_i^b. Under this assumption, this subset of tasks are all periodic (rather than sporadic), and it is not hard to show that R_i^b, the best-case response time of τ_i, is equal to the smallest response time that is experienced by any of τ_i's jobs in the FP schedule of the synchronous arrival sequence of just these periodic tasks over an interval from zero to the least common multiple of their periods. This smallest response time of any of τ_i's jobs is easily determined by simulating the schedule. Since such simulation takes time proportional to the least common multiple of the periods, one may wonder whether one can do better; unfortunately, the answer appears to be NO. This is because we show in the **appendix** that the Best-Case Response Time (BCRT) Problem, defined as follows:

[3]This seems a reasonable assumption, since it is already assumed in [1] that the target task itself is secure against such attacks; hence other critical tasks are also likely to be similarly secure. In the event that there are no tasks other than τ_i that are invulnerable to such an attack, this modification has no impact, positive or negative, on the overall implementation.

The Best-Case Response-Time (BCRT) Problem

INSTANCE: An FP-scheduled sporadic task system $\mathbb{T}$ (with best- and worst-case execution times specified), and a positive integer a.

QUESTION: Does each job of the lowest-priority task in $\mathbb{T}$ have a response time $\geq a$ in the schedule of its synchronous arrival sequence?

is coNP-hard. This essentially implies that we should not expect to find a polynomial-time algorithm for solving it; additionally, the fact that this problem is coNP-hard (rather than NP-hard, as FP-schedulability is known to be [9]) offers some evidence that simple recurrence equations similar in form to the standard response-time equation (see any standard text-book, e.g. [10], or a tutorial paper for an explanation):

$$R_i^w = c_i^w + \sum_{\tau_j \in hp(\tau_i)} \left\lceil \frac{R_i^w}{h_j} \right\rceil c_i^w \qquad (2)$$

that is used to compute the worst-case response time of FP-scheduled tasks are unlikely to be successful in computing BCRT's.

An additional observation: Notice that the pre-runtime analysis described in [1] requires bounds on both the best-case response times and the worst-case response times (i.e., it needs both the R_i^b and the R_i^w values). Since it is known [9] that determining the worst-case response time is NP-hard, our proof in the appendix that determining the best-case response time is coNP-hard immediately implies that the problem of determining both the upper and the lower response-time bounds is hard for both the complexity classes NP and coNP. It is widely believed in the computational complexity theory community (see, e.g., [11]) that a problem that is both NP-hard and coNP-hard is unlikely to be contained in either of the complexity classes NP and coNP, leading to the conclusion that *the pre-runtime analysis of [1] in fact requires solving a problem that does not lie in the first level of the polynomial hierarchy [11].*

Our **third adaptation** is based on the observation that the Butterfly Attack requires the attacker to choose an entry task that has greater scheduling priority than the target task. The attack surface would hence be reduced by having fewer non-critical tasks at higher priorities. To this end, we propose a modification to the manner in which task priorities are assigned; rather than simply assigning priorities in rate-monotonic order, we propose that priority-assignment be done according to the Audsley algorithm [12], [13] for repeatedly choosing the lowest-priority task but with non-critical tasks favored over the critical ones for this purpose. (That is if there is a choice of tasks to which the lowest priority may be assigned, we will assign it to a non-critical task if possible – this results in fewer non-critical tasks being assigned higher priorities.) This algorithm can be implemented efficiently in the following manner.

1) Maintain two lists that together contain all the tasks that

have not yet been assigned a priority. One list comprises the critical tasks and the other the non-critical tasks, each sorted in non-decreasing order of period parameter. (These lists are initialised to contain all the critical/ non-critical tasks respectively.)

2) While both lists are non-empty, repeatedly determine the task that is to be assigned the lowest priority task from amongst all the tasks that have not yet been assigned priorities, as follows.
   - First, determine whether the largest-period non-critical task that has not yet been assigned a priority is able to be the lowest priority task. This can be determined by checking whether this task τ_i's worst-case response time as computed by Equation 2, with $hp(\tau_i)$ set equal to all the other remaining tasks in both lists, is no larger than h_i. If so, assign this task the lowest priority from amongst all the tasks that have not yet been assigned priorities, and remove it from the list.
   - If not, determine whether the largest-period critical task that has not yet been assigned a priority is able to be the lowest priority task. This, too, can be determined as above: by checking whether this task τ_i's worst-case response time as computed by Equation 2, with $hp(\tau_i)$ set equal to all the other remaining tasks in both lists, is no larger than h_i. If so, assign this task the lowest priority from amongst all the tasks that have not yet been assigned priorities, and remove it from the list.
   - Else, report failure and exit.

3) If one of the lists is empty, then the tasks in other list are assigned rate-monotonic priorities.

A heuristic refinement. We will see in Section IV below that if the controller is designed to allow for 'double-sided jitter' (this term is explained in Section IV), then once the best-case and worst-case response time values of the task modeling the controller have been computed there is some freedom in designing the controller to optimize for performance whilst remaining in its stability region. The priority-assignment algorithm discussed above may be adapted to account for this fact in the following manner. If during some iteration of the priority-assignment algorithm it is a critical task that must be assigned the lowest priority, one may take into consideration the process of actually designing the controller (as discussed in Section IV) in addition to simply determining whether or not its worst-case response time is less than its period. In this manner we get to generalize the application of Adusley's priority-assignment algorithm to the 'real-time plus control' use-case, to consider not just timeliness (the ability to meet deadlines) but also control performance and stability in determining the assignment of priorities to individual tasks.

IV. Applying Control Theory

In this section, we take a closer look at the control-theoretic aspects of the Butterfly Attack. Specifically, we focus upon the control attributes of a control loop that needs to be defended against such an attack, and consider both (i) what can be done to render such attacks less likely to succeed against this control loop; and (ii) how likely an attack that has been launched is to succeed in actually destabilizing the control loop (as opposed to merely exiting the stability region in its latency-versus-jitter graph plot).

Single-sided and double-sided jitter. In Section II-A, we had defined jitter J (Expression 1) such that it could be either additive $(L+J)$ or subtractive $(L-J)$ with the latency parameter L to yield the bounds on actual delay. In control theory, two notions of jitter are studied: an additive/ subtractive notion such as is described above is referred to as *double-sided* jitter, while in *single-sided* jitter the range of actual delays is given by $[L, L+J]$. Although Mahfouzi et al. [1] cite the work of Cervin [3], which primarily addresses double-sided jitter (see, e.g., [3, Fig. 2]), for their notion of jitter, it appears from a careful reading of [1] that they are actually using single-sided jitter (notice, as discussed in Section II-C, that they choose to assign $L_i \leftarrow R_i^b$ and $J_i \leftarrow (R_i^w - R_i^b)$; hence actual delay is never less than L_i — jitter is never negative).

Regardless of whether Mahfouzi et al. [1] actually use single-sided or double-sided jitter in their models, we conducted extensive evaluations comparing the two, and find the evidence to be very strong that double-sided jitter is generally preferable for both control loop performance and stability. Briefly, this is because a control algorithm is generally designed to apply an actuation signal to the controlled plant a particular duration after having sampled the plant output — this is the latency parameter for the controller. Controller performance is best if the actuation signal is actually applied to the plant with exactly this delay, but tends to falls off if the actuation signal is applied at a different point in time – the greater the difference (i.e., the greater the magnitude of the jitter), the poorer the performance. Hence for best performance, it is best to minimize the magnitude of jitter; it is generally the case that the magnitude of jitter can be minimized if it is allowed to be both positive and negative.

Of course, the stability region of the jitter-versus-latency plane depends upon whether single-sided or double-sided jitter is assumed; in generating the stability region plot (as in Figure 2), Cervin's Jitter Margin toolbox [3] allows the user to specify which form of jitter is being modeled. In our evaluations, we used the Jitter Margin toolbox [3] to obtain the stability regions for a wide range of controllers under both assumptions; after extensive comparisons we feel comfortable in asserting that both worst-case performance and stability appear to be superior for double-sided jitter, for a wide range of worst-case performance metrics.

Double-sided jitter: choosing latency & jitter values. For single-sided jitter, the assignment of values to the latency and jitter parameters that was done by Mahfouzi et al. [1] (as we have described in Section II-C) seems exactly right: latency *must* be set to the best-case response time, and jitter *must* be set to the difference between worst-case response time and best-case response time. For double-sided jitter, however, the values of the best-case and worst-case response times do not uniquely determine the values that should be assigned to these two parameters;[4] for instance, one could minimize the jitter magnitude (and thereby minimize the worst-case performance degradation that could possibly be experienced on any single iteration of the control loop) by simply setting latency to be the mean of the best-case and the worst-case response times – this would cause jitter to be equal to half the difference between the worst-case and the best-case response times.

But such a choice, while perhaps minimizing performance degradation, does not address the issue of control-loop *stability* at all: it is entirely possible that the pair of values for latency and jitter pair computed above does not lie within the stability region of the control loop. What we are essentially seeking is a point (L_i, J_i) within the stability region of the controller modeled by τ_i such that both the following hold

$$\begin{aligned} L_i - J_i &\leq R_i^b \\ L_i + J_i &\geq R_i^w \end{aligned}$$

(Here, R_i^b and R_i^w denote the computed best-case and worst-case response times of the control task τ_i.)

The choice of this point (L_i, J_i) is complicated by the fact that although (as mentioned earlier – footnote 4) the running time of a controller does not much depend upon the specific parameter values used in its computation, its stability region very much does. Hence the step of finding an appropriate pair of (L_i, J_i) values is likely to be an iterative search process: while scheduling theory can be deployed (as described in Section III to obtain the values of R_i^b and R_i^w, one may need to repeatedly iterate to find (L_i, J_i) values within the stability region of the current controller configuration, update the controller for the value chosen for L_i and determine the stability region for the updated controller, and ensure that the point (L_i, J_i) continues to line in the stability region (while providing good performance) for this updated controller.

How conservative is the stability region? Stability regions of the jitter-versus-latency plane (such as the one depicted in Figure 2) represent safe choices from the perspective of stability: controllers with (latency, jitter) parameters that fall within the stability region are guaranteed to be stable. However, controller with their (latency, jitter) parameters that lie outside the stability region are not guaranteed to be unstable; e.g., the model underlying the Jitter Margin tool that generated these plots "does not care about *how* or *whether* the actual controller timing varies from period to period" [3, Sec. II-B]. Thus a successful instantiation of the Butterfly Attack would need to not only change the jitter and latency parameters enough to exit the stability region, it would need to sustain a 'worst-case' pattern of enforced delays (which it is very non-trivial to determine, let along achieve during an attack) across multiple consecutive invocations (i.e., jobs) of the controller task. In the absence of such a concerted effort across multiple invocations, Butterfly Attacks appear to be very unlikely to succeed. As an illustrative example, we considered the operation of system and controller discussed in Figure **??** outside its stability region (as determined by the sufficient analysis of the Jitter Margin toolbox), by simulation (across 1000 runs for each plotted point) with the actual jitter experienced by each invocation of the controller being uniformly drawn from the interval $[0, J]$ where J denotes the maximum jitter (i.e., the parameter that, for the chosen latency value, lies outside the stability region). Our findings are depicted in Figure 3; the boundary of the stability region in the jitter-versus-latency plane that is determined by Jitter Margin (assuming single-sided jitter) is depicted in blue, and the boundary of the region within which the system remains stable in these simulations is depicted in red. The takeaway message again appears to be that *the Butterfly Attack is exceedingly difficult to carry out in practice*; this has certainly been in case in an actual physical setting, as we discuss in the next section.

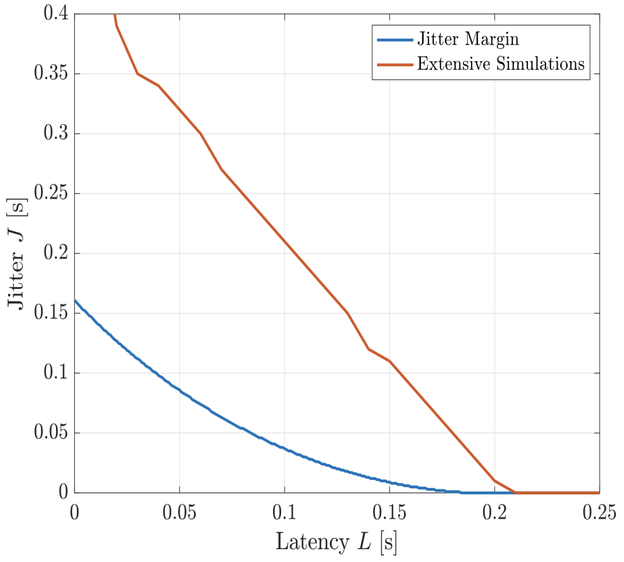


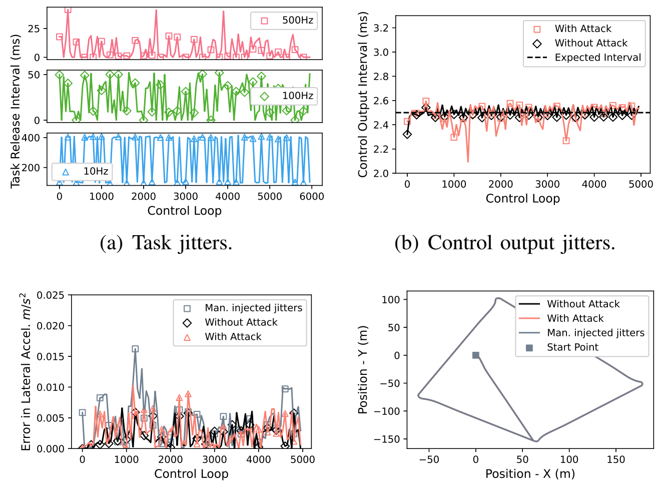
Fig. 3. How conservative is the stability region determined by the Jitter Margin tool [3]? – stability regions as determined by Jitter Margin versus by simulation. (The lines denote the upper envelopes of the stability regions.)

V. Applying Principles of System Security

The success of the Butterfly attack in real systems hinges on the success of three essential steps. **`S1`**: The first step is to manipulate the timing of an entry task via external inputs, where the tasks under attack are often the non-critical and less protected tasks. In general, the greater the magnitude and the

[4]We point out that a single controller execution typically comprises a series of matrix operations. While the values populating these matrices do depend upon the latency for which the controller is being designed, the dimensions of the matrices, the number of operations, etc., do not change; hence the execution-time of the code implementing the controller does not generally depend significantly upon the specific latency for which it has been designed. Hence, one can complete the design of the controller –i.e., populate the values in the matrices– *after* the value to be assigned to L_i is determined.

finer-grained the control, the higher the likelihood of causing the butterfly effect on the rest of system. **S2**: The second step is to exploit the timing dependency between the entry task and the target task to manipulate the timing of the target task using the influence on the entry task. Generally, both predictability and strength of the influence upon the target task are the key factors. **S3**: The last step transfers the timing impact to variation in control performance. The success of this step hinges on the control algorithm's sensitivity to the effected manipulation of the critical control task's timing. As a result, the impact of the attack on the physical state of the system depends on not only the successful execution of individual steps, but also the interplay of these factors. In the remainder of this section, we aim to share our experience realizing the attack on two CPS platforms to highlight the gaps our community needs to bridge towards a more holistic understanding of the exploitation process of timing vulnerabilities in cyber-physical systems.[5]



(a) Task jitters. (b) Control output jitters.

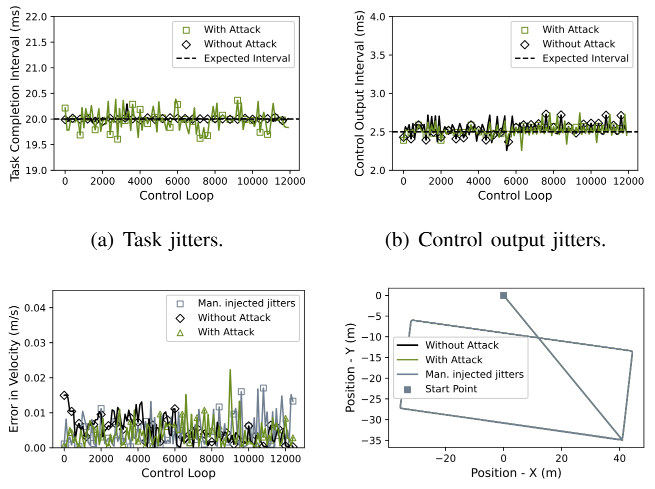
(c) Control errors. (d) Trajectories.

Fig. 4. Experimental results on ArduRover, an unmanned ground vehicle. The line in Figure 4(a)-4(c) is downsampled by 50, and the marker is downsampled by 200 to enhance the visualization.

A. Case Study I: an Automotive Application

The experimental setup in the original Butterfly Attack paper [1] assumes an architecture where multiple ECU functionalities (e.g., motor control, propulsion-control software, etc.) are on the same ECU. This is a futuristic setting with no existing automotive system adapting such a design to the best of our knowledge. Furthermore, there is no target software/hardware described in the original paper. Hence we aren't able to realize the attack on actual hardware, software and physical platform. To make the best effort in recreating the experiment under the same spirit, we conduct this experiment using a ground vehicle with ArduRover, where all controls are on the same SoC.

[5]Experimental artifacts are available at https://butterflyexploitation.github.io/



(a) Task jitters. (b) Control output jitters.

(c) Control errors. (d) Trajectories.

Fig. 5. Experimental results on ArduCopter, an unmmaned aerial vehicle. The line in the Figure 5(a)-5(c) is downsampled by 100, and the marker is downsampled by 400 to enhance the visualization.

With the change of platform, a key design choice we had to make was how to launch the attack on the new system where components communicate not via CAN bus but direct messages within the same program. The core idea of the attack is to use a message to influence the entry task, and this is similar to the radio control message in ArduRover which is implemented as MAVLink message. Therefore in our experiment we use the MAVLink message as the attack vector for influencing the timing behavior of the entry task in the system. The experiments are conducted on an autonomous ground vehicle powered by Raspberry Pi 3 Model B coupled with a Navio2 peripheral daughter board [14]. The software is ArduRover [15] package (version Rover-4.0) running on a single SoC (Pi3).

The attack we implemented is as follows. The entry task is `handleMessage`, which is a non-critical task responsible for processing Mavlink messages. To have a similar message pattern as CAN bus, we use a ground station to send Mavlink messages to ArduRover periodically. The adversary intentionally jams three out of every four messages, following the same method in [1], to induce jitters in the task `handleMessage()`. There are two main factors from the `handleMessage()` task that determine the degree of jitter: its period and computation time. Since the workload of `handleMessage()` stays fixed, we explore the impact at different frequencies (10Hz, 100Hz, and 500Hz). Note that most of the existing vehicles generally generate the heartbeat message at 10 Hz [16], therefore the settings under investigation is advantageous for the attacker.

Step 1 - We measured the actual task release intervals of `handleMessage()` under attack in the three said experimental settings. The results are shown in Figure 4(a). The CPU time consumed is 0.00125%, 0.11%, and 0.48% respectively. From the figure, we can observe: First, the task release intervals can be directly manipulated by attackers, demonstrating the attacker's capability to execute a predictable attack. Second, while the CPU consumption remains relatively low, it increases linearly with the message frequency. This linear relationship can be attributed to the predictable and straightforward workload associated with handling Mavlink messages.

Step 2 - In ArduRover, `Rover::set_servos()` sends the control output at a frequency of 400Hz. To assess its sensitivity to jitters introduced by the `handleMessage()` task, we measured its task release time across our three experimental scenarios. Under normal conditions, the average interval is 2.49 ms with a variance of 0.0063 ms. For the three different settings, the average intervals are 2.49 ms (variance: 0.0022 ms), 2.49 ms (variance: 0.0042 ms), and 2.48 ms (variance: 0.0081 ms), respectively. Figure 4(b) plots the results for the setting with a frequency of 500Hz, which showed the most significant impact. The limited impact from the attacks is due to two key factors. First, the `Rover::set_servos()` task is time-triggered and therefore task release times remain the same, even if additional CPU resource is available. Second, the computation time of `handleMessage()` task remains relatively lightweight compared to the aggregated load of the processor. This highlights the importance of identifying an entry task where there exist strong temporal dependency between entry task and victim task, which can be non-trivial to do.

Step 3 - Figure 4(c) shows the average control errors in lateral acceleration under both attack and baseline conditions. Lateral acceleration is used as a representative control state due to its direct impact on safety, and it is also the metric with the largest deviation in our experiments, which represents a worst case from our empirical study. Under attack conditions, the average error is $0.00237 m/s^2$ with a variance of $0.000037 m/s^2$. In contrast, the baseline condition exhibits an error of $0.00223 m/s^2$ with a variance of $0.000031 m/s^2$. This represents a 6.3% increase, yet it wasn't significant enough to cause the control system failure. To further understand the feasibility, a jitter of 0.83 ms is manually injected into the control output task (0.83 ms is one third of the period 2.5 ms). It is selected since the original attack [1] on automotive injects a jitter of 10 ms to a victim task with 30ms period. The attack resulted in an average error of $0.0040 m/s^2$, which was still not significant enough to cause the vehicle to deviate from the original trajectory. The trajectories for the execution of the same mission with and without attacks are shown in Figure 4(d). From these experiments, we find that the existing design and deployment of the control software are incredibly resilient to computational jitter. Though the attacks are highly novel, it can be challenging to realize the exploitation in some systems.

B. Case Study II: Unmanned Aerial Vehicles

The second case study in the Butterfly Attack paper [1] focuses on Ardupilot [17] (an open-source UAV software) through two real-time tasks: `update_GPS` and `run_nav_updates`. In our efforts to realize the attack, we found that the actuation is sent via the `fast_loop()` task in Ardupilot 4.0. As a result, to be consistent with the original design of the attack, our experiment assumes that the victim task is `fast_loop()` instead. Our experiments were conducted using a self-assembled drone equipped with a Raspberry Pi 3B coupled with a Navio2 board, running Ardupilot 4.0. To emulate the GPS attack, the software is instrumented to drop three out of every four GPS messages.

Step 1 - Figure 5(a) shows the time intervals between job completions of the `update_GPS()` task with and without the attack. Without the attack, the average interval is 20.0051 ms, with 99.71% of the intervals lie within one variance (0.2227 ms). When the system is under attack, the average interval is 20.016 ms, with 99.88% of the intervals are also within one variance (0.6478 ms). The difference as compared to the previous study is likely due to the difference in the target software. In [1], the task is assumed to have an execution time of 4 ms with period of 5 ms, while in ArduPilot 4.0 Copter configuration, the average execution time of the task is 56.6231 us with a period of 20,000 us. The source code for `update_GPS()` and its period is detailed in Appendix B-A.

Step 2 - To understand the feasibility of the second step, we manually inject a 60 ms jitters into the task `update_GPS()`, since the jitters from step 1 was not large enough in our evaluation platform. The 60 ms was chosen because it is three times its period 20 ms, consistent with [1], where the jitter is 15ms in the entry task `update_GPS()`(three times of the period of 5ms). The resulting control command output intervals are shown in Figure 5(a). Under normal conditions, the average interval is 2.5080 ms with a variance of 0.4266 ms, and 99.95% of the intervals fall within one variance. In the attack scenario, the average interval is 2.5083 ms with a variance of 0.4449 ms, and 99.95% of the intervals also fall within one variance. From the experiment, we observe no significant difference in the control intervals, prompting a closer examination at the root cause. It turns out in ArduPilot 4.0, the output control command is executed prior to the task `update_GPS()` within the main control loop, therefore control has no direct dependency on `update_GPS()`. The source code for this design is detailed in Appendix B-B. The only exception is when the prior cycle of main control loop does not finish before the deadline. However, under such circumstances there are many other safety problems due to critical tasks missing their deadlines.

Step 3 - The translation from victim task jitter to physical state deviation depends highly on the robustness of the control algorithm. To gain a better understanding of the feasibility of this step upon a physical platform, a 1 ms jitter is manually injected into the control task *fast_loop* (since the second step did not result in substantial jitter). The 1 ms jitter was chosen in our target system, where *fast_loop* runs at 400Hz with a period of 2.5 ms, to be consistent with the original attack [1], where a 4 ms jitter is injected into a task with a period of 10 ms. Figure 5(c) presents the errors in the velocity controller, which is a vital controller for maintaining the stability of the drone. The average errors under normal conditions and during an attack are $0.0047 m/s$ with variance of $0.0000149 m/s$ and $0.0048 m/s$ with variance of $0.0000128 m/s$, respectively. Regarding the injected jitter, it causes an average error of $0.0049 m/s$ with variance of $0.0000134 m/s$. The trajectories for the execution of the same mission with and without attacks are shown in Figure 5(d).

VI. Context and Conclusions

The work reported in this paper can be looked upon as an illustrative example validating our **thesis** that *securing safety-critical real-time control loops from attack by malicious adversaries requires coordinated effort from domain experts in real-time scheduling, control theory, and systems security*. For us, this thesis arose out of our experiences with the Butterfly Attack [1], a proposed attack upon a safety-critical control loop that is conceptually very elegant. The idea of the Butterfly Attack was, to our knowledge, novel (and very interesting); this motivated us to investigate it closely with the objective of understanding its significance and its applicability, and in order to develop effective mitigations.

Amongst the aspects of our findings that we would like to particularly highlight is the relatively straightforward nature of the fix: simply modeling the system using the sporadic (rather than periodic) task model where appropriate during pre-runtime analysis is an adequate countermeasure against Butterfly Attacks. This leads us to suggest that the real-time scheduling community needs to do a better job of communicating our basic results to other research communities in order that they may use these results to build better (in this specific case, more secure) systems. The additional modifications to pre-runtime analysis (reported in Section III) that are inspired by scheduling theory, alongside the results reported in Section IV and Section V, also points to the need for real-time scheduling theory researchers to work in collaboration with systems builders, in order to facilitate the effective application, after appropriate adaptation, of results from real-time scheduling theory.

Perhaps somewhat ironically, our experiences reported here also provide evidence that similar coordinated effort from domain experts in real-time scheduling, control theory, and systems security is also required in order to carry out successful attacks – as we have reported in Section V, merely identifying a single vulnerability (in [1], this was the ability to move a control loop out of its stability region) is not in itself adequate to successfully carry out an actual physical attack.

Acknowledgements

We are grateful to the authors of [1], Petru Eles in particular, for feedback and for assistance in preparing the camera-ready version of this paper. This research was funded in part by the US National Science Foundation (Grants CPS-1932530, CNS-2141256, CNS-2229290, CNS-2038995, and CNS-2238635) and the Swedish Research Council grant 2018-04446.

References

[1] 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)*, 2019, pp. 93–106.
[2] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.
[3] A. Cervin, "Stability and worst-case performance analysis of sampled-data control systems with input and output jitter," in *2012 American Control Conference (ACC)*, 2012, pp. 3760–3765.
[4] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *Proceedings of the Real-Time Systems Symposium*. Tucson, AZ: IEEE Computer Society Press, December 2007, pp. 239–243.
[5] A. Burns and R. I. Davis, "A survey of research into mixed criticality systems," *ACM Comput. Surv.*, vol. 50, no. 6, pp. 82:1–82:37, Nov. 2017. [Online]. Available: http://doi.acm.org/10.1145/3131347
[6] S. Baruah and A. Burns, "Sustainable scheduling analysis," in *Proceedings of the IEEE Real-time Systems Symposium*. Rio de Janeiro: IEEE Computer Society Press, December 2006, pp. 159–168.
[7] A. Burns and S. Baruah, "Sustainability in real-time scheduling," *Journal of Computing Science and Engineering*, vol. 2, no. 1, pp. 74–97, 2008.
[8] T. Baker and S. Baruah, "Sustainable multiprocessor scheduling of sporadic task systems," in *Proceedings of the EuroMicro Conference on Real-Time Systems*. Dublin: IEEE Computer Society Press, July 2008.
[9] P. Ekberg and W. Yi, "Fixed-priority schedulability of sporadic tasks on uniprocessors is NP-hard," in *2017 IEEE Real-Time Systems Symposium, RTSS 2017, Paris, France, December 5-8, 2017*. IEEE Computer Society, 2017, pp. 139–146. [Online]. Available: https://doi.org/10.1109/RTSS.2017.00020
[10] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, 2nd ed. Springer, 2005.
[11] S. Arora and B. Barak, *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009.
[12] N. C. Audsley, "Optimal priority assignment and feasibility of static priority tasks with arbitrary start times," The University of York, England, Tech. Rep., 1991.
[13] ——, "Flexible scheduling in hard-real-time systems," Ph.D. dissertation, Department of Computer Science, University of York, 1993.
[14] "Emlid navio2 - autopilot hat for raspberry pi," https://ardupilot.org/copter/docs/common-navio2-overview.html, 2023.
[15] "Ardupilot rover," https://ardupilot.org/rover/, 2023.
[16] "Autoware auto gitlab," https://gitlab.com/autowarefoundation/autoware.auto, accessed: 2022-12-12.
[17] "Ardupilot - a trusted, versatile, and open source autopilot system," https://ardupilot.org/, 2022.
[18] J. P. Lehoczky, "Fixed priority scheduling of periodic tasks with arbitrary deadlines," in *IEEE Real-Time Systems Symposium*, Dec. 1990, pp. 201–209.

Appendix A
The Best-Case Response Time Problem

In this appendix, we consider the problem of establishing a lower bound on the best-case response-time (BCRT) of a task in a periodic task system (or equivalently, a sporadic task system when its synchronous arrival sequence is being scheduled) under FP-scheduling. Recall (Section II-B) that in this paper each task $\tau_i \in \mathbb{T}$ is characterized by a period h_i, and a lower bound c_i^b and upper bound c_i^w on its actual execution duration each time it is executed. The hardness of the BCRT problem that we will establish here does not stem from any complicated relationship between the c_i^b and c_i^w parameters; in the following we show that the BCRT problem is coNP-hard even if $c_i^b = c_i^w$ for all tasks.

We reproduce the definition of the BCRT Problem from Section III:

The Best-Case Response-Time (BCRT) Problem

INSTANCE: An FP-scheduled sporadic task system $\mathbb{T}$ (with best- and worst-case execution times specified), and a positive integer a.

QUESTION: Does each job of the lowest-priority task in $\mathbb{T}$ have a response time $\geq a$ in the schedule of its synchronous arrival sequence?

We will establish the coNP-hardness of the BCRT problem by relating it to the worst-case response-time (WCRT) problem. Determining FP-schedulability is equivalent to determining whether the WCRT R_i^w of each task is no larger than its period parameter h_i (if a relative deadline parameter $D_i \leq h_i$ is additionally specified for the task, then this requirement becomes $R_i^w \leq D_i$). We find it convenient in our derivation below to use the following utilization-restricted variant of the WCRT problem, which has itself been shown [9] to be NP-complete.

The Worst-Case Response-Time (WCRT) Problem

INSTANCE: An FP-scheduled sporadic task system $\mathbb{T}$ with $U(\mathbb{T}) \leq \ln 2$, and a positive integer a. (Here $U(\mathbb{T})$ denotes the sum $\sum_{\tau_i \in \mathbb{T}} (c_i^w / h_i)$ of the individual task utilizations.)

QUESTION: Does each job of the lowest-priority task in $\mathbb{T}$ have a response time $\leq a$?

We note that the key difference between the above two problem formulations is that we are asked if the given number a is an upper bound to the possible response times in the WCRT case, and a lower bound in the BCRT case. We will now define a simple reduction from the WCRT problem to the complement of the BCRT problem, thereby showing coNP-hardness for the BCRT problem.

We reduce from the WCRT problem to the BCRT problem by copying the task set $\mathbb{T}$ of the former problem to a task set $\mathbb{T}'$ for the new problem, but changing the period of the lowest-priority task τ_{low} in $\mathbb{T}'$ to equal the hyper-period,

$$T_{\text{low}} = H(\mathbb{T}),$$

(here $H(\mathbb{T})$ denotes the hyperperiod of task system $\mathbb{T}$) and assigning each task $\tau_i \in \mathbb{T}'$ a best-case execution time c_i^b that is equal to the worst-case execution time c_i^w of the corresponding task in the original task system $\mathbb{T}$.

The change to τ_{low}'s period effectively means that it will only release the first job in every hyper-period in $\mathbb{T}'$ compared to $\mathbb{T}$. It is well-known that if the first job in the hyper-period has a response-time $\leq T_{\text{low}}$, then that job has the maximum response time [18]. Since we have $U(\mathbb{T}) \leq \ln 2$, the response-time of the first job must be $\leq T_{\text{low}}$ by Liu and Layland's utilization bound [2], and so τ_{low}'s WCRT must be the same in $\mathbb{T}$ and $\mathbb{T}'$. But since τ_{low} only releases a single job per hyper-period in $\mathbb{T}'$, and since all tasks have $c_i^b = c_i^w$, it must also be the case that τ_{low}'s WCRT and BCRT are the same in $\mathbb{T}'$. In order to answer the WCRT problem for $\mathbb{T}$

> "Does each job of the lowest-priority task in $\mathbb{T}$ have a response time $\leq a$?"

we can simply answer the BCRT problem for $\mathbb{T}'$

> "Does each job of the lowest-priority task in $\mathbb{T}'$ have a response time $\geq a+1$?"

and negate the answer. It follows that the BCRT problem is coNP-hard.

As a corollary we can conclude that the following problem of bounding the response time within an interval is both NP-hard and coNP-hard, and is therefore unlikely to be contained in the first level of the polynomial hierarchy.

The Response-Time Jitter Problem

INSTANCE: An FP-scheduled sporadic task system $\mathbb{T}$ (with best- and worst-case execution times specified), and positive integers a, b.

QUESTION: Does each job of the lowest-priority task in $\mathbb{T}$ have a response time in interval $[a, b]$ in the schedule of its synchronous arrival sequence?

APPENDIX B
ARDUCOPTER 4.0.0 SOURCE CODE

In this section, we showcase simplified source code snippets extracted from Ardupilot, specifically from version Copter-4.0.0[6].

A. *Task Model of* `update_GPS`

Figure 6(a) displays the definition for the period of the `Update_GPS` task, which is set with a frequency of 50Hz, equivalent to a period of 20,000 us.

Meanwhile, Figure 6(b) provides its execution logic. The function `Update_GPS` is initiated with a conditional check. If this condition is not fulfilled, the main portion of the code remains unexecuted. This check determines whether a fresh message has arrived and if the status of this received signal matches previously GPS readings. If yes, it will skip the execution of function `camera.update`, whose workload is shown in Figure 6(c). Therefore, should an attacker disrupt the GPS signal, the computational duration for the complete `update_GPS` task will see a reduction. From our profiling

[6]https://github.com/ArduPilot/ardupilot/tree/Copter-4.0.0

```
/*
  scheduler table for fast CPUs - all regular tasks apart from the fast_loop()
  should be listed here, along with how often they should be called (in hz)
  and the maximum time they are expected to take (in microseconds)
 */
const AP_Scheduler::Task Copter::scheduler_tasks[] = {
    SCHED_TASK(rc_loop,              100,    130),
    SCHED_TASK(throttle_loop,         50,     75),
    SCHED_TASK(update_GPS,            50,    200),
```

(a) Period of `update_GPS`.

```
// called at 50hz
void Copter::update_GPS(void)
{
    static uint32_t last_gps_reading[GPS_MAX_INSTANCES];   // time of last gps
message
    bool gps_updated = false;

    gps.update();

    // log after every gps message
    for (uint8_t i=0; i<gps.num_sensors(); i++) {
        if (gps.last_message_time_ms(i) != last_gps_reading[i]) {
            last_gps_reading[i] = gps.last_message_time_ms(i);
            gps_updated = true;
            break;
        }
    }

    if (gps_updated) {
#if CAMERA == ENABLED
        camera.update();
#endif
    }
}
```

(b) Workload of `update_GPS`.

```
// Camera Update - take a picture
void AP_Camera::update()
{
    // . . .
    // take a local picture:
    trigger_pic();

    // tell all of our components to take a picture:
    mavlink_command_long_t cmd_msg {};
    cmd_msg.command = MAV_CMD_DO_DIGICAM_CONTROL;
    cmd_msg.param5 = 1;

    // forward to all components
    GCS_MAVLINK::send_to_components(MAVLINK_MSG_ID_COMMAND_LONG,
(char*)&cmd_msg, sizeof(cmd_msg));
}
```

(c) Workload of `camera.update`.

Fig. 6. Period and workload of task `update_GPS`

results, this reduction are around 56.6231 us with variance of 62.1848 us.

B. Scheduler in ArduCopter

```
void AP_Scheduler::loop()
{
    // Execute the fast loop
    // ---------------------
    _fastloop_fn();
   if (now - sample_time_us < loop_us) {
        // get remaining time available for this loop
        time_available = loop_us - (now - sample_time_us);
    }
    // add in extra loop time determined by not achieving scheduler tasks
    time_available += extra_loop_us;

    // run other tasks
    run(time_available);
    // …
}
```

Fig. 7. Scheduling logic of ArduCopter.

Figure 7 illustrates the task scheduling mechanism within Ardupilot. The primary control loop is the first to be scheduled during each scheduling cycle (emphasized in green). If there is extra available time, the scheduler will run other tasks (highlighted in yellow), where `update_GPS` is one of the ‘other tasks’.