CYBER-PHYSICAL ANALYSIS AND HARDENING OF ROBOTIC AERIAL VEHICLE CONTROLLERS

by

Taegyu Kim

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

School of Electrical and Computer Engineering

West Lafayette, Indiana

May 2021

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Dongyan Xu, Chair

Department of Computer Science

Dr. Dave (Jing) Tian

Department of Computer Science

Dr. Xiaojun Lin

Department of Electrical and Computer Engineering

Dr. Xiangyu Zhang

Department of Computer Science

Dr. Vijay Raghunathan

Department of Electrical and Computer Engineering

Approved by:

Dr. Dimitrios Peroulis

To my parents and my sister

ACKNOWLEDGMENTS

First of all, I would like to thank my advisor, Professor Dongyan Xu. His guidance played a significant role in establishing me as a researcher. When I was a first-year Ph.D. student who does not much knowledge of security research, he believed that I would grow as a mature researcher. Hence, he has taught how to initiate research ideas based on small observations from day-to-day discussions, how to lead a novel project, and how to deliver research ideas to readers and audiences. As a result of his sincere advice and guidance, I successfully published meaningful papers which are recognized by top-tier security researchers.

I would also like to express my great thanks to Professor Dave (Jing) Tian. Professor Tian has broadened my view on systems and presentation with his experience and insights as my co-advisor. Furthermore, he provided his brilliant insights and feedback on my research ideas and thoroughly edited my manuscripts with late-night teamwork. Finally, I learned how to make my research ideas more persuasive from day-to-day discussions. Thanks to his advice, I could learn how to research system security projects.

I have been fortunate to work with many excellent senior researchers during my Ph.D. study. To solidify this work, I have gotten a lot of help from Rohit Bhatia, Antonio Bianchi, Berkay Celik, Xinyan Deng, Sriharsha Etigowni, Fan Fei, James Goppert, Chung Hwan Kim, Kyungtae Kim, Yonghwi Kwon, Vireshwar Kumar, Byoungyoung Lee, Yuhong Nan, Junghwan Rhee, Brendan Saltaformaggio, Zhan Tu, and Xiangyu Zhang. They shared their research experience and insights. Also, they helped me do experiments and edit my manuscripts to strengthen my work. Thank you very much for spending time and working with me.

I have worked with many colleagues and junior researchers during my Ph.D. study. I have gotten a lot of help from Abdulellah Alsaheel, Aolin Ding, Yuseok Jeon, Arslan Khan, Hyungsub Kim, Hui Peng, and Gregory Walkup. Thanks to spending time with them, I could enjoy my Ph.D. study. Furthermore, I fortunately worked with Jizhou Chen, Sungwoo Kim, William Wang, and Gisu Yeo. Thanks to them, I could get help for my research projects. Because I could enjoy mentoring them, I was sure that I want to join the academic institute. I will miss the time with everyone for research, and I hope they will enjoy their future Ph.D. study.

Many people helped me on my journey to completing this work. My parents, Byungok Kim and Myungsuk Yoon, have supported me with their hearts without any doubt of completing my Ph.D. study. Thanks to their unconditional support, I successfully completed my Ph.D. journey. I would like to thank my sister, Jinah Kim. At the beginning of my Ph.D. study, she gave me a lot of help to settle at Purdue. I would also like to thank my friends of Jinwoong Ju, Hyungjong Koh, Sungmin Lee, Yeonho Lee, Jaekyun Nam, Jihoon Kim, Myungseok Kim, Woohyun Kim, and Kyuhoon Sung as my closest friends while spending a happy and memorable time with me.

Finally, I would like to thank my plan of study committee members for participating in my preliminary and final exams. I deeply appreciate their reasonable criticism and insightful suggestions for my dissertation.

This work was supported in part by ONR under Grants N00014-17-1-2045 and N00014-20-1-2128. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the ONR.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Robotic aerial vehicles (RAVs) have been increasingly deployed in various areas (e.g., commercial, military, scientific, and entertainment). However, RAVs' security and safety issues could not only arise from either of the "cyber" domain (e.g., control software) and "physical" domain (e.g., vehicle control model) but also stem in their interplay. Unfortunately, existing work had focused mainly on either the "cyber-centric" or "control-centric" approaches. However, such a single-domain focus could overlook the security threats caused by the interplay between the cyber and physical domains.

In this thesis, we present cyber-physical analysis and hardening to secure RAV controllers. Through a combination of program analysis and vehicle control modeling, we first developed novel techniques to (1) connect both cyber and physical domains and then (2) analyze individual domains and their interplay. Specifically, we describe how to detect bugs after RAV accidents using provenance (Mayday), how to proactively find bugs using fuzzing (RVFuzzer), and how to patch vulnerable firmware using binary patching (DisPatch). As a result, we have found 91 new bugs in modern RAV control programs, and their developers confirmed 32 cases and patch 11 cases.

1. INTRODUCTION

Robotic aerial vehicles (RAVs), such as commodity drones, are cyber-physical systems which are widely deployed for autonomous transportation. Each RAV is typically equipped with a computing board with control hardware (e.g., micro-controller) and software (e.g., real-time control program). The on-board control program continuously senses the vehicle's physical state (e.g., position and velocity) and actuates the motors to control the vehicle's movement and accomplish a given mission. RAVs have been being increasingly utilized for various applications in commercial, industrial, entertainment, and law enforcement domains. For instance, logistics companies (e.g., USPS, DHL, and Amazon) have introduced drone delivery services to meet the rapidly growing demand in e-commerce [1]–[4].

With their increasing adoption in real-world applications, RAVs are facing threats of cyber and cyber-physical attacks that exploit their wide attack surfaces. More specifically, an RAV's attack surface spans multiple aspects, such as (1) physical vulnerabilities of its sensors that enable external sensor spoofing attacks [5]–[7]; (2) traditional "syntactic" bugs in its control program (e.g., memory corruption bugs) that enable remote or trojaned exploits [8]–[11]; and (3) cyber-physical bugs in its control program that enable attacks via remote control commands. In the prior art, while there have been extensive research efforts in defending RAVs against attacks that exploit (1) and (2) [10], [12]–[17], attacks exploiting (3) have not received sufficient attention. Hence, the RAV's attack surface with respect to (1) and (2) is expected to get smaller, which may prompt attackers to increasingly look at the cyber-physical bugs for new exploits.

In this thesis, we secure RAV control programs by focusing on an important type of cyber-physical bugs. A cyber-physical bug involves an incorrect or missing validity check on a control parameter-change or mission-change input in the *"cyber"* domain. Such input is provided to the control program via a remote control command, which could trigger RAV controller malfunction. Consequently, such controller malfunction leads to physical impacts on the vehicle (i.e., in the *"physical"* domain), such as mission disruption, vehicle instability, or even vehicle damage/crash. To prevent such destructive interplay between the *"cyber"* and *"physical"* components caused by cyber-physical bugs, we need to have dedicated frameworks to discover and patch cyber-physical bugs.

However, such security analysis and hardening are challenging because cyber-physical bugs are largely orthogonal to the traditional "syntactic" bugs (e.g., buffer overflow and use-after-free bugs). Hence, they cannot be detected and hardened by existing software techniques. The fundamental reason why existing techniques are not applicable is that they perform security analysis of cyber and physical domains independently. Such "independent" analysis may miss critical hints/traces in finding cyber-physical bugs because their impacts and their root causes (i.e., cyber-physical bugs) appear in the cyber and physical domains, respectively. Therefore, there is a pressing need to efficiently and accurately find and eliminate cyber-physical bugs by simultaneously considering various components in both domains.

To overcome the aforementioned challenges, we design the cross-domain "cyber-physical" analysis and hardening frameworks based on the following three key techniques. First, we developed the *control instability detector* to detect controller anomaly that can be developed into destructive physical impacts (e.g., vehicle crashes) caused by cyber-physical bugs. Next, we developed the *mapping graph* to bridge the *control model* abstracting vehicle dynamics (e.g., physical model and motor power) in the "physical" domain and *control program* (running control algorithms) in the "cyber" domain. Finally, we developed the *cross-domain analyzer* to identify key controller components and cyber-physical bugs. Those three techniques enable to track from the detected controller anomaly in the "physical" domain, through the mapping graph, to cyber-physical bugs and related controller components in the "cyber" domain.

Based on our aforementioned key techniques, we propose the following three cyber-physical analysis and hardening systems: (1) Mayday [18] for attack investigation for cyber-physical bugs, (2) RVFuzzer [19] for cyber-physical bug discovery, and (3) DisPatch [20] for controller-semantic-aware decompiling and patching the controller components in the binary control program. We present the details of these systems as follows.

- Mayday is an attack investigation framework to reactively locate the root cause in the program domain. For that, there are two key challenges in evidence and methodology: (1) Current RV' flight log only records high-level vehicle control states and events, without recording control program execution; and (2) The capability of "connecting the dots" – from controller anomaly to program variable corruption to program bug location – is lacking. To address these challenges, MAYDAY maps a control model to a control program (source code) to enable (1) in-flight logging of control program execution and (2) traceback of a root cause based on control and program-level logs. We have applied MAYDAY to ArduPilot to investigate ten RAV accidents caused by real ArduPilot bugs. We also found four recently patched bugs still vulnerable, and all of them have been confirmed by the ArduPilot team. This demonstrates that MAYDAY is able to pinpoint the root cause of these accidents within the program with high accuracy and minimum run-time and storage overhead.

- RVFUZZER is a fuzzing framework to proactively find cyber-physical bugs in RAV control programs through control-guided input mutation. Furthermore, RVFUZZER runs on a physical simulator without requiring source code. Hence, RVFUZZER overcomes the limitations of MAYDAY which can reactively locate cyber-physical bugs and require source code and physical RAV accidents. To realize such functionality, RVFUZZER is designed to satisfy the two design requirements as follows: *(1) how to detect the cyber-physical bugs, and (2) how to improve the fuzzing throughput.* To satisfy design requirement (1), we observed that exploiting cyber-physical bugs leads to controller anomaly at the end. Therefore, I developed a control instability detector based on control model properties. To satisfy design requirement (1), we created a new fuzzing input mutation scheme to reduce large fuzzing input space (e.g., ArduPilot has hundreds of tunable control parameters). Specifically, this newly developed scheme focuses on mutating control parameters based on control theoretical properties. In our evaluation of RVFUZZER on two popular RAV control programs, a total of 89 cyber-physical bugs are found, with 87 of them being zero-day bugs. So far, 28 of our found cyber-physical bugs have been confirmed, and 11 of them have been patched by the RAV software developers.

- DisPatch is a controller-semantic-aware disassembling and hardening system for the binary control program. The RAV software developers design RAV software to support hundreds of physical RAV models (e.g., weight) with various hardware components (e.g., sensors and actuators). Since each physical model is vulnerable to specific cyber-physical bugs, we must apply *model-specific* patches dedicated to the specific physical model. However, for patching such specific cyber-physical bugs, we need to first semantically identify the location of the target controller in the binary code without the knowledge of control-semantics. To solve this problem, we propose DisPatch which incorporates model-specific patches to the given firmware and physical control model. First, DisPatch requires two inputs: (1) locations of commonly used controller codes which match with the mathematical model [21], and (2) model-specific controller structures to identify controllers' inter-dependencies. Then, DisPatch discovers the semantic roles of respective controllers by identifying dependencies between controllers and matching them with the target theoretical control model. We expect that DisPatch will provide a mitigation framework against cyber-physical bugs including those that are identified by RVFuzzer and Mayday.

We structure the rest of this thesis as follows: Section 2 illustrates the methods employed in Mayday to perform attack investigation caused by cyber-physical bugs; Section 3 describes the techniques utilized in RVFuzzer to discover cyber-physical bugs; Section 4 illustrates the RAV binary firmware decompiling and patching techniques to protect against the attacks caused by cyber-physical bugs; Section 5 summarizes the related work; Section 6 describes our future research agenda; Section 7 concludes this dissertation.

2. FROM CONTROL MODEL TO PROGRAM: INVESTIGATING ROBOTIC AERIAL VEHICLE ACCIDENTS WITH MAYDAY

Our initial research is started to investigate why and how RAV physically crashes in order to prevent attackers from exploiting the same cyber-physical bugs in the future. As described in my previous work (BlueBox [13]), we have observed that *cyber-physical bugs eventually lead to controller anomaly; hence, they can be detected with the help of control theory.* Researchers in the physical domain developed a "physical-centric" root cause investigation tool with the logging of a vehicle's control states. However, "physical-centric" analysis tools cannot find a physical crash root cause hidden in the control program. This is because control-level logs do not contain any information to inspect control program execution in detail. To the best of my knowledge, no existing work can trace an RAV crash's root cause from "physical" to "cyber" domain. In this chapter, we present Mayday [18] which supports the cross-domain investigation by connecting the dots from controller anomaly to program variable corruption to program bug locations.

2.1 Background and Models

RAV Control Model. Mayday is driven by the RAV control model, which encompasses (1) vehicle dynamics, (2) controller organization, and (3) control algorithm. For vehicle dynamics, an RAV stabilizes movements along the six degrees of freedom (6DoF) such as the x, y, z-axes and the rotation around them, namely *roll*, *pitch*, and *yaw* as described in Figure 2.1. Each of the 6DoF is controlled by one *cascading* controller, with dependencies shown in Figure 2.2.

Inside each 6DoF controller, a cascade of *primitive controllers* controls the position, velocity, and acceleration of that "degree", respectively. The control variables of these primitive controllers have dependencies induced by physical laws. Figure 2.3 shows such dependencies using the x-axis controller as an example. For the x-axis position controller ($c_1(t)$, left-most), $x_x(t)$ is the *vehicle state* (i.e., position). $r_x(t)$ is the *reference* which indicates the desired

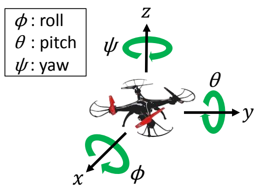
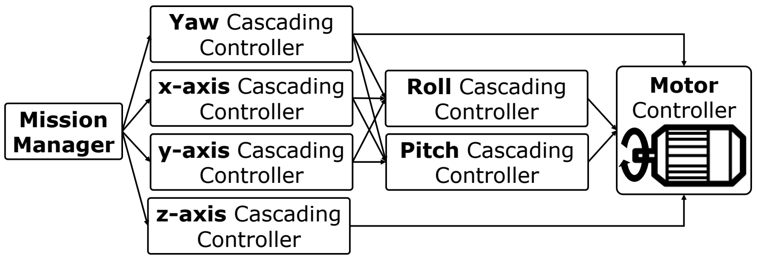


Figure 2.1. An RAV's six degrees of freedom (6DoF).



Mission Manager
Yaw Cascading Controller
x-axis Cascading Controller
y-axis Cascading Controller
z-axis Cascading Controller
Roll Cascading Controller
Pitch Cascading Controller
Motor Controller

Figure 2.2. Dependencies of an RAV's Six degrees of freedom (6DoF) cascading controllers.

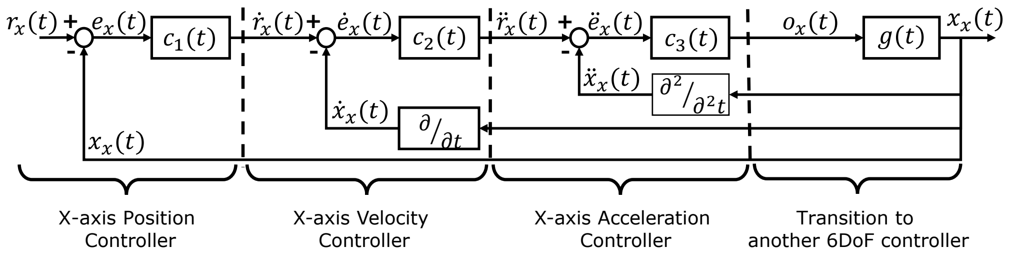


Figure 2.3. Primitive controllers in x-axis cascading controller.

position. $\mathrm{e}_x(t) = r_x(t) - x_x(t)$ is the *error*, namely difference between the state and reference. Intuitively, the goal of the controller is to minimize $\mathrm{e}_x(t)$.

Similarly, the velocity and acceleration primitive controllers have their own sets of control variables: $\dot{x}_x(t)$, $\dot{r}_x(t)$, $\dot{\mathrm{e}}_x(t)$ for x-axis velocity; and $\ddot{x}_x(t)$, $\ddot{r}_x(t)$, $\ddot{\mathrm{e}}_x(t)$ for acceleration (the "dot" symbol denotes differentiation). The three primitive controllers work in a cascade: the output (reference) of one controller becomes the input of its immediate downstream controller. Each controller also accepts other inputs, such as flight mission and control parameters. The output of a cascading controller (e.g., $o_x(t)$) can be either a motor throttle value or a reference input for another 6DoF controller (e.g., from the x-axis controller to the roll angle controller).

RAV Control Program. The RAV control program implements the RAV control model. It accepts two types of input: (1) sensor data that measure vehicle states and (2) operator commands from ground control (GCS). GCS commands are typically issued to set/reset flight missions (e.g., destination and velocity) and control parameters (e.g., control gain). The control program runs periodically to execute the multiple controllers. For auditing and troubleshooting, most RAV control programs record controller states (e.g., vehicle state and reference) and events (e.g., sensor and GCS input) in each control loop iteration and store them in on-board persistent storage.

Trust Model and Assumptions. MAYDAY is subject to the following assumptions: (1) We assume the soundness of the underlying RAV control model. (2) We assume that the RAV control program already generates high-level control log, which at least includes each primitive controller's reference, state and input. This is confirmed by popular RAV control programs ArduPilot [22], PX4 [23] and Paparazzi [24]. (3) We assume the integrity of logs and log generation logic in the control program, which can be enforced by existing code and data integrity techniques [25]–[27]. After a crash, we assume that the logs are fully recoverable from the vehicle's "black box". (4) We assume the control flow integrity of control program execution. Hence traditional program vulnerabilities/exploits, such as buffer overflow, memory corruption, and return-oriented programming, are outside the scope of MAYDAY. There exists a wide range of software security techniques to defend against such attacks [10], [28]–[30].

Soundness of Control Model. To justify Assumption (1) of the trust model, we show that the underlying RAV control model adopted by ArduPilot is theoretically sound. For the model's *vehicle dynamics*, prior work [13] has analytically proved its correctness by modeling a standard rigid body system using Newton-Euler equations. For the model's *control algorithm* and *controller organization*, every primitive controller (e.g., those in Figure 2.3) instantiates the classic PID (proportional-integral-derivative) algorithm; whereas all the controllers are organized in a dependency graph (CVDG, to be presented in Section 2.4.1), which reflects the classic RAV controller organization for controlling the vehicle's 6DoF.

Based on the sound control model elements, the model's stability has been proved in prior work [31]. Furthermore, the control model – by design – tolerates vehicle dynamics changes (e.g., payload change) and disturbances (e.g., strong wind) to a *bounded* extent. We note that MAYDAY investigates accidents/attacks when the vehicle is operating *within* such bounds; and the triggering of the cyber-physical bug will make an *originally sound* control model unsound, by corrupting its control/mission parameter(s), leading to instability of the system. Finally, theoretical soundness of the RAV control model is also testified to by its wide adoption by RAV vendors such as 3D Robotics, jDrone, and AgEagle for millions of robotic vehicles [32].

Threat and Safety Model. MAYDAY addresses safety and security threats faced by RAVs, with a focus on finding cyber-physical bugs in RAV control programs after accidents. These accidents may be caused by either safety issues (e.g., buggy control code execution or operator errors) or attacks (e.g., deliberate *negligence* or *exploitation* by a malicious insider). We assume that attackers know the existence of a cyber-physical bug and its triggering condition. Then, an attacker may (1) continue to launch flight missions under the bug-triggering condition (e.g., strong wind) or (2) adjust vehicle control/mission parameters to create the bug-triggering condition (demonstrated in [19]). Action (1) requires the operator to simply "do nothing"; whereas action (2) will only leave a minimum bug-triggering footprint which could gradually corrupt controller states over a long period of time (Section 2.2). Such small footprint and long "trigger-to-impact" time gap make investigation harder. Furthermore, attacks exploiting cyber-physical bugs do not require code injection, sensor/GPS spoofing, or blatantly self-sabotaging commands. As such, the security threat posed by cyber-physical bugs is real to RAV operations and "exploit-worthy" to adversaries. All accident cases in our evaluation (Section 2.7) can happen in either accidental (i.e., safety) or malicious (e.g., security) context, reflecting the broad applicability of MAYDAY for RAV safety and security.

Meanwhile, accidents caused by either *physical* failures/attacks or generic software bugs are out of scope, as they have been addressed by existing efforts. For example, built-in logs can provide information for investigating either suspicious operator commands or physical attacks/accidents [33]–[35], without cross-layer (i.e., from control model to program) anal-

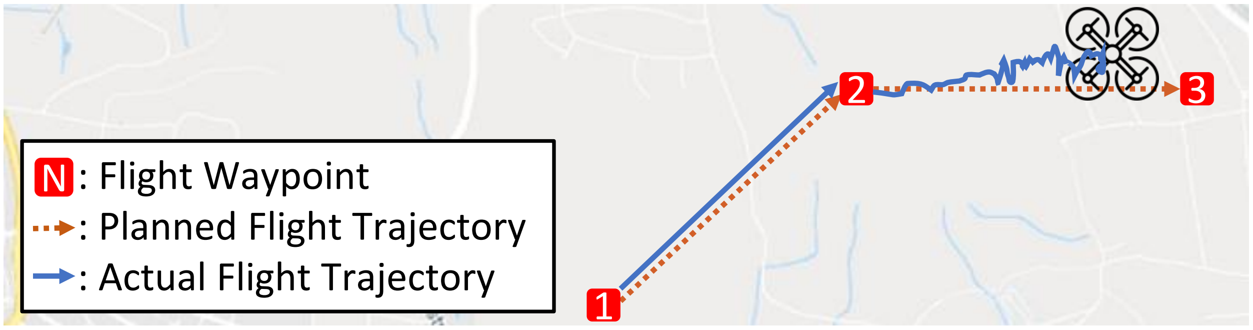


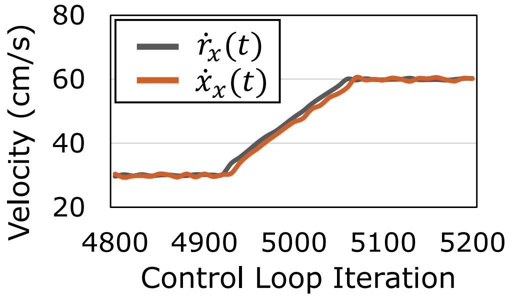
Figure 2.4. Motivating example flight. An RAV first flies to the north east with 60 cm/s (only in east, 30 cm/s) and then flies to east with 60 cm/s speed.

ysis (Section 2.8); and there has been a large body of solutions targeting generic software vulnerabilities [36]–[43].

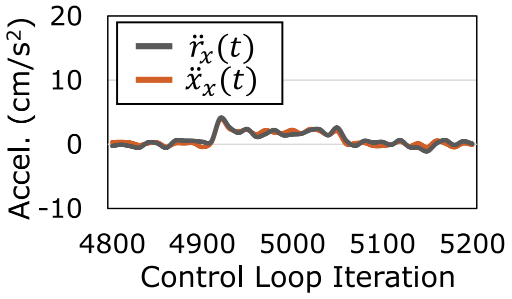
Finally, we note that there are multiple possible root causes to check after an RAV accident (e.g., software bugs, mechanical issues, and human operator factors). Mayday, which specializes in cyber-physical bugs, is only *one of* multiple investigation tools (e.g., those for physical attacks) to enable a thorough, multi-aspect investigation.

2.2 Motivating Example

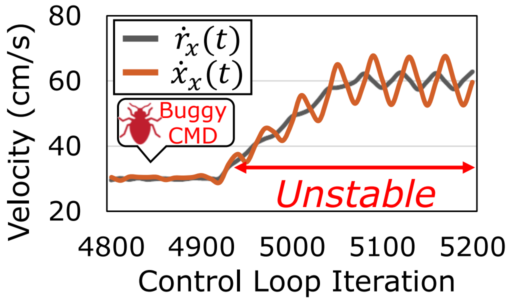
Modern control programs are robust systems that operate while addressing and minimizing the impact of not only various physical non-deterministic factors (e.g., inertia and noise) but also controller anomaly and security attacks [44]–[46]. However, we have found that such robustness is not enough to tackle all safety and security issues. Specifically, combined impacts of (i) operational inputs (e.g., mission, parameter changes) with (ii) particular altered physical conditions may go beyond the protection capability of a control system, which is an implication of a cyber-physical bug. As a result, such impact starts to appear in a control variable of an exploited controller and will be propagated to its dependent controllers and can be *signified over the multiple control loop iterations*. To illustrate this, we introduce the following intuitive motivating accident case (more cases are discussed in Section 2.7) *only with* high-level control logs recorded by a built-in flight recorder.



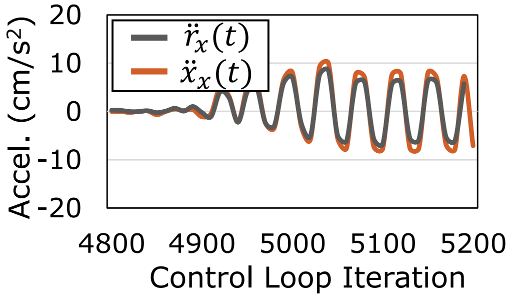
(a) Velocity without a bug.



(b) Acceleration without a bug.



(c) Velocity with a bug.



(d) Acceleration with a bug.

Figure 2.5. Controller states with and without the x-axis velocity parameter manipulation. The control loop iterates at a 10 Hz interval.

In this example, we assume that our target RAV loads an item to deliver (as performed by real RAVs [3], [4], [47]) and flies to the north east with 60 cm/s (only in east, 30 cm/s) as described in Figure 2.4. At Iteration 4,850, the RAV operator increases Parameter P of x-axis velocity controller to make up for the weight gain. In the next 80 iterations of the control loop, the RAV continues to operate normally (i.e., the x-axis controller maintains a stable state). At a scheduled turn (i.e., flying east in Figure 2.4), the RAV is supposed to drastically decrease its x-axis velocity and to exhibit a behavior similar to that of the velocity and acceleration references depicted in Figure 2.5a and Figure 2.5b, respectively. However, at the junction, the changed parameter P unexpectedly leads to a corrupt state; the x-axis velocity started showing digression (Figure 2.5c) and generating a corrupt x-axis acceleration reference. Consequently, the RAV completely failed to stabilize, ultimately resulting in a crash due to intensified digression over the multiple control loop iterations. We note that our example case is realistic because this accident can be triggered via a remote operational interface (e.g., MAVLink [48]).

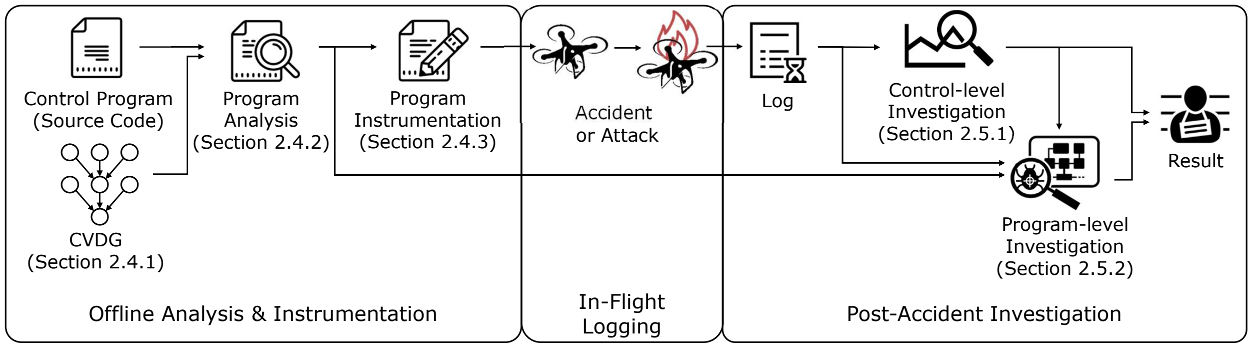


Figure 2.6. MAYDAY Framework.

Unfortunately, to answer "why did my drone crash" in this case, the existing flight status logging is not sufficient for root cause analysis. Unlike control-level investigation based on built-in flight control data logging, there is no evidence available for program-level investigation. While investigators may be able to identify a malicious command by cross-checking the command logs recorded by the GCS and by the on-board logging function, such a method cannot investigate (1) accidents caused by malicious or vulnerable commands that are indeed issued from the GCS (e.g., by an insider threat) or (2) accidents not triggered by external commands (e.g., divide-by-zero). Most importantly, such a method cannot pinpoint the root cause of the accident. In other words, observing the RAV controller anomaly does not reveal *what is wrong inside the control program*. We need to bridge the semantic gap between the safety/security impacts in the control (physical) domain and the root causes in the program (cyber) domain.

2.3 Mayday Framework

MAYDAY spans different phases of an RAV's life cycle, shown in Figure 2.6. In the offline phase, MAYDAY defines a formal description of the RAV control model, and uses it to enable CVDG-guided program-level logging during the control program execution via automatic instrumentation (Section 2.4). Then the RAV goes back into service with the instrumented control program, which will generate both control- and program-level logs during flights. In the case of an accident or attack, MAYDAY retrieves the logs and performs a two-stage forensic analysis, including control- and program-level investigations (Section 2.5). The

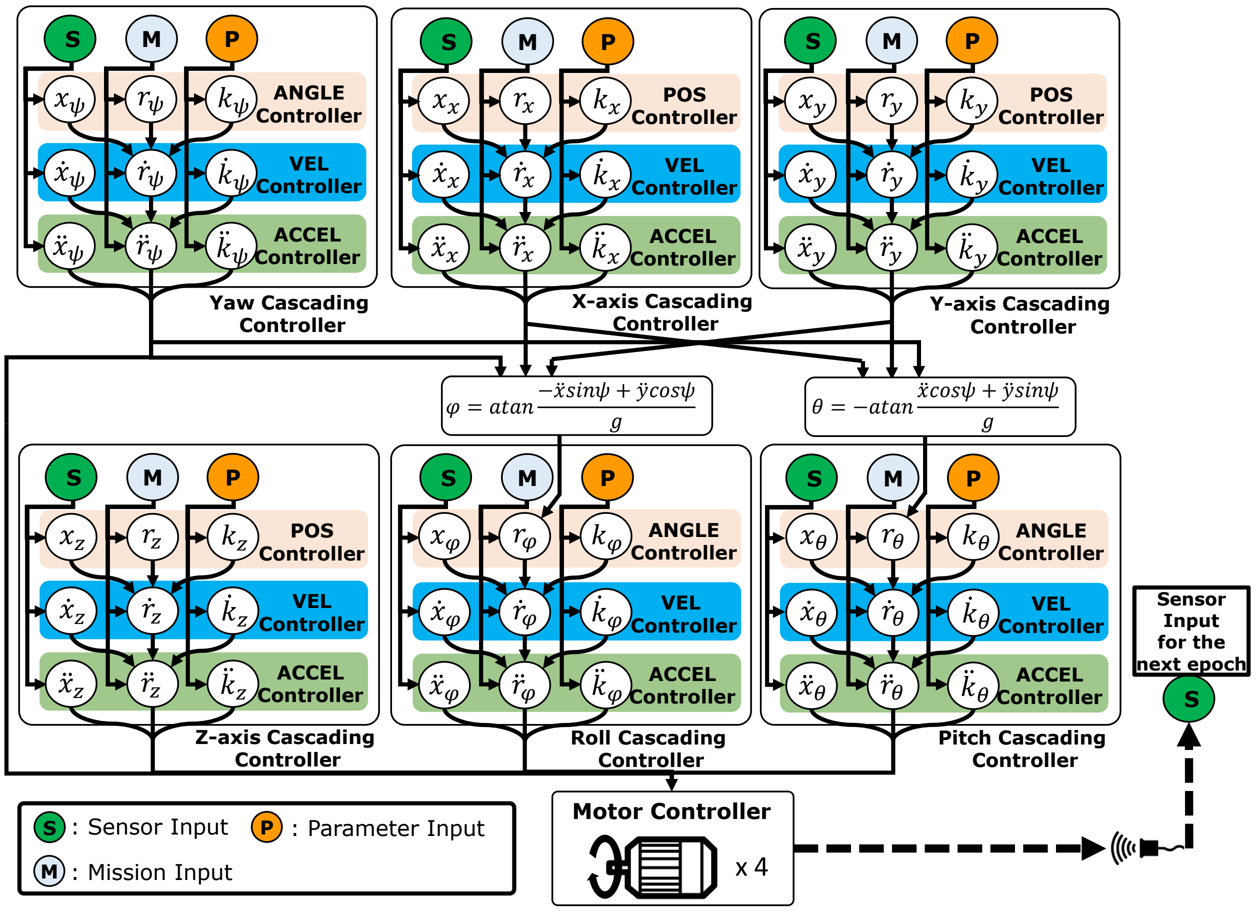

S
M
P
ANGLE Controller
VEL Controller
ACCEL Controller
Yaw Cascading Controller
POS Controller
X-axis Cascading Controller
Y-axis Cascading Controller
$\varphi = atan\frac{-\ddot{x}sin\psi + \ddot{y}cos\psi}{g}$
$\theta = -atan\frac{\ddot{x}cos\psi + \ddot{y}sin\psi}{g}$
Z-axis Cascading Controller
Roll Cascading Controller
Pitch Cascading Controller
Sensor Input for the next epoch
S : Sensor Input
P : Parameter Input
M : Mission Input
Motor Controller
x 4

Figure 2.7. Control Variable Dependency Graph (CVDG).

investigations will lead to the localization of the cyber-physical bug in the control program – the root cause of the crash.

2.4 Control-Guided Control Program Analysis and Instrumentation

This offline phase of MAYDAY formalizes a generic RAV control model using a Control Variable Dependency Graph (CVDG) (Section 2.4.1), which will guide the analysis (Section 2.4.2) and instrumentation (Section 2.4.3) of the control program, in preparation for the run-time program execution logging and the post-accident investigation (Section 2.5).

2.4.1 Control Variable Dependency Graph

MAYDAY is guided by the RAV's control model, with dependencies among controllers and control variables and execution paths. To capture such dependencies, we define the Control Variable Dependency Graph (CVDG) which acts as a *mapping graph* between a control model to a control program. Figure 2.7 shows a generic CVDG that applies to a wide range of RAVs, such as rigid-body trirotors, quadrotors, and hexarotors. The CVDG captures generic dependencies among the 6DoF controllers without assuming any specific control algorithm. Inside each controller, there is a cascade of three *primitive controllers* that control the position, velocity, and acceleration for that DoF, respectively. Each node in the CVDG represents a control variable or a controller input. Each control variable represents a vehicle *state* (e.g., x_x, $\dot{x}_x$, or $\ddot{x}_x$), *reference* (e.g., r_x, $\dot{r}_x$, or $\ddot{r}_x$), or control parameters (e.g., k_x, $\dot{k}_x$ or $\ddot{k}_x$). The controller accepts three types of input S, M, and P: S represents inputs from various sensors, which will become vehicle state after pre-processing (e.g., filtering); M and P represent mission plan and control parameter inputs, respectively. Each directed edge in the CVDG indicates a dependency between its two nodes. For example, the edge from $\dot{r}_x$ to $\ddot{r}_x$ in the x-axis controller indicates that $\ddot{r}_x$ depends on $\dot{r}_x$.

Inter-Controller Relation. We also define the "parent-child" relation between two controllers with edge(s) between them. More specifically, if primitive controller C's reference is the output of controller C, then C and C have a parent-child relation. Within a 6DoF cascading controller, the state of a child controller (e.g., x-axis acceleration) is the *derivative* of its parent controller (e.g., x-axis velocity). The relation between 6DoF controllers is more complicated. For example, the roll angle (ϕ) controller has three parent controllers (i.e., yaw (ψ), x, and y acceleration controllers). Mathematically, the input of the roll angle controller is determined by the outputs of its three parent controllers as: $\phi = atan((-\ddot{x}sin(\psi) + \ddot{y}cos(\psi))/g)$ (Figure 2.7, g is the standard gravity).

2.4.2 Mapping CVDG to Control Program

Mapping CVDG Nodes to Program Variables. We now establish a concrete mapping between the CVDG and the control program that implements it. First, we map the CVDG nodes (control variables) to the corresponding control program variables, which are either global or heap-allocated. For most CVDG control variables, the control program's existing logging functions directly access and log the corresponding program variables. For certain CVDG variables, we need to look deeper. For example, the x-, y-, and z-axis velocity states are retrieved via function calls. To handle such cases, we perform backtracking on LLVM bitcodes (i.e., the intermediate representation (IR) of the Low Level Virtual Machine (LLVM)): Starting from the logged (local) variable in a logging function, we backtrack to variables whose values are passed (without processing) to the logged variable. Among those, we select the first *non-local* variable (e.g., a class member variable) as the corresponding program variable.

Mapping CVDG Edges to Program Code. Next MAYDAY analyzes the control program to map each CVDG edge to the portion of control program codes that implement the data flow between the two nodes (variables) on the edge. For each edge, MAYDAY conservatively identifies all possible program paths that induce data flows between the source node and sink node.

Our analysis is performed by Algorithm 1 at LLVM bitcode level. It is inter-procedural and considers pointer aliases of the control variables as well as other intermediate variables for completeness. It first performs a path-insensitive and flow-sensitive points-to analysis [49] to identify all aliases of the control variables (Line 2-3). For each alias identified, the algorithm performs backward slicing [50] to identify the program code that may influence the value of the control variable (Line 4, 10-23). As a result, each slice contains all the instructions that directly read or write the control variable and those that indirectly affect its value through some intermediate variables. Since the intermediate variables may have aliases not covered in the previous steps, Algorithm 1 recursively performs both points-to analysis and backward slicing on those variables to identify additional instructions that may affect the value of the control variable (Line 16-22). As new intermediate variables may be found in the identified

Algorithm 1 Mapping CVDG edges to program code.

```
Input: Control variable set in the CVDG ($CV$)
Output: Mapping control variables to backward sliced instructions ($M$)
1: Initialize $M$                                          ▷ Our algorithm entry point
2: for $cv_i \in CV$ do                                     ▷ Backward slicing for each $CV$
3:     $PV \leftarrow$ POINTS-TOANALYSIS($cv_i$)
4:     $S \leftarrow$ BACKWARDSLICINGVARSET($PV$)              ▷ Backward slicing for aliases of $cv_i$
5:     $N \leftarrow$ GETAFFECTINGNODES($S$)                   ▷ Get CVDG nodes connected to $cv_i$
6:     for $n_i \in N$ do
7:         e $\leftarrow$ GETEDGE($cv_i, n_i$)     ▷ Get a CVDG edge connecting between $cv_i$ and another CVDG node
8:         $M$[e] $\leftarrow$ GETINSTSFOREDGE(e, $S$)          ▷ Mapping instructions to each edge
9: return $M$
10: function BACKWARDSLICINGVARSET($SV$)                     ▷ This function is called recursively
11:     $V \leftarrow SV$
12:     $S \leftarrow \emptyset$                                ▷ Backward slicing set for the given variable set
13:     for $v_i \in SV$ do
14:         $S \leftarrow$ BACKWARDSLICINGONEVAR($v_i$)
15:         $S \leftarrow S \cup S$                             ▷ Add new slicing results for each $v_i$
16:         $V \leftarrow$ GETAFFECTINGVARS($S$) $- V$            ▷ Get newly found variables
17:         $V \leftarrow V \cup V$
18:         for $v_i \in V$ do                                ▷ Perform recursive slicing on new variables
19:             $PV \leftarrow$ POINTS-TOANALYSIS($v_i$)
20:             $S \leftarrow$ BACKWARDSLICINGVARSET($PV$)      ▷ Recursive slicing
21:             $V \leftarrow V \cup$ GETAFFECTINGVARS($S$)
22:             $S \leftarrow S \cup S$                         ▷ Add new slicing results for each $v_i$
23:     return $S$
```

slices during a recursion, this process will continue until no more affecting variable or alias exists.

In the final step, Algorithm 1 goes through the identified program code paths for each CVDG edge and reports only those that begin and end – respectively – with the source and sink variables on the CVDG edge (Line 5-8).

2.4.3 Control Program Instrumentation

With the mapping from control model to program (CVDG nodes $\rightarrow$ variables; edges $\rightarrow$ code), MAYDAY now instruments the control program for logging the execution of the *CVDG-mapped portion* of the program, which bridges the semantic gap between control-level incidents and program-level root cause analysis. To achieve this, MAYDAY instruments LLVM bitcodes by inserting program-level logging functions at entries of basic blocks selected from the CVDG-mapped portion of the control program, and adds control loop iteration number into a logging function.

Efficient Logging of Program Execution. A key requirement of control program execution logging is high (time and space) efficiency. MAYDAY meets this requirement via two methods. The first method is *selective basic block logging*. MAYDAY only instruments the basic blocks of the CVDG-mapped program code. For example, in ArduPilot, the CVDG-mapped basic blocks are about 40.08% of all basic blocks. The second method is *execution path encoding*, which involves inserting logging functions at proper locations to record *encoded* program execution paths. We adopt Ball-Larus (BL) algorithm [51] – an efficient execution path profiling technique with path encoding. Under BL algorithm, each execution path is associated with a path ID, which efficiently represents its multiple basic blocks in the order of their execution.

Temporal Log Alignment. To temporally align the control log and the added program execution log, MAYDAY generates control loop iteration numbers (plus timestamps) at run-time and tags them to both control and program execution logs. Such alignment enables temporal navigation of log analysis during a post-accident investigation.

2.5 Post-Accident Investigation

After control-guided program analysis and instrumentation, the subject RAV will be back in service and start generating both control- and program-level logs during its missions. In the case of an accident, the logs will be recovered and analyzed by MAYDAY in a two-stage investigation to reveal the accident's root cause.

2.5.1 Control-Level Investigation

The control-level investigation has two main steps: (1) identify which controller, among all the primitive controllers in the CVDG, was the *first* to go wrong during the accident (Section 2.5.1); (2) infer the possible sequence of control variable corruption, represented as a corruption path in the CVDG, that led to that controller's malfunction.

Initial Digressing Controller Identification

During an RAV accident, multiple controllers in the CVDG may go awry, which leads to the operation anomaly of the vehicle. However, because of the inter-dependency of controllers (defined in the CVDG), there must exist one controller that is *initially* malfunctioning, whereas the others are causally affected and go awry later following the inter-dependency and control feedback loop. To uncover the root cause of the accident, it is necessary to identify the first malfunctioning controller, as well as the time when the malfunction started.

More formally, the malfunction of a controller manifests itself in two perceivable ways [19]: (1) non-transient digression between the control *state and reference* and (2) non-transient digression between the control *reference and mission input*. (1) means that the real state of the vehicle cannot "track" (i.e., converge to) the reference (i.e., desired state) generated by the controller; whereas (2) means that the reference cannot approach the target state set for the flight mission. As such, we call the first controller that exhibited (1) or (2) the *initial digressing controller*; and we call the time when the digression started the *initial digressing time*.

To identify the initial digressing controller and time, MAYDAY examines the control log. Similar to [19], a sliding window-based digression check is performed on each primitive controller (1) between state and reference and (2) between reference and mission input. Unlike the previous work, MAYDAY uses the Integral Absolute Error (IAE) formula [52] in a distinct way to identify the initial digression in a *reverse* temporal order (details are discussed in the next paragraph). By performing the digression check with the sliding window from the crash point backward, we identify the first digression window (hence time) of that controller, from which the digression persists toward the end of the log. The controller with the *earliest* first-digression window is the initial digressing controller.

Parameters for Digression Determination. We used threshold (Thr) and time window (w) (refer to the IAE formula [52] and Equation 2.1) for both initial digression determination 2.5.1 and state consistency check in Section 2.5.1. For the selection of reasonable Thr and w, we used the three-sigma rule [53] with fifty different experimental missions for 6DoF, similar to the previous work [19]. Compared to w in the previous work, we used much smaller windows to detect the more accurate time when the initial digression occurred. Specifically, we used 0.5 seconds for the x-, y-axis controllers, z-axis position, acceleration controllers, and yaw and yaw rate controllers. In addition, we used 0.25 seconds for the z-axis velocity controller, and roll and roll rate controllers, and pitch and pitch rate controllers.

CVDG-Level Corruption Path Inference

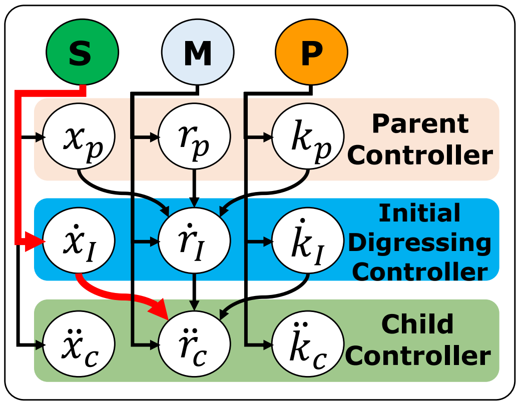
Given the initial digressing controller and the pair of digressing variables (i.e., "state and reference" or "reference and mission input"), MAYDAY will infer the sequence of operations on relevant control variables that had caused the initial digression. Such inference is guided by the CVDG model and the operation sequence of digression-inducing variables is called *CVDG-level corruption path*, represented by a directed path in the CVDG.

We first define several terms. Each primitive controller has three inputs: sensor input S, flight mission M, and control parameter P, with M and P coming from ground control (GCS). x_I, r_I, and k_I denote the control state, reference, and parameter (a vector) of the initial digressing controller – denoted as C_I. x_c, r_c, and k_c denote the control state, reference, and parameter of C_I's child (i.e., immediate downstream) controller – denoted as C_c, respectively. Now we present the inference of CVDG-level corruption path as summarized in Figure 2.8.

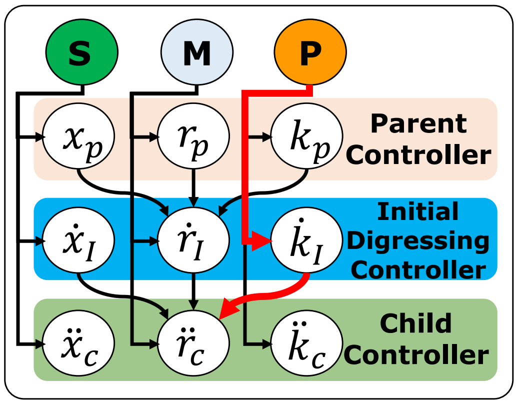
If the initial digression is between x_I **and** r_I, we can infer that x_I failed to track r_I. There are three possible causes for this, which correspond to different CVDG-level corruption paths:

- **Type I**: x_I was corrupted "locally" during the sensor input data processing (e.g., filtering). In the CVDG, such corruption corresponds to path $S \rightarrow x_I \rightarrow r_c$ as described in Figure 2.8a.

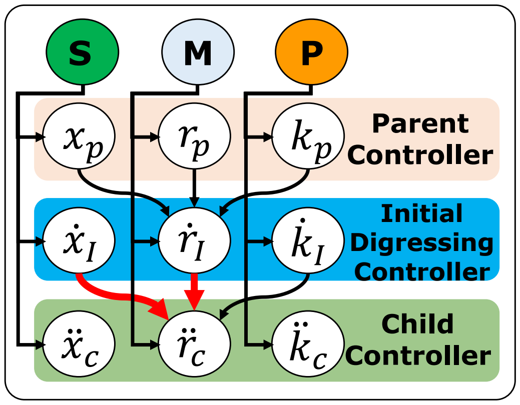
- **Type II**: x_I was corrupted indirectly via the control feedback loop. In this case, the control parameter k_I was first corrupted via GCS input (e.g., a parameter-changing command), which then corrupted r_c, the output of C_I. In C_c's effort to track the corrupted r_c, it generated the corrupted reference for its own child controller, and so on so forth. Finally, the RAV motors physically changed the vehicle's state, leading to the anomalous change of x_I. In the CVDG, such corruption corresponds to path $P \rightarrow k_I \rightarrow r_c$ as described in Figure 2.8b.



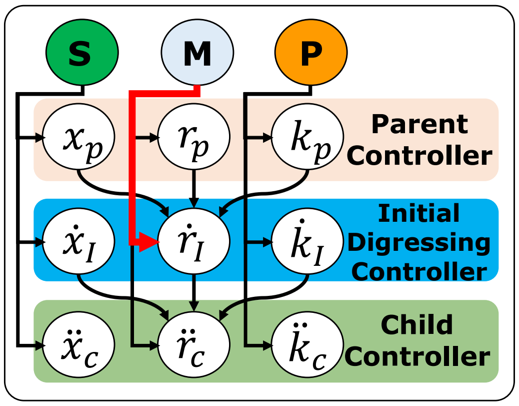
(a) Type I CVDG-level path



(b) Type II CVDG-level path



(c) Type III CVDG-level path



(d) Type IV CVDG-level path

Figure 2.8. Summary of CVDG-level corruption paths according to different corruption types.

- **Type III**: x_I was similarly (to Type II) corrupted via the control feedback loop, due to the corruption of r_c. Unlike Type II, r_c's corruption was not triggered by external input. Instead, it was caused by some execution anomaly along CVDG edge $x_I \rightarrow r_c$ or $r_I \rightarrow r_c$ as described in Figure 2.8c.

We point out that, between x_I and r_I, r_I cannot be initially corrupted by C_I's parent (upstream) controller. This can be proved by contradiction based on the CVDG model: If r_I were initially corrupted by its parent controller C_p, the corruption would have happened before C_I's initial digression. However, without C_I's digression, C_p would not be triggered by the control feedback loop to generate a corrupted r_I, unless C_I experienced a digression itself. But that would contradict with the fact that C_I is the *first* digressing controller.

To determine if an accident is caused by Type I or II/III corruption path, Mayday needs to check if x_I is corrupted locally or indirectly. This is done by checking the *state consistency* between C_I and C_c (i.e., between x_I and x_c). Intuitively, the state consistency is an indication that C_c makes control decisions following the "guidance" – either right or wrong – of C_I; and the observation of C_c is consistent – according to physics laws – with that of C_I. For example, if C_I is a velocity controller and C_c is an acceleration controller, then x_I (velocity) is consistent with x_c (acceleration), provided that the observed velocity x_I closely matches the velocity computed using the actual acceleration x_c (via integration) in each iteration. Since x_c did not digress from r_c when x_I digressed from r_I (by C_I's definition), if x_I and x_c are consistent, then we can infer that x_I is not locally corrupted and the CVDG-level path for x_I's corruption should be of Type II or III. Otherwise, the corruption path for x_I's corruption should be of Type I.

If the initial digression is between r_I and mission input M (Type IV), we can infer that a mission input (e.g., a GCS command to change trajectory or velocity) must have led to the change of r_I; and the new r_I value made C_I malfunction. In the CVDG, the corruption of r_I happened on path $M \rightarrow r_I$, as described in Figure 2.8d. Similar to Types I-III, we can prove that the parent controller of C_I cannot initially corrupt r_I.

Since x_I and x_c are not directly comparable, we leverage the following equation to define state consistency:

$$err(C_I, C_c) = \frac{\int_t^{t+w_I} |x_I(s+w_c) - x_I(s) - \int_s^{s+w_c} \lambda(x_c(v))dv|ds}{w_I} \tag{2.1}$$

In summary, Table 2.1 shows all four types of CVDG-level corruption paths and their determination conditions, to be applied during the investigation. Notice that the four types fully cover the CVDG edges in the initial digressing controller.

Table 2.1. Four types of CVDG-level corruption paths.

Type	Initial Digressing Variables	x_I and x_c Consistent?	Initially Corrupted Variable	CVDG-Level Corruption Path
I	Between r_I and x_I	No	x_I	$S \rightarrow x_I \rightarrow r_c$
II	Between r_I and x_I	Yes	k_I	$P \rightarrow k_I \rightarrow r_c$
III	Between r_I and x_I	Yes	r_c	$x_I \rightarrow r_c$; $r_I \rightarrow r_c$
IV	Between M and r_I	Yes	r_I	$M \rightarrow r_I$

2.5.2 Program-Level Investigation

The control-level investigation generates two outputs: (1) the initial digressing controller (and time) and (2) the CVDG-level corruption path that had led to the digression. With these outputs, MAYDAY transitions to its program-level investigation, analyzing a *narrowed-down* scope of the control program execution log. The final result of this investigation is a small subset of control program code (in basic blocks) where the bug causing the accident can be located.

Transition to Program-Level Investigation

MAYDAY first makes the following preparations: (1) mapping the control variables on the CVDG-level corruption path to program variables, based on the control model → program mapping established during the offline analysis (Section 2.4); (2) locating the program trace for the initial digressing iteration – recall that the log has been indexed by control loop iteration number – as the starting point for (backward) log analysis; and (3) restoring the LLVM instruction trace from the encoded log for LLVM bitcode-level data flow analysis.

CVDG-Guided Program Trace Analysis

MAYDAY first identifies the data flows of program-level variable corruptions representing the CVDG-level corruption path. It runs Algorithm 2 to identify such data flows, starting from the initial digressing iteration and going backward. There are four inputs to Algorithm 2: (1) the restored LLVM bitcode-level program trace, indexed by control loop iteration

Algorithm 2 Identification of basic blocks implementing a CVDG-level corruption path.

Input: CVDG (G), decoded program execution logs (L), CVDG-level corruption path (P_{cvdg}), control loop iteration with initial digression ($\mathrm{i}_{digress}$)
Output: A set of basic blocks of the program-level corruption paths

```
1: $P_{prog} \leftarrow$ BACKTRACK($P_{cvdg}, 0, \mathrm{i}_{digress}$)          ▷ Get program-level data flows
2: $\mathrm{i}_{trigger} \leftarrow P_{prog}.\mathrm{i}_{start}$          ▷ Control loop iteration with the triggering input
3: while $\mathrm{i}_{trigger} \leq \mathrm{i}_{digress}$ do          ▷ Find additional data flows
4:   $\mathrm{i}_{digress} \leftarrow P_{prog}.\mathrm{i}_{end} - 1$
5:   $P_{prog} \leftarrow P_{prog} \cup$ BACKTRACK($P_{cvdg}, \mathrm{i}_{trigger}, \mathrm{i}_{digress}$)
6: return GETBB($P_{prog}$)
7: function BACKTRACK($P_{cvdg}, \mathrm{i}_{start}, \mathrm{i}_{end}$)
8:   $P_{prog} \leftarrow \emptyset$
9:   for e $\in P_{cvdg}$ do
10:      $P_{prog} \leftarrow P_{prog} \cup$ BACKTRACKSRCSINK(e.$src$, e.$sink$, $\mathrm{i}_{start}, \mathrm{i}_{end}$)
11:   return $P_{prog}$
12: function BACKTRACKSRCSINK($src, sink, \mathrm{i}_{start}, \mathrm{i}_{end}$)
13:   if $src = sink$ then
14:     return $\emptyset$
15:   $P_{prog} \leftarrow \emptyset$
16:   for i $\in \{\mathrm{i}_{end} ... \mathrm{i}_{start}\}$ do          ▷ Backtrack the executed paths at every iteration
17:     $P_{\mathrm{i}} \leftarrow G$.GETDATAFLOWPATHS($L$[i], $src, sink$)          ▷ Between source and sink
18:     for $p \in P_{\mathrm{i}}$ do
19:       for $sink_p \in p.sinks$ do          ▷ Consider intermediate variables
20:         $P_{prog} \leftarrow P_{prog} \cup$ BACKTRACKSRCSINK($src, sink_p, \mathrm{i}_{start}$, i)
21:   return $P_{prog}$
```

number; (2) the initial digressing iteration number ($\mathrm{i}_{digress}$); (3) the source and sink program variables that correspond to the start and ending nodes on the CVDG-level corruption path; [1] (4) the mapping between instructions in the trace and the program basic blocks they belong to. The output of Algorithm 2 is a small subset of control program basic blocks that may have been involved in the CVDG-level corruption path.

To explain Algorithm 2, we show a simple example in Figure 2.9: The initial digressing controller is the x-axis velocity controller, and the CVDG-level corruption path is $P \rightarrow \dot{k}_x \rightarrow \ddot{r}_x$. The initial digressing time is Iteration 4930. P, $\dot{k}_x$, and $\ddot{r}_x$ are mapped to program variables (`msg`, `_pi_vel_xy._kp`, and `_accel_target.x`). Algorithm 2 starts from the sink variable (`_accel_target.x`) in Iteration $\mathrm{i}_{digress}$ (4930) and finds a variable-corruption data flow from source variable `msg`, through intermediate variable `_pi_vel_xy._kp` (Line 1, 7-21), to sink variable `_accel_target.x`. Data flows that go through the intermediate variables (e.g., `_pi_vel_xy._kp`) are reconstructed using the additional sink information (Line 19-20). This information is retrieved via backward slicing (Line 17) as described in Section 2.4.2. In Figure 2.9, the data flow is $P_{4850} \rightarrow \dot{k}_{x,4850} \rightarrow V_{4,4850} \rightarrow V_{8,4929} \rightarrow \ddot{r}_{x,4930}$, which realizes

[1]↑For a Type II CVDG-level path (Table 2.1), we also identify the program variable that corresponds to the intermediate node k_I on path $P \rightarrow k_I \rightarrow r_c$.

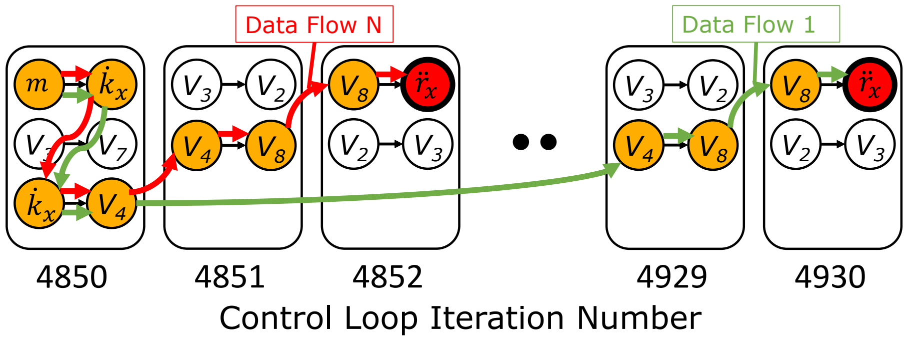


Figure 2.9. An example showing the working of Algorithm 2.

CVDG-level path $P \rightarrow \dot{k}_x \rightarrow \ddot{r}_x$. In particular, Iteration 4850 is the starting iteration of control variable corruption with the triggering input (P). We denote this iteration as $\mathrm{i}_{trigger}$.

After identifying the latest (relative to $\mathrm{i}_{digress}$) program-level variable corruption data flow, Algorithm 2 will continue to identify all *earlier* data flows that reflect the same CVDG-level corruption path between Iterations $\mathrm{i}_{trigger}$ and $\mathrm{i}_{digress}$ (Line 3-5, 7-21). In Figure 2.9, such an earlier data flow is $P_{4850} \rightarrow \dot{k}_{x,4850} \rightarrow V_{4,4850} \rightarrow V_{8,4851} \rightarrow \ddot{r}_{x,4852}$. We point out that, different from traditional program analysis, MAYDAY needs to capture the influence on the corrupted control variable ($\ddot{r}_x$) in *multiple* control loop iterations towards (and including) $\mathrm{i}_{digress}$. This is because, in a control system, each update to that variable may contribute to the final digression of the controller – either directly or via the control feedback loop – and hence should be held accountable.

Once Algorithm 2 finds all the data flows of program-level variable corruption, it can identify the corresponding basic blocks that implement each of the corrupting data flows (Line 6). In most cases, the multiple data flows will be mapped to the *same* set of program basic blocks, because of the iterative nature of control program execution. For example, the two corruption data flows in Figure 2.9 share the common segment $V_4 \rightarrow V_8 \rightarrow \ddot{r}_x$ implemented by the same set of basic blocks. This helps keep the number of basic blocks reported by Algorithm 2 small, making it easy for investigators to examine the source code of those basic blocks to finally pinpoint the bug that caused the accident.

2.6 Implementation

We have implemented Mayday for an IRIS+ quadrotor with a Raspberry Pi 3 Model B (RPi) [54] as the main processor board powered by a 1.2 GHz 64 bit quadcore ARM Cortex-A53 CPU with 1 GB SDRAM. Attached to the RPi are a Navio2 sensor board and a 64 GB SD card. The sensor board has a number of sensors (GPS, gyroscope, barometer, etc.) and is equipped with four actuators and a telemetry radio signal receiver. The control program is the popular ArduPilot 3.4 on Linux 4.9.45, with the main control loop running at a default frequency of 400 Hz.

For Mayday's control program analysis (Section 2.4.2), we leverage the SVF 1.4 static analysis tool [49] for the points-to analysis. We modified SVF to support our inter-procedural backward slicing and control program instrumentation on LLVM 4.0. Mayday's control- and program-level investigation functions (Section 2.5) are implemented in Python 2.7.6. The entire Mayday system contains 10,239 lines of C++ code and 7,574 lines of Python code.

2.7 Evaluation

We evaluate Mayday's effectiveness with respect to RAV accident investigation (Section 2.7.1) and bug localization (Section 2.7.2); and Mayday's efficiency with respect to run-time, storage, and energy overhead (Section 2.7.3).

2.7.1 Effectiveness of Accident Investigation

Summary of Cases. We investigated 10 RAV accidents based on *real* cyber-physical bugs in ArduPilot 3.4. Table 2.2 summarizes the nature of the 10 accidents, with respect to categorization, physical impact, triggering condition, nature of control program bug, patching status, and vulnerability status. We chose these cases by the following criteria: (1) their root causes are real cyber-physical bugs; (2) the specific nature of the bugs should be representative (e.g., invalid control/mission parameter values, integer overflow, and divide-by-zero);

(3) the initial digressing controllers in these cases should cover all six degrees of 6DoF; and (4) the CVDG corruption paths in these cases should show diversity.

Specifically, Cases 1-4 are caused by controller parameter corruption, which corresponds to Type II CVDG-level path in Table 2.1 (Section 2.5.1) and results in unrecoverable vehicle instability, deviation, or even crashes. Cases 5-7 are caused by corruption of flight missions (e.g., location, velocity), which corresponds to Type IV CVDG-level path in Table 2.1. Cases 8-10 are caused by data (e.g., sensor or GCS input) processing errors such as divide-by-zero, which corresponds to either Type I (Case 10) or Type II (Cases 8-9) CVDG-level path in Table 2.1.

The root causes of these accidents are real cyber-physical bugs that exist in ArduPilot 3.4 or earlier. The ones in Cases 5-10 are known bugs that have since been patched; whereas the bugs in Cases 1-4 still exist in the later version of ArduPilot 3.5. Our code review shows that the patches for those four bugs only fix the RAV's pre-flight parameter-check code, but not the in-flight parameter adjustment code. We alerted the ArduPilot team that the bugs in Cases 1-4 are not fully patched. Their reply was that, the four bugs were recently reported and confirmed along with other "invalid parameter range check" bugs. However, if ArduPilot fixes every parameter check, the firmware size may not fit in the memory of some resource-constrained micro-controllers supported by ArduPilot [2].

The "Patch Commit Number" column in Table 2.2 shows the patch commit numbers for all cases. Detailed ArduPilot bug-patching history, including the code snippets involved, can be accessed at: *https://github.com/ArduPilot/ardupilot/commit/[commit number].*

Note that these accidents are not easy to reproduce or investigate. Their occurrences depend on vehicle-, control-, and program-level conditions. For example, the control program bugs may be triggered only when the vehicle takes a certain trajectory (Cases 1-4) and/or accepts a certain controller parameter or flight mission (Cases 1-9). Or they can only be triggered by a certain environment factor (e.g., wind speed in Case 10). Such accidents abound in real-world RAV operations [48]. Due to their hazardous nature and in compliance with safety regulations, we run these realistic accidents using a software-in-the-loop (SITL) RAV simulator [55], with a real control program and logs but simulated vehicle and exter-

[2]↑https://github.com/ArduPilot/ardupilot/issues/12121

Table 2.2. List of accident cases caused by cyber-physical bugs.

Case ID	Category	Impact	Condition	Root Cause (Bug)	Patch Commit Number	Still Vulnerable in ArduPilot 3.5 and up?
1	Controller Parameter Corruption	Extreme vehicle instability or fly off course	Command & turn	No range check of k_P parameter for x, y-axis velocity controllers	9f1414a*	Yes
2		Extreme vehicle instability or crash	Command & altitude change	No range check of k_P parameter for z-axis velocity controller	9f1414a*	Yes
3		Extreme vehicle instability or crash	Command & turn	No range check of k_P parameter for roll angular controller	9f1414a*	Yes
4		Extreme vehicle instability or crash	Command & turn	No range check of k_P parameter for pitch angular controller	9f1414a*	Yes
5	Flight Mission Corruption	Crash after slow movement	Command & speed change	Wrong variable name leading to out-of-range x, y-axis velocity	e80328d	No
6		Moving to an invalid location	Command	Wrong waypoint computation based on non-existent coordinate	9739859	No
7		Crash	Command	Invalid type-casting of z-axis location causing an integer overflow	756d564	No
8	Data Processing Error	Crash	Command	Missing divided-by-zero check of k_P parameter for z-axis position controller	c2a290b	No
9		Crash	Command	Missing divided-by-zero check of k_P parameter for x, y-axis position controllers	c03e506	No
10		Crash	Weak or no wind	Missing divided-by-zero check in angular calculation	29da80d	No

* The bug is partially patched by ArduPilot developers and still vulnerable.

nal environment. Widely used in drone industry, the SITL simulator provides high-fidelity simulation of the vehicle as well as the physical environment it operates in (including aerodynamics and disturbances). We leverage MAVLink [48] to trigger cyber-physical bugs by issuing GCS commands to adjust control/mission parameters. MAVLink is able to communicate with both real and simulated RAVs.

Investigation Results. Table 2.3 presents the results of our investigations using Mayday. For each case, Mayday first performs the control-level investigation, which identifies the initial digressing controller and infers the CVDG-level corruption path(s) by analyzing the control-level log. Mayday then performs then program-level investigation, which identifies the portion of control program code that implements the CVDG-level paths. We clarify that the final output of Mayday is not the specific buggy line of code per se. Instead, it is a small subset of program code (basic blocks) which the investigator will further inspect to pinpoint and confirm the bug.

Control-Level Investigation. The 2nd and 3rd columns of Table 2.3 show the initial digressing controller and the CVDG-level corruption path identified in each case, respectively. The 4th column shows the number of control loop iterations (duration) between the initial corruption of the control variable and the initial occurrence of controller digression. For Cases 1-7, that duration can be arbitrarily long. More specifically, the initial corruption of a control variable on the CVDG-level path may happen first in just a few iterations (e.g., 8 in Case 1). But the controller's initial digression could happen an arbitrary number of iterations later, depending on the timing of the vehicle's operation that "sets off" the digression (e.g., a turn or a change of altitude). Such "low-and-slow" nature of accidents makes it harder to connect their symptoms to causes and highlights the usefulness of Mayday.

Program-Level Investigation. The 5th and 6th columns of Table 2.3 show respectively the number of control program basic blocks and lines of source code identified by Mayday for each case. Notice that the numbers are fairly small (from 7 to 50 basic blocks, or 22 to 137 lines of code), indicating a low-effort manual program inspection. We confirm that the actual bug behind each case is indeed located in the code identified by Mayday.

Bug Detection Capability Comparison. We have also conducted a comparative evaluation with (1) two off-the-shelf bug-finding tools: Cppcheck 1.9 [56] and Coverity [57], and (2) RVFuzzer [19], to detect the bugs behind the 10 accident cases. We used the most recent stable version of Cppcheck with all its available analysis options to leverage Cppcheck's

Table 2.3. Investigation results of accident cases in Table 2.2. SLoC: Source lines of code.

Case ID	Control-Level Investigation			Program-Level Investigation		
	Initial Digressing Controller	CVDG-Level Corruption Path	# of Iterations from Initial Corruption to Initial Digression	# of Basic Blocks	SLoC	Bug Found?
1	x, y-axis Velocity	$P \rightarrow \dot{k}_{xy} \rightarrow \ddot{r}_{xy}$	≥ 4	34	89	✓
2	z-axis Velocity	$P \rightarrow \dot{k}_{z} \rightarrow \ddot{r}_{z}$	≥ 4	32	85	✓
3	Roll Angle	$P \rightarrow k_{roll} \rightarrow \dot{r}_{roll}$	≥ 4	50	121	✓
4	Pitch Angle	$P \rightarrow k_{pitch} \rightarrow \dot{r}_{pitch}$	≥ 4	50	121	✓
5	x, y-axis Velocity	$M \rightarrow \dot{r}_{xy}$	≥ 4	12	44	✓
6	x, y-axis Position	$M \rightarrow r_{xy}$	≥ 4	48	137	✓
7	z-axis Position	$M \rightarrow r_{z}$	≥ 4	48	135	✓
8	z-axis Position	$P \rightarrow k_{z} \rightarrow \dot{r}_{z}$	4	9	30	✓
9	x, y-axis Position	$P \rightarrow k_{xy} \rightarrow \dot{r}_{xy}$	4	41	94	✓
10	Roll, Pitch, Yaw Angle	$S \rightarrow x_{rpy} \rightarrow \dot{r}_{rpy}$	1	7	22	✓

full capability. For Coverity, we used its online service version. For RVFuzzer, we used its latest version. The results are shown in Table 2.4.

Comparison with Cppcheck and Coverity Neither Cppcheck nor Coverity reported any of the bugs behind the 10 cases. For Cases 1-6, without knowledge about the control model, it is impossible for Cppcheck and Coverity to check the validity of control/mission parameter input, or to determine if the RAV controller state – manifested by program state – is semantically valid or corrupted. For Case 7, the overflow of an integer program variable was not detected by either Cppcheck or Coverity. This was also confirmed by a Cppcheck developer[3]. For Cases 8-10, accurate detection of divide-by-zero bugs is hard for static analysis-based tools such as Cppcheck and Coverity. Without a concrete execution confirming a divide-by-zero instance, they cannot detect such bugs with low false positive and false negative rates.

[3]↑https://sourceforge.net/p/cppcheck/discussion/development/thread/eed7d492df

Table 2.4. Bug detection capability comparison results. ✓: bug triggered and located in source code, Δ: bug triggered and faulty input constructed, and ✗: bug not detected.

Case ID	Nature of Bug	Mayday	Cppcheck [56]	Coverity [57]	RVFuzzer [19]
1	Missing controller parameter range check	✓	✗	✗	Δ
2	Missing controller parameter range check	✓	✗	✗	Δ
3	Missing controller parameter range check	✓	✗	✗	Δ
4	Missing controller parameter range check	✓	✗	✗	Δ
5	Comparison with a wrong variable	✓	✗	✗	Δ
6	Wrong waypoint computation based on non-existent coordinate	✓	✗	✗	✗*
7	Integer overflow on a mission variable	✓	✗	✗	Δ
8	Divide-by-zero caused by invalid controller parameter	✓	✗	✗	Δ
9	Divide-by-zero caused by invalid controller parameter	✓	✗	✗	Δ
10	(Probabilistic) Divide-by-zero caused by sensor input	✓	✗	✗	✗

∗ The bug cannot be triggered under the default configuration of RVFuzzer. However, it can be triggered if RVFuzzer's flight simulation is re-configured.

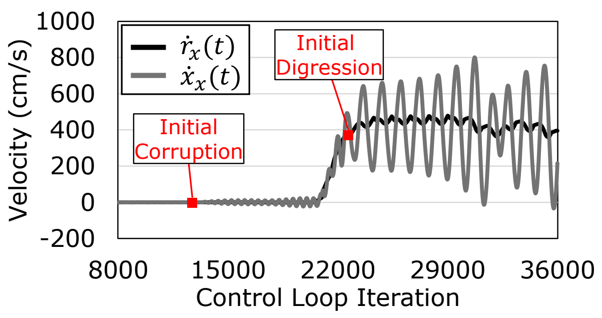
Our comparison results highlight the key differences between Mayday and the off-the-shelf bug-finding tools. First, Mayday complements the generic tools by serving as a specialized tool (i.e., for RAV control programs) for uncovering cyber-physical bugs that cause controller anomalies, instead of "syntactic" bugs that cause generic symptoms such as memory corruption and CFI violation. Second, unlike program debuggers, Mayday debugs an entire cyber-physical system based on both control- and program-level traces. Third, Mayday's bug localization is guided by the RAV control model and its mapping to the control code; whereas off-the-shelf debuggers are without such domain-specific knowledge.

Even if a static analysis tool is aware of value ranges of control parameters, Mayday is still necessary because (1) there is no existing static analysis tool that comes with or generates a parameter-range specification; (2) static analysis is prone to high false positives/negatives when detecting divide-by-zero bugs (Cases 8-10); and (3) static analysis cannot detect semantic bugs such as a wrong variable-name (Case 5), due to unawareness of control semantics. Mayday, based on actual RAV control program runs, overcomes these limitations.

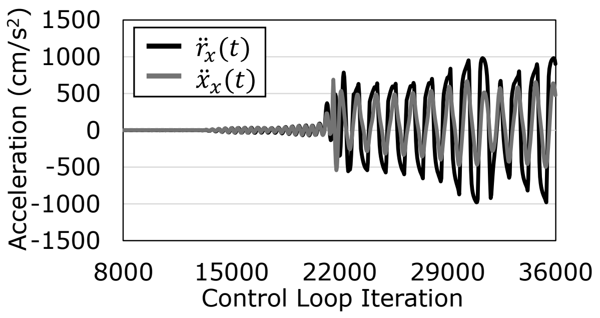
Comparison with RVFuzzer Among the 10 cases, RVFuzzer was able to trigger eight cases caused by GCS input validation bugs (i.e., lack of valid range check for run-time-adjustable control or mission parameters, as defined in [19]). RVFuzzer did not trigger Cases 6 and 10 for different reasons: (1) For Case 6, the reason is insufficient flight simulation time under RVFuzzer's default configuration. In this case, given an invalid input, RVFuzzer's simulation run terminated *before* controller anomaly could occur. However, RVFuzzer would have detected the bug in Case 6, if the simulation had run longer (for hours instead of minutes by default) for each input value. We note that RVFuzzer limits the simulation time to achieve high fuzzing throughput; and Case 6 manifests the trade-off between fuzzing coverage and throughput. (2) Case 10 cannot be detected by RVFuzzer because the bug is not a GCS input validation bug. Instead, it is triggered *probabilistically* by the wind speed sensor input.

In addition to Cases 6 and 10, we have found another interesting bug that RVFuzzer cannot detect: `PSC_ACC_XY_FILT` is a runtime-adjustable control parameter (which smooths the change in x, y-axis acceleration reference), with a default value of 2.0. During fuzzing, no controller anomaly is observed, when the value of `PSC_ACC_XY_FILT` is set to 2.0 and when the value is set to 0. Following its fuzzing space reduction heuristic, RVFuzzer will not test any other value between 0.0 and 2.0, assuming that [0, 2.0] is a safe range. But in fact, a positive value close to 0.0 (e.g., 0.0001) for `PSC_ACC_XY_FILT` *will* lead to controller anomaly and hence be missed by RVFuzzer. This bug can be demonstrated with a concrete attack, which can be investigated by Mayday similar to Cases 1-4 with a Type II CVDG-level corruption path.

More fundamentally, Mayday and RVFuzzer differ in two aspects: (1) Mayday reactively performs investigation to localize the bug in the *source code* that had led to an accident. Mayday involves CVDG-guided source code analysis and instrumentation to bridge the RAV control model and control program. RVFuzzer proactively discovers vulnerable



(a) X-axis velocity controller.



(b) X-axis acceleration controller.

Figure 2.10. Case 1: History of x-axis velocity and acceleration controllers – the former is the initial digressing controller.

inputs that cause controller anomalies, by treating the control *binary code* as a blackbox. (2) RVFuzzer automatically mutates values of control parameters that can be dynamically adjusted via GCS commands, to uncover vulnerable value ranges of those control parameters – namely *input validation bugs*. On the other hand, MAYDAY aims to trace back and pinpoint *cyber-physical bugs*, which include not only input validation bugs (e.g., Cases 1-4) but also other types of bugs such as flight mission corruption (e.g., Cases 6) and data processing error (e.g., Case 10).

Finally, our comparison between MAYDAY and RVFuzzer suggests an *integration* opportunity: Given an RAV control program (with both source and binary), we can first apply RVFuzzer to construct a concrete attack/accident – instead of waiting for one to happen – that indicates the existence of a vulnerable control/mission parameter. We then use MAYDAY to reproduce the accident/attack with the same malicious input, collect the control and program logs, and locate and patch the bug at the source code level. We can perform such integrated "fuzzing – debugging – patching" workflow for the eight cases detected by RVFuzzer.

Case Study: "Unexpected Crash after Turn"

We now present the investigations of Cases 1 and 5 as detailed case studies. In Case 1, the quadrotor's mission was to first stop at waypoint A to pick up a package, then fly straight north (along the y-axis) to waypoint B, where it would make a 90-degree turn to

```
 1 void GCS_MAVLINK::handle_param_set(..      // Parameter update
 2     ...
 3     //No range check
 4     vp->set_float(packet.param_value, var_type);
 5 Vector2f AC_PI_2D::get_p() const{
 6     ...
 7     return (_input * _kp);                  // No range check
 8 void AC_PosControl::rate_to_accel_xy(...  // Controller
 9     ...
10     //Access parameter _kp
11     vel_xy_p = _pi_vel_xy.get_p();          // No range check
```

Listing 2.1 Cyber-physical bug behind Case 1. The range check patch can be applied in Line 7.

fly east (along the x-axis) to the destination. After the pickup, to maintain the y-axis speed (5 m/s) with the increased payload, the operator issued a parameter-changing command via GCS to increase the k_P parameter, shared by both x- and y-axis velocity controllers. The flight from A to B was normal. Unexpectedly, when the vehicle made the scheduled turn at B, it became very unstable and soon lost control and crashed.

MAYDAY first performs the control-level investigation. By analyzing the control-level log, MAYDAY finds that the initial digressing controllers are the x- and y-axis velocity controllers, both with digression between the vehicle velocity state ($\dot{x}_{xy}$) and reference ($\dot{r}_{xy}$) starting at around Iteration 23267 (after the scheduled turn at Iteration 20858). Figure 2.10a shows the x-axis velocity state and reference. [4] Next, MAYDAY checks their child controllers (i.e., the x, y-axis acceleration controllers) and confirms that the child controllers did not exhibit any digression (i.e., $\ddot{x}_{xy}$ always tracked $\ddot{r}_{xy}$), even after the velocity controllers' digression. Figure 2.10b shows the x-axis acceleration state and reference. Based on Table 2.1, MAYDAY infers that the CVDG-level corruption path is $P \rightarrow \dot{k}_{xy} \rightarrow \ddot{r}_{xy}$ (Type II).

MAYDAY then performs the program-level investigation. It runs Algorithm 2 on the program execution log, starting from Iteration 23267 and going backward, to find data flows that correspond to the CVDG-level corruption path. The multiple data flows found by the algorithm reveal that they all started from the parameter-changing GCS command (P), which led to the modification of k_P (which is part of $\dot{k}_{xy}$) during Iteration 13938 – much

[4]↑Those for y-axis velocity are omitted to avoid duplication.

earlier than the digression (23267). k_P remained unchanged after Iteration 13938. Finally, MAYDAY maps the data flows to 34 basic blocks, among which we (as investigator) find the actual bug.

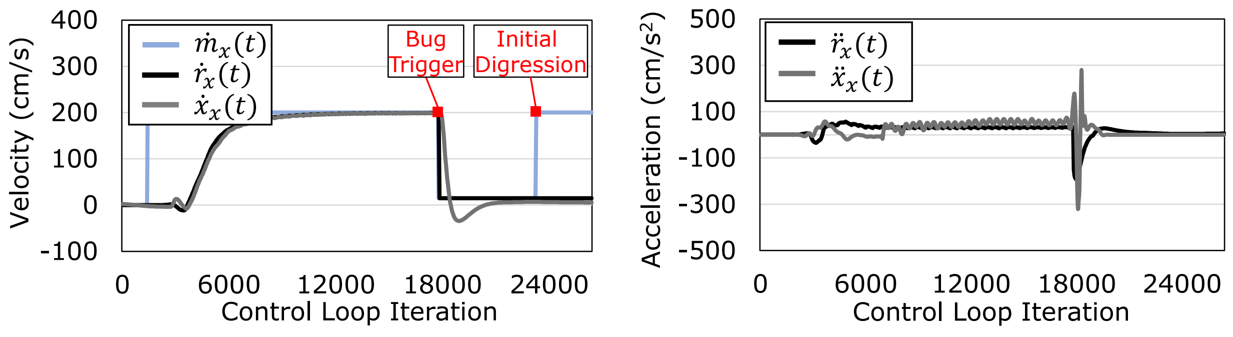
Listing 2.1 shows the code snippets with the bug. When a parameter-changing command is received, `set_and_save` saves the new parameter value. The value is later retrieved by `get_p`, when `rate_to_accel_xy` is called by the x, y-axis velocity controller. The code indicates that the controller would accept any k_P value from the GCS without a range check! (A range check should be added at Line 7.) The relevant log also shows that, despite the improper k_P value, the vehicle remained stable from A to B. This is because the x- and y-axis velocity controllers are not sensitive to k_P under *constant* speed with negligible instantaneous error (i.e., $\dot{r}_{xy} - \dot{x}_{xy}$). However, when the vehicle turned 90 degrees, the x-axis velocity had to increase from 0 m/s to 5 m/s (and the opposite for y-axis velocity) and the impact of k_P manifested itself during the acceleration/deceleration.

Case Study: "'Frozen' Velocity after Slowdown"

While Case 1 was caused by corruption of control parameters (Type II), Case 5 was triggered by corruption of flight mission (Type IV). We note that this case was first discussed by [19] as an attack scenario; and the corresponding vulnerability was found but *without* exact reasoning of the root cause (bug) at source code level. Here, we demonstrate how MAYDAY can locate the bug via post-accident/attack investigation.

In Case 5, the quadrotor flew east-bound (along the x-axis) at a velocity of 2 m/s. During one segment of the flight, the vehicle is supposed to take aerial survey video of a specific landscape (e.g., an archaeology site) hence the operator issued a mission-changing command to reduce the vehicle speed to 15 cm/s so that the on-board camera could capture detailed, slow-progressing view of the landscape. After the video-shooting operation, the vehicle was supposed to resume the 2 m/s cruising velocity. However, it seemed to get "stuck" in the 15 cm/s velocity and did not respond to any velocity-changing command from the operator.

MAYDAY first performs the control-level investigation. From the control-level log, it finds that the initial digressing controller is the x-axis velocity controller, with the digression



(a) X-axis velocity controller (b) X-axis acceleration controller

Figure 2.11. Case 5: History of x-axis velocity and acceleration controllers – the former is the initial digressing controller.

```
class AC_PosControl {
public:
  float get_max_speed_xy() const { return _speed_cms; }
...
void AC_WPNav::set_speed_xy(float speed_cms) {
  // range check new target speed
- if(_pos_control.get_max_speed_xy() >=
-     WPNAV_WP_SPEED_MIN){                        // Buggy code
+ if(speed_cms >= WPNAV_WP_SPEED_MIN){            // Patched code
     _pos_control.set_max_speed_xy(_wp_speed_cms);
     // flag that wp leash must be recalculated
     _flags.recalc_wp_leash = true;
```

Listing 2.2 Cyber-physical bug behind Case 5.

between the velocity reference $\dot{r}_x$ and the operator-set velocity (which is part of mission M), starting from Iteration 23629 (Fig 2.11a). Different from Case 1, there is no digression between the x-axis velocity state ($\dot{x}_x$) and reference ($\dot{r}_x$), hence the vehicle did not lose control during the entire flight, despite the "frozen" speed. MAYDAY also confirms that the child controller (i.e., the x-axis acceleration controller) did not exhibit any digression (Fig 2.11b). In other words, both velocity and acceleration states correctly tracked their respective references and hence are consistent. Based on Table 2.1, MAYDAY infers that the CVDG-level corruption path is $M \rightarrow \dot{r}_x$ (Type IV).

Next, MAYDAY performs the program-level investigation. Starting from the program execution log at Iteration 23629 and moving backward. Algorithm 2 finds the data flow that corresponds to the CVDG-level corruption path: It started from the velocity-changing (from

2 m/s to 15 cm/s) command at Iteration 17736, which led to the modification of x-axis velocity reference ($\dot{r}_x$) at Iteration 17742. MAYDAY reports 12 basic blocks that may be involved in the data flow.

From the 12 basic blocks, we pinpoint the bug as shown in Listing 2.2. The code *intends* to enforce a minimum mission velocity (`WPNAV_WP_SPEED_MIN`, which is 20 cm/s in ArduPilot) through a range check on the *flight mission* velocity input (`speed_cms`) (Line 9, which is the patch). But the code, by mistake, compares the minimum mission velocity with the *current* velocity `_pos_control.get_max_speed_xy()`, not with the *set* velocity `speed_cms` (Line 7)! This bug caused the control program to accept the 15cm/s velocity, which is lower than the minimum mission velocity. Even worse, after this velocity change, the x-axis velocity controller will refuse to accept *any other* velocity change, because the result of the (buggy) comparison will always be FALSE. The 12 basic blocks identified by MAYDAY cover the buggy statement with the wrong variable name, which RVFuzzer [19] cannot report.

2.7.2 Scope Reduction for Bug Localization

As shown in Section 2.7.1, MAYDAY can significantly narrow down the scope of control program code for manual inspection to pinpoint a bug, thanks to 1) control model (CVDG)-guided corruption inference and 2) program execution logging. In this section, we define and implement a baseline investigation method *without* adopting these two ideas. We then compare MAYDAY with the baseline, with respect to the number of basic blocks they identify for bug localization.

The baseline model only analyzes the control program source code and control-level log. To its favor, we assume that the baseline method is able to identify at least one corrupted control variable based on the control-level log. From the corrupted variable, it performs static analysis (i.e., point-to analysis and backward slicing) to identify the corresponding basic blocks that implement the slice. Figure 2.12 shows, in log scale, the number of basic blocks reported by the baseline method for each of the 10 cases in Section 2.7.1, comparing with MAYDAY. For each case, the baseline method reports thousands of basic blocks for bug localization; whereas MAYDAY reports tens of them. This comparison highlights the

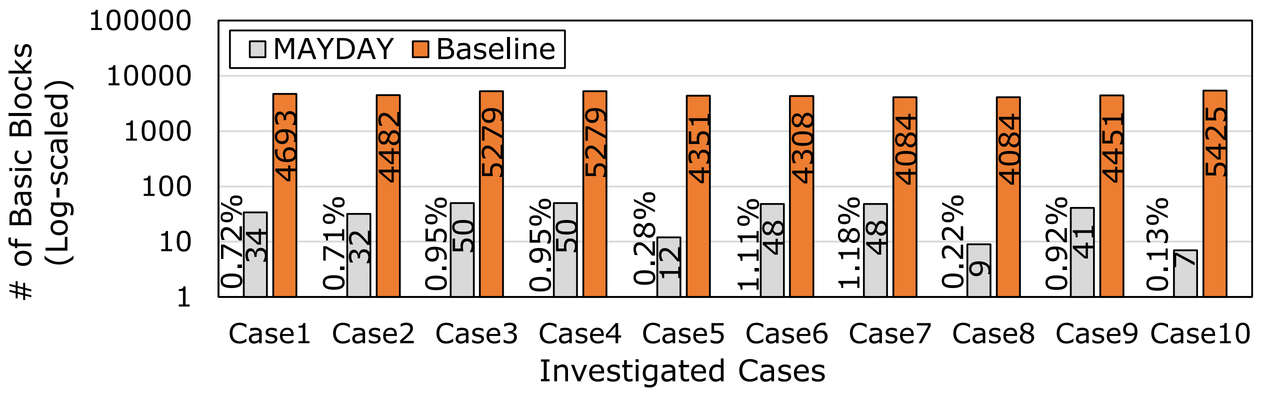


Figure 2.12. Number of basic blocks reported by the baseline investigation method and by MAYDAY.

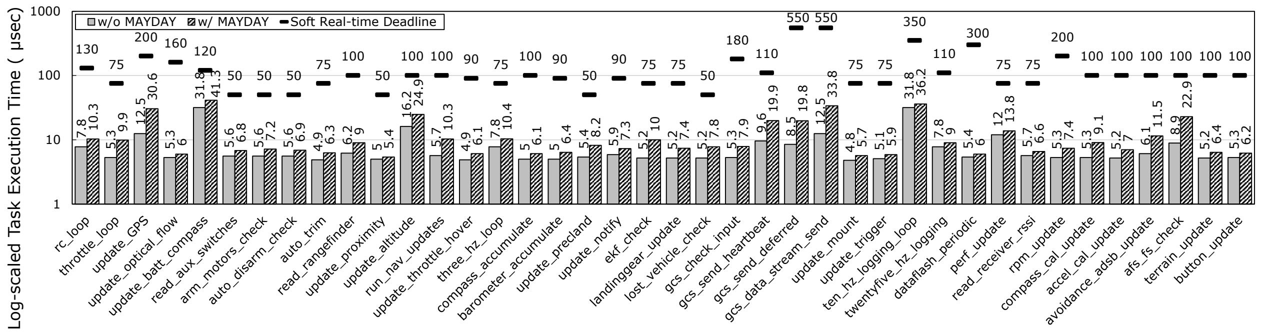


Figure 2.13. Run-time overhead of MAYDAY: average execution time of soft real-time tasks with and without MAYDAY in log scale. While MAYDAY introduces run-time overhead, it still meets the real-time requirement without missing deadlines.

benefit (and novelty) of MAYDAY's control model guidance and program-level logging, which mitigates the long-existing problem of state explosion [41], [58] faced by generic program attack provenance.

2.7.3 Run-time, Storage and Energy Overhead

By identifying the basic blocks that implement the data flows in the CVDG (Section 2.4.3), we instrumented and logged 40.08% of the basic blocks in ArduPilot, introducing run-time, storage, and energy overheads. We measure these overheads using a *real quadrotor RAV*.

Run-time Overhead. We measure the execution time of the 40 soft real-time tasks in ArduPilot during 30-minute flights with twenty random and different flight operations, with and without MAYDAY. The execution frequencies of the ArduPilot tasks vary, from 0.1 Hz to 400 Hz. The results are shown in Figure 2.13, with each task's average execution time and its soft real-time deadline (defined in ArduPilot) in log scale.

The results show that MAYDAY does increase the task execution time. Relative to the execution time without MAYDAY, the increase ranges from 8% to 170% However, *comparing to the soft real-time deadline* of each task, the increase (i.e., the increment/deadline ratio) is small, ranging from 0.02% to 14.0% and averaging at 3.32%. As expected, our selective instrumentation method tends to impose higher overhead on functions that frequently access control variables (e.g., `update_GPS` and `run_nav_updates`) and lower overhead on functions that do not.

We further breakdown the logging overhead between log generation (e.g., program path encoding) and I/O (writing to SD card), as shown in Table 2.5. With a 400 Hz control loop frequency, MAYDAY's logging takes 7.6% of the time in one iteration – 190.72 μs in total. We note that such run-time fine-grain program tracing is feasible, thanks to the intrinsically low control frequency of cyber-physical systems, relative to that of their controller CPUs.

Table 2.5. Logging overhead breakdown.

	Average Latency / Iteration (μs)	Breakdown (%)
Log generation	37.22	19.71
Log I/O	153.5	81.29
Logging total	190.72	100

Storage Overhead. We measure MAYDAY's log data generation rate and volume during the 30-minute experiment. The average log generation rate is 742.8 KB/s: 15.4 KB/s for ArduPilot's existing vehicle control log and 717.4 KB/s for our program execution log. The total log volume is no more than 1.3 GB in 30 minutes, which is the typical maximum flight time for many commodity RAVs, such as Navio2, DJI Phantom 4 and Parrot Bebop2. Such a volume can be easily accommodated by lightweight commodity storage devices (e.g., our 64 GB SD card).

Battery Consumption. MAYDAY consumes fairly small amount of battery power, compared with the RAV motors. Our quadrotor is equipped with four motors whose total power consumption is approximately 147.5-177.5 Watts [59] excluding the computing board's power consumption (2.5 Watts). According to specifications, our sensor board consumes no more than 0.65 Watt [60], and its main processor board consumes a maximum of 5.0 Watts (less than 3.69% of the overall power consumption), with other attached devices (e.g., SD card) powered via the main processor board [61]. MAYDAY's power consumption is covered by the main processor board and therefore an even smaller fraction of the overall power consumption.

2.8 Discussion

Code and Log Protection. We assume code integrity after instrumentation, log integrity, and log recover-ability in MAYDAY. To achieve code integrity, we can apply content-based integrity checking [25], [26] via remote attestation [62], [63]. We can also apply disk content integrity techniques [27] for log integrity. To recover from log corruption, special file system techniques (e.g., journaling file systems [64]) may be applied.

To protect kernel and flight data recording (FDR) modules at run-time, we could apply kernel hardening (e.g., SecVisor [65], NICKLE [66], and nested kernel [67]) and persistent data protection (e.g., InkTag [68]) techniques. However, many of those techniques are not suitable for resource-constrained RAV micro-controller platforms. Fortunately, there exist lightweight memory isolation techniques [10], [11], [69] that can protect security-critical modules (e.g., kernel and FDR) with low overhead. In particular, MINION [10] can be readily deployed with ArduPilot for memory access protection, even on low-end micro-controllers with only an MPU (memory protection unit). Additionally, we could consider Date Execution Prevention (DEP) [30] for thwarting code injection.

Log Volume Reduction. We assume that the subject RAV has enough storage space to store logs in light-weight, low-cost devices such as commodity SD cards. However, future control programs may generate a larger volume of logs due to the complexity of their control algorithms and the fact that MAYDAY must record fine-grain, reproducible program execution paths/traces. Existing techniques reduce log size by (1) compressing the entire log [36] or (2) identifying and removing redundant log entries [43], [58]. Similar to (2), we plan to leverage control- and program-level dependencies to further reduce the log volume.

Scope of Applicability. We clarify that, rather than being a generic bug-finding tool, MAYDAY specializes in finding RAV cyber-physical bugs, which involve incomplete or incorrect *implementation* of the underlying control theoretical model. As acknowledged in Section 2.1, there exist other types of vulnerabilities in RAV systems, such as traditional program vulnerabilities and vulnerabilities in physical components (e.g., sensors). For physical attacks (e.g., sensor and GPS spoofing), MAYDAY is fundamentally not suitable, as the root cause of those attacks lies in the physical component (e.g., vulnerable sensing mechanism of a gyroscope device [70]), not in the control program. Hence MAYDAY's program execution trace analysis would not be necessary for detecting or investigating physical attacks.

Fortunately, defenses against physical attacks exist and can be deployed alongside with MAYDAY. Many sensor attacks can be detected by checking the RAV control log [33] for anomaly and inconsistency among sensors [70]. Physical sensor spoofing attacks can be detected by cross-checking the observed and expected controller states [12], [13]. GPS spoofing attacks can be detected by commodity hardware (e.g., `u-blox M8`) and advanced techniques [71], [72]. Jamming attacks can be defended against via existing solutions [73], [74].

More Robust Control Models. We acknowledge that more robust control models are technically possible and can make the RAV more tolerant of disturbances and changes. For example, a "self-examining" control algorithm can be designed to dynamically compute and verify the system's stability properties, in response to every GCS command. As another example, the PID control algorithm can be replaced by more advanced ones such as the Linear-quadratic regulator controllers [46] to better mitigate disturbances. However, such advanced control models are not yet widely adopted in commodity control programs (e.g., ArduPilot and PX4).

More importantly, the program-level *implementation* of advanced control theoretical models may still be buggy, due to programming errors (e.g., wrong variable names, missing parameter range checks, etc.) that MAYDAY is tasked to find out. In other words, despite increasing robustness of RAV control models, MAYDAY will continue to help debug their implementation at the program level to avoid misuses or exploits.

3. RVFUZZER: FINDING INPUT VALIDATION BUGS IN ROBOTIC VEHICLES THROUGH CONTROL-GUIDED TESTING

Despite the new paradigm of "cyber-physical" crash investigation introduced by MAYDAY, three open challenges remain. First, RAVs still have latent cyber-physical bugs since MAYDAY can only reactively find vulnerabilities after RAV accidents. Second, MAYDAY requires source code to (1) map the control model to the control program and (2) identify cyber-physical bugs. However, we cannot guarantee that source code is always available if RAV programs are legacy or closed-source programs. Finally, despite our purpose to discover cyber-physical bugs, we may not want to crash physical RAVs due to safety and cost issues.

To solve the aforementioned problems for traditional programs, researchers on the cyber side have developed black-box fuzzing techniques to hunt cyber-physical bugs without source code proactively. Unfortunately, those fuzzing works have focused only on detecting traditional "syntactic" bugs in the cyber domain. This is far from sufficient to detect cyber-physical bugs without considering the "physical" aspect. In this chapter, we will present RVFUZZER to bring the "cyber-physical" aspect to black-box fuzzing techniques for cyber-physical bug discovery.

3.1 Background

RAV Control Model. The RAV control model is the generic theoretical underpinnings that control the vehicle's movements and operations during its missions (e.g., flying in a trajectory with multiple waypoints). The RAV's movements are along its six degrees of freedom (6DoF), which include the x, y, and z-axes for movement and the roll, pitch, and yaw for rotation (Figure 2.1). The control model consists of multiple controllers, each for a specific degree of the 6DoF. For example, the x-axis controller is shown in Figure 2.3.

Inside the x-axis controller, there are three primitive controllers in a cascade, which are responsible for controlling the vehicle's position, velocity, and acceleration along the x-axis, respectively. Each primitive controller takes two state inputs: a reference state

($r(t)$) computed by its upstream primitive controller; and an vehicle state ($x(t)$) reported by sensors. The goal of the controller is to keep the vehicle state close to the reference state, via its core function of *controller state stabilization*. The output of the function is the reference state for its downstream primitive controller. Each primitive controller has multiple adjustable parameters and accepts high-level mission directives (e.g., change of target location or speed).

Overall the RAV control model involves complex dependencies between the 6DoF controllers, each having multiple parameters and accepting mission directives. Moreover, the controllers, sensors, and the vehicle's physical operations (e.g., those of motors) create a feedback loop, which enables the periodic, iterative working of the controllers.

RAV Control Program. An RAV control program implements the RAV control model. Correspondingly, it involves the following main modules: (1) a sensor module to collect sensor inputs (e.g., from GPS, inertial measurement unit, etc.) for periodic vehicle state observation, (2) a controller module to generate control output based on current mission, reference state, and sensor input, and (3) a mission module to interpret mission directives and execute them. These modules execute iteratively in the periodic control epochs.

During a flight, the RAV communicates with a ground control station (GCS), which may issue a variety of GCS commands to the control program. Many of those commands allow RAV operators to dynamically adjust the controller and mission parameters. We note that such a dynamic parameter change may be necessary to improve vehicle control performance (e.g., enhancing stability), in response to mission dynamics such as payload change and non-trivial external disturbances.

In addition to the control and communication functions, most RAV control programs have a runtime controller state logging function, for record-keeping and troubleshooting purposes. Real-world commodity RAVs (e.g., Intel Aero [75], 3DR IRIS+ [76], and DJI drone series [77]), as well as their simulators, log in-flight controller states in persistent storage. RVFuzzer leverages such logs for automatic determination of controller malfunction.

Control Parameters. Because of the complexity and generality of RAV control model and program, a large number (hundreds) of configurable parameters exist in the control program. Many of them are dynamically adjustable at run-time via the GCS command interface. For example, in the ArduPilot software suite [22], there are 247 configurable control parameters, including 111 parameters for the x-, y-axis controller, 119 for the z-axis controller, 29 for the roll controller, 29 for the pitch controller, 30 for the yaw controller, 103 for motor control, and 40 for mission specification. We note that, while the total number of the parameters is 247, some of the parameters are shared by multiple controllers. When receiving a GCS command to adjust one of these parameters, the control program is supposed to perform an input validity check to determine if the new value is within the safe range of that parameter. Unfortunately, such a check may be missing or based on an erroneous value range.

3.2 Attack Model

Attack Model and Assumptions. Attacks that exploit cyber-physical bugs are characterized as follows: Knowing an adjustable control parameter with incorrect or missing range check logic in the control program[1], the attacker concocts and issues a seemingly innocent – but actually malicious – parameter-change GCS command to the victim RAV. Without correct input validation, the illegitimate parameter value will be accepted by the control program and cause at least one of the RAV's 6DoF controllers to malfunction – either immediately or at a later juncture, inflicting physical impacts on the RV. When planning an attack, the attacker may also opportunistically exploit a certain environmental condition (e.g., strong wind) under which a parameter-change command would become dangerous. For example, he/she might wait for the right wind condition (e.g., by following weather forecast) to launch an attack with high success probability. Such a case will be presented in Section 3.5.5.

The attacker can be either an external attacker or an insider threat. In the case of an external attacker, we assume that he/she is able to perform GCS spoofing to issue the malicious command, which is justified by the known vulnerabilities in the wireless/radio communication protocols between RAV and GCS [78]–[82]. In the case of an insider threat,

[1]↑The attacker may acquire such knowledge via a program vetting tool (such as RVFuzzer).

we assume that the insider is a rogue RAV operator (not a developer), who does not have access to control program source code and cannot update the control program firmware.

Attack Model Justifications. Our attack model is realistic (and attractive) to attackers for the following reasons: (1) Such an attack incurs a very small footprint – just one innocent-looking command, without requiring code injection/trojaning, memory corruption, or sensor/GPS spoofing; (2) The attack can still be launched even after the control program has been hardened against traditional software exploits [10], [30], [83], [84]; (3) The attack looks like an innocent "accident" because the malicious parameter value passes the control program's validity check. In some cases (i.e., range specification bugs), it is even in the valid range set in the control program's specification.

Why would the attacker bother to manipulate control parameter values, instead of just taking control of, or crashing the vehicle? A key observation provides the answer: If the attacker is not aware of – and hence does not manipulate – illegitimate-but-accepted control parameter values, it would actually *not* be easy to disrupt or crash an RAV with minimum footprint[2]. This is because both the RAV control program and control model already achieve a level of robustness for the RAV to resist being commanded into instability or danger: The control program can identify and reject many illegitimate commands; and the control model can filter or mitigate the impacts of some commands that escape the control program's check [44], [45]. Moreover, an internal attacker is also motivated to exploit illegitimate control parameter values that are erroneously considered normal in the RAV's specification (i.e., range specification bugs), as the attacker could evade attack investigation by claiming that he/she was following RAV control specification when issuing the command in question.

We do acknowledge that there exist scenarios where attackers can successfully launch attacks without exploiting cyber-physical bugs. For example, an insider could hijack an RAV by changing its trajectory, when working alone without a co-operator (who might otherwise catch the attack in action).

[2]↑The minimum footprint would help avoid detection before the attack succeeds.

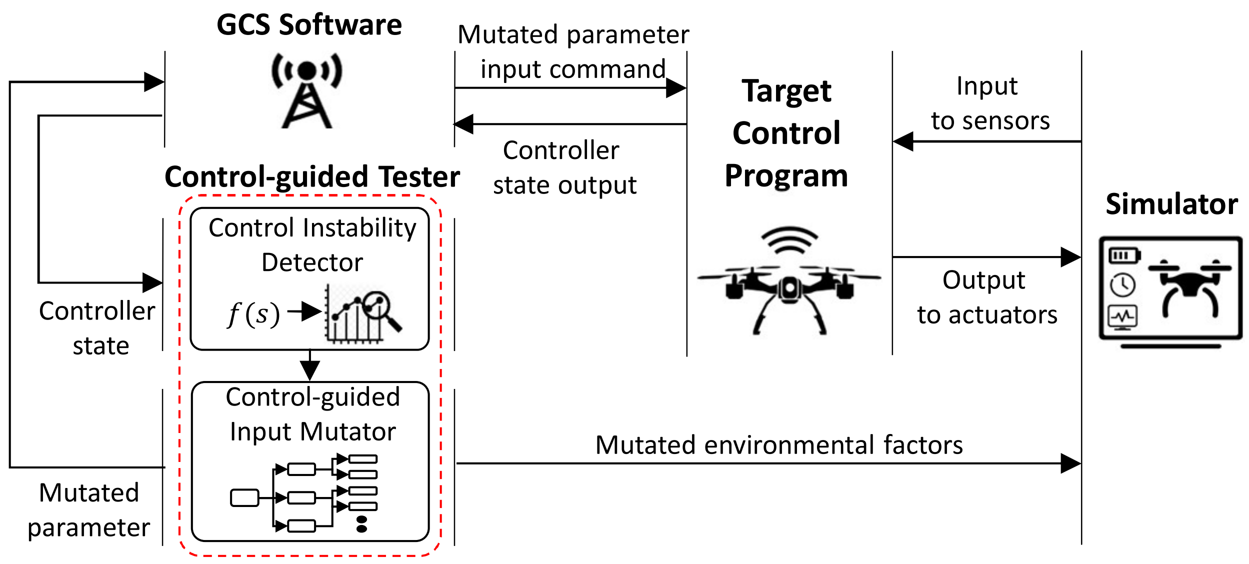


Figure 3.1. Overview of RVFuzzer.

3.3 RVFuzzer Design

In this section, we present the design of RVFuzzer. We first give an overview of RVFuzzer's architecture (Section 3.3.1) and then present detailed design of two key components of RVFuzzer: (1) the control-guided instability detector that monitors the vehicle's controller state to detect controller malfunction (Section 3.3.2) and (2) the control-guided input mutator that generates control program inputs for efficient program testing (Section 3.3.3).

3.3.1 Overview

RVFuzzer is designed to (1) detect physical instability of the RAV during testing and (2) generate test inputs iteratively to achieve high testing efficiency and coverage. Figure 3.1 presents an overview of RVFuzzer, which consists of four main components: a GCS program, the subject control program, a simulator, and a control-guided tester – the core component of RVFuzzer. The roles of the first three components are as follows: the GCS software is responsible for issuing RAV control parameter-change commands; the subject control program, as the testing target, controls the operations of the (simulated) RAV; and the simulator emulates the physical vehicle and its operating physical environment. We note that (1) the GCS and RAV control programs are from real-world GCS and RAV; and (2) our simulators [55], [85] are widely adopted for robotic vehicle design and testing.

RVFuzzer's control-guided tester consists of two sub-modules: (1) control instability detector and (2) control-guided input mutator. During testing, the control instability detector detects non-transient controller anomalies of the target RAV (e.g., crash and deviation), as indication of control program execution anomaly caused by an cyber-physical bug. The control-guided input mutator is a feedback-driven input mutator for efficient mutation of control parameter and environmental factor values. Using the results of the control instability detector as feedback, the mutator adaptively mutates control parameter values via a well-defined RAV control interface (i.e., GCS commands created and issued by the GCS software). In addition, it mutates environmental factors (e.g., wind) by re-configuring the simulator.

3.3.2 Control Instability Detector

The goal of the control instability detector is to continuously monitor RV controller state to determine if a specific GCS command has induced non-transient controller anomaly. Such a controller anomaly can be considered as an indication of an cyber-physical bug. We note that cyber-physical bugs may not lead to program crash, a common indicator of traditional bugs (e.g., memory corruption).

We first define a rule to detect controller anomalies, which is tailored for cyber-physical bugs. We then describe the mechanism to monitor the RVFuzzer's 6DoF controller states for detecting such a controller anomaly.

Indication of Controller State Deviation Exploitation of an cyber-physical bug will cause an RAV's failure to stabilize its controller states and/or complete its mission. To accurately detect bug-induced controller anomaly, RVFuzzer must be equipped with the capability of controller state deviation detection. Among the possible controller anomalies experienced by an RAV, there are two types of controller state deviation: (1) vehicle state deviation and (2) reference state deviation. Accordingly, we define a detection rule to determine if one of the two types of controller state deviation has occurred.

The first type – vehicle state deviation – is the case where a controller (e.g., the primitive x-axis velocity controller) fails to stabilize its vehicle state ($x(t)$) according to its reference state ($r(t)$). In the theoretical control model, a controller always tries to keep $x(t)$ close to $r(t)$ (Section 3.1). Consequently, if the difference between $x(t)$ and $r(t)$ keeps increasing and exceeds a certain threshold, the vehicle state will be considered deviating from the reference state. To quantify the vehicle state deviation, we leverage the integral absolute error (IAE) formula [52] which is widely used as a stability metric in control systems.

$$deviation(t) = \int_{t}^{t+w} \frac{|r(s) - x(s)|}{w} ds \tag{3.1}$$

Given a time window w and starting from a certain time instance t, the formula quantifies the level of deviation ($deviation(t)$). If $deviation(t)$ is larger than a pre-determined threshold τ, our rule will determine that there is a controller state deviation starting at t. We will describe how to experimentally determine w and τ for each 6DoF controller state in Appendix 3.4.1.

The second type – reference state deviation – is the case where an RAV deviates from its given mission. A controller is expected to adjust its reference state to track its mission. If a controller fails to do that, it is considered malfunctioning. To detect such a deviation, our rule will check whether the difference between the reference state and the mission target becomes persistently greater than a threshold.

We note that our detection rule only considers non-transient controller state deviation. An RV may experience transient controller state deviation during normal operation but can effectively recover from it, thanks to the robustness features of the controllers such as the extended Kalman filter [86]–[88].

Control Instability Detection We now apply our "vehicle-reference" and "reference-mission" deviation determination rule to detect control instability. During a test mission, the control program readily logs all its 6DoF controller states (Section 3.1). The log data can be retrieved by the GCS software, which will then be accessed by the Control Instability Detector and applied to the evaluation of the detection rule (Fig.3.1). Note that the controller states include those of the three primitive controllers (for position, velocity, and acceleration control) inside each 6DoF controller; and each primitive controller logs its observed, reference and mission states. As such, the Control Instability Detector can apply the detection rule to detect controller state deviation at any primitive controller.

3.3.3 Control-Guided Input Mutator

A software testing system needs to judiciously generate program inputs to achieve high bug coverage while reducing the number of the subject program's test runs. In other words, the set of generated testing inputs should be representative to produce the same or similar results when other untested inputs were provided to the program. We first define RVFuzzer's input mutation space (i.e., types and value ranges of dynamically adjustable control parameters). We then describe our control-guided input mutation strategy to generate representative testing inputs, with consideration of environmental factors that affect the RAV operation and control.

Our input generation method considers both control parameters and environmental factors[3]. For control parameters, we first define their value mutation spaces. We then present the feedback-driven input mutator which generates a reduced set of control parameter-change test inputs. The mutator also mutates the external environmental factors and tests the control program under different combinations of input control parameter values and environment factor values.

Control Parameter Mutation Space

The input mutation space of the subject control program consists of: (1) the list of dynamically adjustable control parameters, (2) the range of all possible values for each parameter, and (3) the default value of each parameter.

The list of control parameters is obtained from the specification of control program and the GCS command interface. We note that this is public information even for a close-source control program. The three most popular control software suites (i.e., ArduPilot [22],

[3]↑Environmental factors are not program input but physical context in which the RAV operates.

PX4 [23], and Paparazzi [24]) all support a common parameter tuning interface defined in MAVLink [48], the de facto protocol for RAV-GCS communications.

The value ranges of control parameters can be decided (1) by the data type of the control parameter and (2) by polling the control program itself. For a control parameter, its data type generically sets its value range. For example, the range of a 32-bit integer parameter is $[-2^{31}, 2^{31}-1]$. Interestingly, the ranges of many control parameters can be narrowed by polling the control program. This can be done by first sending a parameter-change command with a very large/small value; and then querying the actual value of that parameter, which now becomes the maximum/minimum value of the parameter defined in the control program. While the possibility of such a probe is specific to control program implementation, we do observe such implementation logic in ArduPilot and PX4.

The mutator also selects a default value within the range of each control parameter. Such a default value will be used in the input space search during mutation. We note that the set of default values of control parameters is normally made available by RAV vendors (e.g., 3DR, DJI, and Intel), as a guidance to RAV users when tuning the control parameters.

Feedback-Driven Parameter Input Mutator

RVFuzzer's input mutator accepts two inputs: the control parameter mutation space and the result of the Control Instability Detector from the previous run of the control program. The output of the mutator is the testing input for the program's next run. The efficiency of the control program vetting process depends on how well the mutated inputs are generated to trigger cyber-physical bugs without launching too many program test runs with different inputs. To explain our mutation strategy and methods, we first introduce the underlying intuition of our strategy and then describe our feedback-driven testing process with two steps: one-dimensional mutation and multi-dimensional mutation.

Input Space Reduction Strategy. The purpose of RVFuzzer is to find vulnerable – i.e., illegitimate but accepted – values for each dynamically adjustable control parameter. However, it is infeasible to test all possible values of a parameter. To improving testing efficiency, RVFuzzer must be able to selectively skip certain ranges of parameter values, if they lead to the same or similar outcome as the tested values. The value range-skipping idea is feasible thanks to the following observation: When control instability starts to be observed while increasing (decreasing) the value of a control parameter, further increase (decrease) of the parameter value will only maintain or intensify the instability.

We note that the aforementioned observation is generally valid. More specifically, in a control model, controllers and filters can be lumped together as part of its dynamics. Based on Root Locus [89], the trajectory of the loci always follows some asymptote. Hence, the change of a parameter will cause a *monotonic* change in stability. Sensor calibration can be considered as a constant controller anomaly, which will cause system response to degrade as the magnitude of the anomaly increases. Mission parameters will have different effects: Some can be grouped as part of the dynamics based on Root Locus; Some others, such as angle limitations, could cause an excessive response that introduces undesirable overshoot. This can be viewed as an integral windup, with a larger limit causing a larger overshoot.

Based on this observation, we propose two features for the mutator. (1) It will report valid/invalid value ranges — not individual values. Such a range will have a lower (minimum) and upper (maximum) bound. Any parameter value outside the range will cause control instability. (2) The mutator will be driven by feedback from the Control Instability Detector (Section 3.3.2) to determine the next testing input. Such feedback-driven mutation will be able to skip certain parameter value ranges for efficiency.

One-dimensional Mutation. In the first step of control software vetting, RVFuzzer's input mutator determines the valid/invalid range for each control parameter *independently*. The mutator isolates the impact of the target parameter on the controller state deviation by setting the values of all other parameters to their *default* values.

We present the one-dimensional mutation procedure in Algorithm 3. For each target control parameter, the mutator determines the upper and lower bounds of the valid value range by utilizing a mutation-based binary search method. We elaborate the method (Algorithm 3) to find the upper bound of the valid range as follows. We note that the mutator follows a syntactically similar method to find the lower bound of the valid range.

To find the upper bound, the mutator will iteratively launch test runs, using the binary search method to set the next run's *input value* and to update the *working range.* It will set the initial *min-limit* of the working range as the default value of the target parameter; the initial *max-limit* of the working range as the maximum possible value of the target parameter; and the initial input value as the mid-point between the *min-limit* and *max-limit* values. Thereafter, in each run, the mutator obtains the output of the Control Instability Detector under the current input value, and updates the working range in the next run by considering the following two cases based on the detector's output (Line 14).

- **Case 1** (Line 17-18): If the mutator observes that the current input value does not cause any deviation, it skips the lower half of the working range in the next run and sets the new *min-limit* as the current input value. This decision is justified by our earlier observation on the monotonic property of control instability. For the next run, the mutator will again set the new input value as the mid-point between *min-limit* and *max-limit.*

- **Case 2** (Line 15-16): If the current input value leads to controller state deviation, the mutator concludes that there are other values lower than the current input value which can also cause deviation. Hence, for the next run, the mutator will skip the upper half of the working range by setting *max-limit* as the current input value and the new input value as the mid-point between *min-limit* and *max-limit.*

We highlight that, after each run, the mutator skips the values corresponding to one half of the working range. This input space reduction strategy ensures that the mutator covers all possible values of the target control parameter efficiently. After determining the working range for the next run, the mutator sets the input value for the next run as the mid-point of the new working range (Line 19), following the binary search method. The mutator continues the (detector) feedback-driven search method, until the difference between the input values in the current and the next runs is less than a pre-determined threshold $MinDiff$ (Line 20). Finally, the mutator determines the valid value range and the corresponding vulnerable value range (i.e., invalid range) for the target control parameter.

Algorithm 3 One-dimensional Mutation.

Input: Input mission (M), input parameter (P), test environmental factor (E), controller state deviation threshold set for all primitive controllers (τ)
Output: An invalid range for a target parameter (R)

```
1: function OneDimensionalMutation($M$, $P$, $E$, $\tau$)                    ▷ Main function
2:   Initialize $R$
3:   $R.max \leftarrow$ OneMutation($M, P, E, \tau, U$)                         ▷ 'U': Upper-bound search
4:   $R.min \leftarrow$ OneMutation($M, P, E, \tau, L$)                         ▷ 'L': Lower-bound search
5:   return $R$                                            ▷ Return an invalid range of one parameter
6: function OneMutation($M$, $P$, $E$, $\tau$, $bound$)
7:   if $bound = U$ then                                   ▷ 'U' indicates an upper-bound search
8:     $\{test, max\text{-}limit, min\text{-}limit\} \leftarrow \{(P.Max - P.Default)/2, P.Max, P.Default\}$
9:   else                                                  ▷ 'L' indicates a lower-bound search
10:    $\{test, max\text{-}limit, min\text{-}limit\} \leftarrow \{(P.Default - P.Min)/2, P.Default, P.Min\}$
11:   $MinDiff \leftarrow 0$
12:   do
13:     $test \leftarrow test$                               ▷ Store the testing value before mutation
14:     $Dev \leftarrow$ RunAndDeviationCheck($M, P, test, E, \tau$)
15:     if ($bound = U$ and $Dev = True$) or ($bound = L$ and $Dev = False$) then
16:       $max\text{-}limit \leftarrow test$                                   ▷ Change the testing range
17:     else
18:       $min\text{-}limit \leftarrow test$                                   ▷ Change the testing range
19:     $test \leftarrow (max\text{-}limit + min\text{-}limit)/2$               ▷ Mutate the testing value
20:   while $|test - test| > MinDiff$                                  ▷ Check the exit condition
21:   return GetInvalidRange($test, test, bound, Dev$)
```

Multi-dimensional Mutation. RVFuzzer also performs a more advanced form of input mutation: multi-dimensional mutation, which finds extra invalid parameter value ranges that one-dimensional mutation may not find. Such extra invalid parameter values are introduced because a target control parameter may have *dependencies* on other parameters. In other words, different (non-default) setting of such other parameters may expand the invalid range of the target parameter.

To test the impact of other parameters (P_{others}), RVFuzzer performs the multi-dimensional mutation for each target parameter (P_{target}) as described in Algorithm 4. In this algorithm, RVFuzzer utilizes the results from the one-dimensional mutation (Algorithm 3) of all control parameters (P_{all}) (i.e., the lower and upper bounds of their valid ranges). For the target parameter, RVFuzzer sets the initial working range as its valid value range obtained from one-dimensional mutation (Line 2). Thereafter, the mutation of the values of the other parameters (Line 8-15) and the target parameter (Line 18-21) are performed recursively.

In each recursion, the value of each of the other parameters is mutated among only three values: the default value, the lower bound of its valid value range and the corresponding upper bound (Line 11). We note that setting the values of one/more of the other parame-

Algorithm 4 Multi-dimensional Mutation.

```
Input: Input mission ($M$), target testing input parameter ($P_{target}$), a set of all input parameters including one-dimensional search results ($PS_{all}$), test environmental factor ($E$), controller state deviation threshold set for all primitive controllers ($\tau$)
Output: An invalid range for a target parameter ($R$)
 1: function MultidimensionalMutation($M$, $P_{target}$, $E$, $PS_{all}$, $\tau$)        ▷ Main function
 2:   $R \leftarrow$ GetInvalidRange($P_{target}$)        ▷ Results from the previous step
 3:   $PS_{others} \leftarrow PS_{all} - \{P_{target}\}$        ▷ A set of other parameters except for $P_{target}$
 4:   $PS_{mut} \leftarrow \emptyset$        ▷ Initialize the mutated parameter set
 5:   $R \leftarrow$ DepMutation($M, P_{target}, E, PS_{others}, PS_{mut}, R, \tau$)
 6:   return $R$        ▷ Return a new invalid range
 7: function DepMutation($M$, $P_{target}$, $E$, $PS_{others}$, $PS_{mut}$, $R$, $\tau$)
 8:   if $PS_{others} \neq \emptyset$ then        ▷ Recursively mutate $PS_{others}$
 9:     $P_{mut} \leftarrow PS_{others}.Pop()$
10:     $PS_{mut} \leftarrow PS_{mut} \cup P_{mut}$
11:     for $PV \in P_{mut}.Min, P_{mut}.Default, P_{mut}.max$ do
12:       $PS_{mut} \leftarrow$ UpdateMutatedValue($PS_{mut}, P_{mut}, PV$)
13:       $R \leftarrow$ DepMutation($M, P_{target}, E, PS_{others}, PS_{mut}, \tau$)
14:   else        ▷ Update the invalid range of $P_{target}$ if all of $PS_{others}$ are mutated
15:     $R \leftarrow$ DepTest($M, P_{target}, E, PS_{mut}, R, \tau$)
16:   return $R$
17: function DepTest($M$, $P_{target}$, $E$, $PS_{mut}$, $R$, $\tau$)
18:   ParameterSet($PS_{mut}$)        ▷ Configure parameters with values of $PS_{mut}$
19:   $Upper \leftarrow$ OneMutation($M, P_{target}, E, \tau, U$)        ▷ 'U': Upper-bound search
20:   $Lower \leftarrow$ OneMutation($M, P_{target}, E, \tau, L$)        ▷ 'L': Lower-bound search
21:   return UpdateInvalidRange($R, Upper, Lower$)
```

ters to their lower/upper bound values leads to an extreme scenario which can potentially exacerbate the impact of the target parameter on the controller state deviation.

After setting the values of the other parameters (Line 18), the mutator follows a procedure similar to the one-dimensional mutation. It employs the mutation-based binary search method to determine and update the lower and upper bounds of the valid value range of the target parameter (Line 20-21). The new (in)valid range is then updated (Line 21).

In essence, as RVFuzzer mutates the values of multiple control parameters together, it can identify additional values of the target parameter that will cause controller state deviation under specific value setting of the other parameters. If such invalid values lie outside the one-dimensional invalid value range, the multi-dimensional mutation will conditionally expand the invalid value range to include those values, subject to the setting of the other parameters. As such, the result of the multi-dimensional mutation can be considered as an incomplete set of constraints on the values of multiple control parameters.

Environmental Factors

In real-world missions, the RAV interacts with the physical environment with external factors such as physical obstacles and wind. Such factors influence RAV's controller state and performance. We note that an external factor (e.g., wind) could make an otherwise valid parameter value cause controller state deviation. This means that such values can be exploited by attackers. To detect such influence, RVFuzzer mutates and simulates the impact of environmental factors along with multi-dimensional mutation of parameter values. We categorize the environmental factors into two types: geography and disturbances.

Typical geographical factors of interest are obstacles encountered by an RAV during its missions. The RAV will need to take actions to avoid such an obstacle. The actions may entail changes in the parameter values to enable a change of trajectory. This may expand the invalid range of the parameter values that will cause controller state deviation. To expose such cyber-physical bugs, RVFuzzer defines and simulates RAV missions in which the RAV needs to avoid obstacles via sudden, sharp trajectory changes. An attack case triggered by obstacle avoidance will be presented in Section 3.5.5.

External disturbances such as wind and turbulence may also disrupt the RAV's operation. RVFuzzer simulates the wind gusts and mutates the wind speed and direction based on real-world wind conditions. Details of the wind factor setup are given in Section 3.5.4. The attack case presented in Section 3.5.5 also exploits the wind condition.

3.4 Implementation

To evaluate RVFuzzer experimentally, we have implemented a prototype of RVFuzzer. The implementation details of its main components are described as follows.

Subject Control Programs. We choose the quadcopter as our subject vehicle as the quadcopter operates in all of the 6DoF and it is one of the most widely adopted types of RAVs [90]–[92]. We point out that the implementation of RVFuzzer is not specific to a certain RAV type or model as RVFuzzer only needs the physical quantities (e.g., weight and inertial parameters) and the corresponding simulator to support a vehicle. This means that RVFuzzer can be reconfigured to support other types of RAVs, such as hexacopters and rovers.

We apply RVFuzzer to vet two control programs that both support the quadcopter: ArduPilot 3.5 and PX4 1.8. The default vehicle control model supported by both programs is that of the 3DR IRIS+ quadcopter [76]. All vetting experiments (on both ArduPilot and PX4) are performed using a desktop PC with quad-core 3.4 GHz Intel Core i7 CPU and 32 GB RAM running Ubuntu 64-bit.

Simulator. To simulate the physical vehicle and environment, we utilized the APM simulator [55] and Gazebo [85], [93] for ArduPilot and PX4, respectively. We note that RVFuzzer's control instability detection and input mutation functions can easily interoperate with these simulators via the interfaces between the simulators and the control and GCS programs.

GCS Program. We used QGroundControl [94] and MAVProxy [95] as the ground control station software for PX4 and ArduPilot, respectively.

Control-Guided Tester. The control-guided tester is the core component of RVFuzzer. It is written in Python 2.7.6 with 5,722 lines of code. To implement the key functions in RVFuzzer, we leveraged the *Pymavlink* library [96], which provides APIs to remotely control the RV via the MAVLink communication protocol [48]. MAVLink is the de-facto communication protocol for robotic vehicles, which is used not only by ArduPilot and PX4, but also by other platforms such as Paparazzi [24], DJI [77], and LibrePilot [97]. MAVLink supports a wide range of GCS commands (e.g., for mission assignment, run-time controller state monitoring, and parameter checking and adjustment) that are leveraged and tested by RVFuzzer.

To test the control performance of the subject vehicle, we adopted the *AVC2013* [98] mission which is an official mission provided by ArduPilot and used in autonomous vehicle competitions to test the control and mission execution capabilities of RAVs. To improve the testing efficiency of RVFuzzer, we adjusted that mission by removing the overlapping flight courses, reducing the distance between each pair of waypoints, and increasing the vehicle's velocity.

To classify and generate the bug discovery results, we leverage a list of dynamically adjustable control parameters provided by ArduPilot and PX4 [99], [100]. Such a list is usually provided in the Extensible Markup Language (XML) format in the source code and can be easily parsed.

3.4.1 Thresholds for Controller State Deviation

We present how to determine the threshold values used by our control instability detector to detect controller state deviation (Section 3.3.2). We use the thirty other experimental missions in our experiments, similar to existing work [12]. Specifically, the thresholds are determined by applying the three-sigma rule [53] on the top deviation values. For the time window (w) in the IAE formula, we set it to the duration of each mission segment (i.e., flight segment between two consecutive waypoints) within a mission. The list of the threshold values that we use for each controller state is presented in Table 3.1.

We note that we do not monitor controller state deviation in the second derivative states of the 6DoF (i.e., acceleration of any of the 6DoF). This is because, if their vehicle states are oscillating, they can potentially cause false positives. In fact, for the same reason, some control programs do not control acceleration in some 6DoF controllers (e.g., ArduPilot does not control the angular acceleration of roll, pitch, and yaw). However, RVFuzzer can detect their controller state deviation via the indirect impacts on the *dependent* states. The controller state deviation in the second derivative states are propagated to their integral states (e.g., the first derivative states of the 6DoF), as their controls are intrinsically related.

3.4.2 Physical Impacts Caused by Cyber-Physical Bug Exploitation

We present more details about the cyber-physical bugs found by RVFuzzer and the implications of the attacks that exploit them in Tables 3.2 (for ArduPilot) and Table 3.3 (for PX4). The columns of each table shows: (1) the control program modules where the bugs belong (Control Program Module), (2) the vulnerable control parameters (Parameter, i.e., with erroneous range specification or range implementation), and (3) the possible physical

Table 3.1. List of threshold values for each controller state.

Control Program	ArduPilot	PX4
Latitude/Longitude Position	11.62 m	10.08 m
Latitude/Longitude Velocity	1.23 m/s	4.71 m/s
Altitude Position	2.06 m	3.43 m
Altitude Velocity	0.26 m/s	2.28 m/s
Roll	2.66 deg	5.56 deg
Roll Rate	2.83 deg/s	3.68 deg/s
Pitch	4.64 deg	18.66 deg
Pitch Rate	10.67 deg/s	15.35 deg/s
Yaw	4.13 deg	21.57 deg
Yaw Rate	16.24 deg/s	14.69 deg/s

impacts caused by the attacks exploiting the bugs (Physical Impacts). While the two tables list a total of 63 parameters, some of the parameters are associated with *both* range implementation and specification bugs. This explains why the total number of bugs (89) is higher than the number of vulnerable parameters.

Depending on the specific (malicious) value of the control parameter, the impact of an attack may vary. Here the possible impacts are categorized into four types as shown in the four sub-columns of the "Physical Impacts" column: "C" – vehicle crash; "D" – deviation from trajectory; "U" – unstable vehicle movement; and "S" – vehicle getting "stuck" at a certain location or speed. All of these impacts are non-transient and cannot be recovered by the controllers.

3.5 Evaluation

We now present evaluation results from our experiments with the RVFuzzer prototype. The three main questions that we want to answer are: (1) How effective is RVFuzzer at finding cyber-physical bugs (Section 3.5.1); (2) How do different input mutation schemes of RVFuzzer contribute to the discovery of cyber-physical bugs (Section 3.5.4); and (3) How can RVFuzzer be applied to discover cyber-physical bugs that would otherwise be exploited to launch stealthy attacks (Section 3.5.5).

Table 3.2. Cyber-physical bugs in ArduPilot and the implications of the attacks exploiting them (C: Crash; D: Deviation from trajectory; U: Unstable movement; S: "Stuck" in certain location or speed).

Control Program Module	Parameter	Physical Impacts			
		C	D	U	S
Controller	PSC_POSXY_P	✓			✓
	PSC_VELXY_P	✓	✓	✓	
	PSC_VELXY_I		✓	✓	
	PSC_POSZ_P				✓
	PSC_VELZ_P	✓			
	PSC_ACCZ_P	✓			✓
	PSC_ACCZ_I	✓	✓	✓	
	PSC_ACCZ_D	✓	✓	✓	
	ATC_ANG_RLL_P	✓			
	ATC_RAT_RLL_I	✓			
	ATC_RAT_RLL_IMAX	✓			✓
	ATC_RAT_RLL_D	✓			
	ATC_RAT_RLL_P	✓		✓	
	ATC_RAT_RLL_FF	✓		✓	
	ATC_ANG_PIT_P	✓			
	ATC_RAT_PIT_P	✓		✓	
	ATC_RAT_PIT_I	✓			
	ATC_RAT_PIT_IMAX	✓			
	ATC_RAT_PIT_D	✓			✓
	ATC_RAT_PIT_FF	✓		✓	✓
	ATC_ANG_YAW_P	✓			
	ATC_SLEW_YAW			✓	
	ATC_RAT_YAW_P			✓	
	ATC_RAT_YAW_I			✓	
	ATC_RAT_YAW_IMAX				✓
	ATC_RAT_YAW_D	✓			✓
	ATC_RAT_YAW_FF	✓		✓	
Sensor	INS_POS1_Z	✓		✓	
	INS_POS2_Z	✓		✓	
	INS_POS3_Z	✓		✓	
Mission	WPNAV_SPEED				✓
	WPNAV_SPEED_UP				✓
	WPNAV_SPEED_DN				✓
	WPNAV_ACCEL	✓			✓
	WPNAV_ACCEL_Z	✓			✓
	ANGLE_MAX	✓			✓

3.5.1 Finding Cyber-Physical Bugs

We present a summary of the cyber-physical bugs discovered by RVFuzzer from ArduPilot and PX4. These bugs are the result of a 8-day, non-stop testing session running RVFuzzer on the two control programs.

Table 3.3. Cyber-physical bugs in PX4 and implications of attacks exploiting them.

Control Program Module	Parameter	Physical Impacts			
		C	D	U	S
Controller	MC_TPA_RATE_P	✓		✓	
	MC_PITCHRATE_FF	✓	✓	✓	
	MC_PITCHRATE_MAX	✓	✓		
	MC_PITCHRATE_P	✓	✓	✓	
	MC_PITCH_P	✓	✓	✓	✓
	MC_ROLLRATE_FF	✓	✓	✓	
	MC_ROLLRATE_MAX	✓	✓		
	MC_ROLLRATE_P	✓	✓	✓	
	MC_ROLL_P	✓	✓	✓	
	MC_YAWRATE_FF			✓	✓
	MC_YAWRATE_P			✓	✓
	MC_YAW_P			✓	✓
	MIS_YAW_ERR				✓
	MPC_TILTMAX_AIR		✓		✓
	MPC_THR_MAX	✓	✓	✓	
	MPC_THR_MIN	✓	✓	✓	
	MPC_XY_P	✓	✓		✓
	MPC_Z_P	✓	✓		✓
	MPC_XY_VEL_P	✓	✓	✓	✓
	MPC_Z_VEL_P	✓	✓		✓
Mission	MC_YAWRAUTO_MAX			✓	✓
	MPC_XY_VEL_MAX		✓		✓
	MPC_XY_CRUISE		✓		
	MPC_Z_VEL_MAX_DN		✓		✓
	MPC_Z_VEL_MAX_UP	✓	✓		✓
	MPC_TKO_SPEED				✓
	MPC_LAND_SPEED				✓

Classification of Cyber-Physical Bugs

The validity of an input value of a control parameter is checked based on the *specified* range that has been determined and documented by developers during the development of the control program. Our subject control programs (ArduPilot and PX4) have the specified ranges of all the control parameters publicly available on their developer community websites [99], [100]. Leveraging these public range specifications, RVFuzzer found a number of cyber-physical bugs through the 8-hour testing session. We classify these cyber-physical bugs into two categories based on their root causes: range implementation bugs and range specification bugs.

- **Range Implementation Bugs** Assuming that the specified valid range of a control parameter is correct, any value outside the specified range should be caught and rejected by the control program. If the implementation of the control program fails to enforce that, an out-of-range parameter value may be maliciously provided and accepted by the program, causing controller state deviations. This is the nature of the range implementation bug which, based on our observation, arises from a lack of or an incorrect implementation of range check logic in the program. To discover range implementation bugs, RVFuzzer employs the one-dimensional mutation strategy. It mutates the value of each target parameter and issues the parameter-change GCS command with the mutated value to the control program. If the Control Instability Detector reports a controller state deviation, RVFuzzer will report a range implementation bug associated with the target parameter.

- **Range Specification Bugs** Ideally, the specified valid range of a parameter should correctly scope the value of the parameter. Unfortunately, this turns out not always the case. To reveal such problems, RVFuzzer first performs one-dimensional mutation and then performs multi-dimensional mutation on each target parameter, determining its invalid value range that will cause controller state deviation. We observe that for some control parameters, their valid value ranges are erroneously specified by developers, allowing dangerous values in the specified – and subsequently implemented – ranges. This is the nature of the range specification bug. Based on our analysis, such bugs exist because a control program enforces a *fixed* valid value range for a control parameter, without considering three critical factors: (1) the difference between hardware models and configurations, (2) inter-dependencies between control parameters, and (3) impact of environmental factors. RVFuzzer reveals that the range of the valid input values of a target parameter tends to "shrink" under these factors, giving rise to range specification bugs.

Table 3.4. Summary of cyber-physical bugs found by RVFuzzer (RIB and RSB denote the number of range implementation and range specification bugs, respectively).

Module	Sub-module	ArduPilot		PX4	
		RIB	RSB	RIB	RSB
Controller	x, y-axis position	1	0	1	1
	x, y-axis velocity	2	1	1	1
	z-axis position	1	0	1	1
	z-axis velocity	1	0	1	0
	z-axis acceleration	3	0	0	0
	Roll angle	1	0	1	1
	Roll angular rate	5	0	3	3
	Pitch angle	1	0	1	1
	Pitch angular rate	5	0	3	3
	Yaw angle	1	0	2	2
	Yaw angular rate	6	0	3	3
	Motor	0	0	3	3
Sensor	Inertia sensor	3	3	0	0
Mission	x, y-axis velocity	1	1	2	0
	z-axis velocity	2	0	4	0
	z-axis acceleration	2	0	0	0
	Roll, Pitch	1	1	1	1
Total	-	36	6	27	20

3.5.2 Detection of Cyber-Physical Bugs

Table 3.4 summarizes the range implementation bugs (RIB) and range specification bugs (RSB) discovered by RVFuzzer in ArduPilot and PX4. The detailed list of the 63 control parameters that are affected by these bugs is presented in Appendix 3.4.2. For coherent presentation in Table 3.4, the control parameters in each of the two control programs are categorized into three modules (i.e., controller, sensor, and mission) and further into their sub-modules. Table 3.4 shows that RVFuzzer detected a total of 89 cyber-physical bugs (42 bugs in ArduPilot and 47 bugs in PX4). As a result, it turns out 87 cyber-physical bugs are new. We note that some of the control parameters are associated with *both* range implementation and the range specification bugs. Hence, the total number of cyber-physical bugs (89) is higher than the total number of affected control parameters (63).

We highlight that only two of the 89 bugs discovered by RVFuzzer were detected and correctly patched by the developers *before* we reported our results to them. Out of the remaining 87 bugs, the developers have so far independently confirmed 28 bugs and patched 11 of them. The delayed response of the developers brings forth an important point: Compared to the traditional "syntactic" bugs (e.g., buffer overflow), discovering, validating and patching cyber-physical bugs require more time and effort. This is because the exploitability of each cyber-physical bug must be fully verified under a spectrum of vehicle configurations and operating environments. In such a scenario, RVFuzzer can be utilized by developers as a helpful tool to automate the discovery and confirmation of cyber-physical bugs.

3.5.3 Impact of Cyber-Physical Bugs

We now detail the physical impacts (on the vehicle's operation) of the attacks that exploit the bugs found by RVFuzzer. We consider four levels of physical impact: crash, trajectory deviation, unstable movement, and frozen controller states. Appendix 3.4.2 presents possible physical impact(s) of attacks that exploit each of the vulnerable control parameters. Here, we summarize the results by analyzing the impact on the modules of the control program. Specifically, we present the causality of the bugs in a bottom-up fashion and assess its impact on the controller state deviation which is detected by RVFuzzer's Control Instability Detector.

Controller Module. Among the control parameters related to the controller module, RVFuzzer discovered 27 range implementation bugs and 1 range specification bug in ArduPilot, and 20 range implementation bugs and 19 range specification bugs in PX4 (Table 3.4). These bugs can be used to maliciously set invalid parameter values or exploit environmental factors, which would directly affect the primitive controllers and corrupt the controller states in the 6DoF. For example, if one of the control parameters related to the z-axis velocity is set to a value in the invalid range due to an cyber-physical bug, the manipulated parameter will corrupt the reference state of the (downstream) z-axis acceleration. As a result, the z-axis acceleration controller will attempt to bring its vehicle state closer to the corrupted reference state, which will cause control instability of the vehicle. Such instability may eventually lead to a crash.

Sensor Module. For this module, while RVFuzzer found 3 range implementation bugs and 3 range specification bugs in ArduPilot, it did not find any cyber-physical bug in PX4 (Table 3.4). We note that the vulnerable control parameters of the sensor module are related to either a sensor calibrator or a sensor filter for noise/disturbance. While the calibrator compensates for manufacturing errors in sensors and adjusts the vehicle state accordingly, the filter smooths out the sensor values and helps the controllers in robustly responding to physical interactions [101]. Hence, if an invalid value is assigned to a control parameter related to a sensor calibrator/filter due to an cyber-physical bug, the primitive controller that consumes the sensor values will compute a corrupted vehicle state. Such corruption will also propagate to its output reference state, and from there to other dependent primitive controllers, leading to unstable movement of the vehicle.

Mission Module. For this module, RVFuzzer discovered 6 range implementation bugs and 2 range specification bugs in ArduPilot, and 7 range implementation bugs and 1 range specification bug in PX4 (Table 3.4). Recall that this module is responsible for setting the mission parameters (e.g., speed and tilting angles) which define or adjust the vehicle's mission. However, if a parameter related to the mission module is manipulated with an invalid value by exploiting an cyber-physical bug, the corresponding controllers will generate misguided reference states. Such mission corruption will mislead one or more of the 6DoF controllers and prevent the vehicle from fulfilling its intended mission (e.g., not moving to the intended destination or at the intended speed), even if the vehicle does not experience any immediate danger.

3.5.4 Effectiveness of Input Mutation

RVFuzzer employs the control-guided input mutation strategy to generate control parameter value inputs and set environmental factors. We evaluate the effectiveness of this mutation strategy in enabling efficient discovery of cyber-physical bugs.

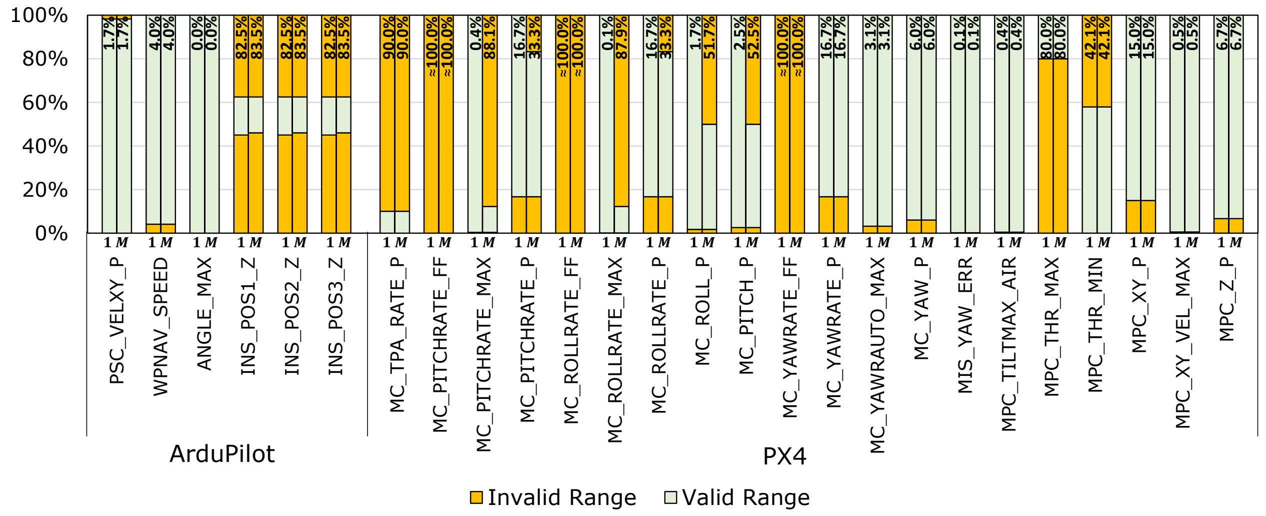


Figure 3.2. Invalid control parameter ranges discovered by RVFuzzer, normalized to the specified value ranges (**1**: One-dimensional mutation, ***M***: Multi-dimensional mutation). Percentage of invalid ranges (%) within the specified value ranges are noted at the top of the bars. [5]

Control Parameter Mutation

RVFuzzer discovers the range implementation bugs using the one-dimensional mutation strategy which detects the erroneous implementation of the parameter's range check logic. Through the extensive black-box-based (i.e., without source code) testing of the control parameters, RVFuzzer discovered a total of 63 range implementation bugs: 36 bugs in ArduPilot and 27 bugs in PX4.

To detect the incorrectly specified ranges of the parameters and find the range specification bugs, RVFuzzer employs one-dimensional mutation followed by the multi-dimensional mutation strategy. We demonstrate the effectiveness of RVFuzzer's mutation strategies in discovering the range specification bugs in Figure 3.2, which presents the valid and invalid value ranges (detected using one-dimensional and multi-dimensional mutation) for the affected control parameters.

- **One-dimensional Mutation** RVFuzzer discovered a total of 26 range specification bugs using one-dimensional mutation: 6 bugs in ArduPilot and 20 bugs in PX4 (Figure 3.2). For example, for parameter `MC_TPA_RATE_P` in PX4, the specified range was between 0 and 1, and the default value was 0. However, RVFuzzer detected controller state deviations with values between 0.1 and 1, and hence found 90% of the values in the specified range belonging to the invalid range. We note that almost 100% of the values in the specified range of the three parameters, `MC_PITCHRATE_FF`, `MC_ROLLRATE_FF` and `MC_YAWRATE_FF`, in PX4 are invalid. This is because, while each of these parameters can be independently configured with a wide range of input values, there is a smaller range of values that are valid when the other parameters take their default values.

- **Multi-dimensional Mutation** Recall that the multi-dimensional mutation further expands the invalid range of the target parameter to include the additional values that cause controller state deviation under specific, non-default settings of the other parameters. In Figure 3.2, we observe that the multi-dimensional mutation expands the invalid ranges of 10 out of 26 range specification bugs found using one-dimensional mutation. For instance, RVFuzzer found that the invalid range of the `MC_ROLL_P` parameter in PX4 was expanded from 1.7% to 51.7% when multi-dimensional mutation was employed. We highlight that for some parameters, RVFuzzer reported a significant increase of invalid range with multi-dimensional mutation. In particular, compared to the invalid ranges detected using one-dimensional mutation, the invalid ranges of the `MC_PITCHRATE_MAX` and `MC_ROLLRATE_MAX` parameters in PX4 increased from 0.4% to 88.1% and from 0.1% to 87.9%, respectively. These results demonstrate that the multi-dimensional mutation strategy can discover invalid values of control parameters with stronger awareness of the inter-parameter dependencies (discussed further in Section 3.6).

Environmental Factor Mutation

RVFuzzer further found that the invalid ranges of some control parameters expand when environmental conditions are taken into account. This is important because the developers may not completely consider the impact of various environmental conditions when specifying

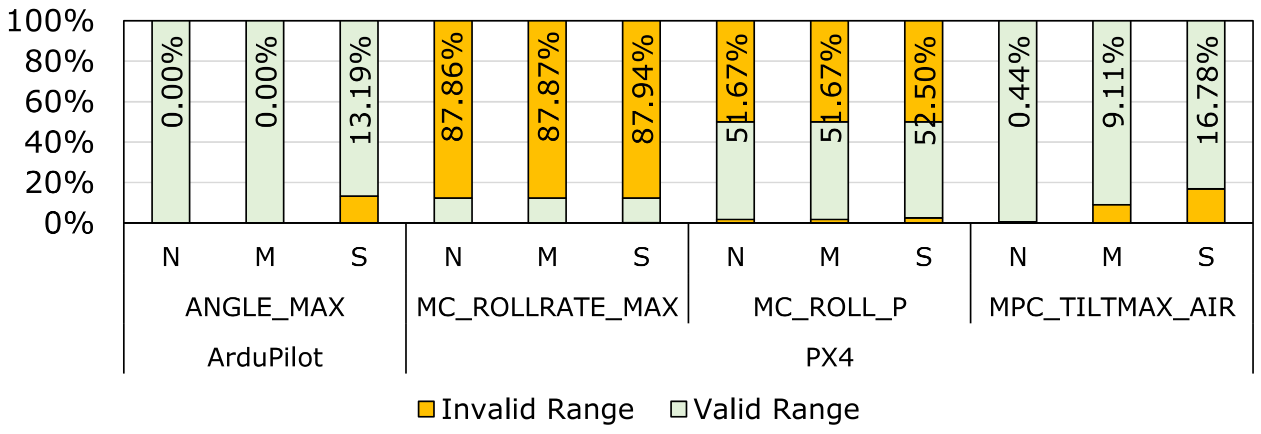


Figure 3.3. Normalized invalid ranges within the specified value ranges under different wind conditions (N: No wind, M: Medium wind, S: Strong wind).

the valid range of a parameter. Based on our observation, two factors may widen the invalid ranges: (1) geographical factor and (2) external disturbance (e.g., strong wind), as described in Section 3.3.3. RVFuzzer found four cases which can be exploited with realistic environmental factors.

We performed tests based on existing wind analysis statistics [102]–[104] and simulated various wind conditions. The wind conditions were divided into three categories: no wind, medium wind (with a horizontal wind component of 5 m/s or a vertical wind component of 1 m/s), and strong wind (with a horizontal wind component of 10 m/s or a vertical wind component of 3 m/s). For each wind condition, the wind gust was simulated from 0 to 360 degrees with 30-degree increments. Simulations were also performed where the wind gust was designed to come in at every 30-degree angle between the horizontal tests and the vertical tests, such that the tested wind vectors approximately formed an ellipsoid. These wind settings enrich our standard test mission (Section 3.4), which already reflects geographical factors as it emulates flight paths with sharp turns for obstacle avoidance.

Figure 3.3 presents the impact of three different wind conditions on the four parameters which cause controller state deviations. RVFuzzer discovered these four cyber-physical bugs using multi-dimensional mutation over the four parameters. We observe that the impact of environmental factors expands the invalid ranges of those parameters. In particular, when the wind conditions were not considered, `ANGLE_MAX` did not have any invalid range under both one-dimensional and multi-dimensional mutations. However, with wind conditions, RVFuzzer reveals that this parameter can be exploited when strong wind is present.

Such an cyber-physical bug is exploitable because a large angular change is required to alter the direction of the vehicle. Specifically, if the maximum allowed angle or angular speed is not large enough (even within the specified value ranges), the vehicle's motors cannot generate enough force to change the direction or resist the wind gusts. As a result, the vehicle may fail to change its direction at sharp turns or it might drift in the wind's direction in the worst case.

We note that the results with environmental factor mutation may be affected by other factors, such as the control model, configuration, and physical attributes (e.g., motor power and the size of the vehicle). For example, if the vehicle is capable of turning with a larger roll angle, has a smaller size, or has stronger motors, it may be able to resist wind gusts when changing its flight direction. Hence, these conditions need to be tested by RVFuzzer for each specific type of vehicle.

3.5.5 Case Studies

We present three representative case studies of cyber-physical bugs. We also discuss how an attacker can exploit these bugs, and how RVFuzzer can proactively discover them. The three cases cover different affected controllers, cause different impacts on the RAV, and require different components of RVFuzzer's testing techniques to detect. Specifically, the bug discussed in Case I (Section 3.5.5) affects the x and y-axes controllers and causes unrecoverable slowdown, but can be discovered by RVFuzzer using the one-dimensional mutation technique. Case II (Section 3.5.5) presents a bug that affects the pitch controller, leads to a crash, and can only be found via multi-dimensional mutation strategy. Finally, the bug in Case III (Section 3.5.5) adversely affects the roll controller and causes significant deviation from the assigned mission, but can be discovered by mutating an environmental factor (wind force).

```
#define WPNAV_WP_SPEED_MIN 100                    // Buggy code 2
#define WPNAV_WP_SPEED_MIN 20                     // Patched code 2
...
void AC_WPNav::set_speed_xy(float speed_cms){
-if(_wp_speed_cms>=WPNAV_WP_SPEED_MIN){          // Buggy code 1
+if(speed_cms>=WPNAV_WP_SPEED_MIN){              // Patched code 1
   _wp_speed_cms = speed_cms;
   _pos_control.set_speed_xy(_wp_speed_cms);
   ...
```

Listing 3.1 Cyber-physical bug case on x, y-axis mission velocity. The parameter can be dynamically changed by either a mission speed-change command or a speed parameter-change command.

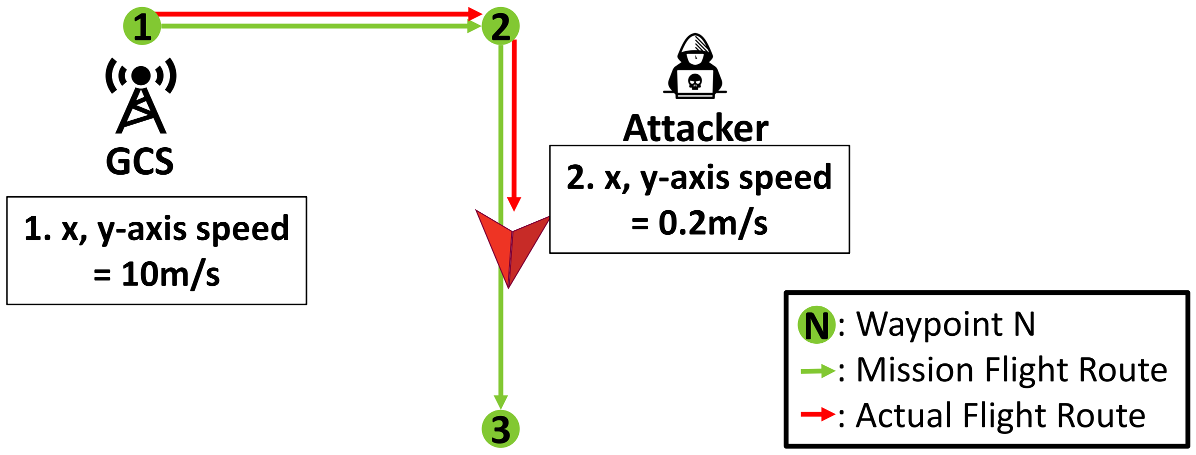


Figure 3.4. Illustration of Case Study I: An RAV cannot recover its normal speed for the segment from Waypoint 2 to Waypoint 3.

Case Study I: Bug Causing "Unrecoverable Vehicle Slowdown" Discovered by One-Dimensional Mutation.

We consider an RAV that is assigned the mission of express package delivery (Figure 3.4). Because of the urgency, the operator sets the RAV's mission speed to 10 m/s at Waypoint 1. During the mission, while the RAV slows down to make a turn at Waypoint 2, the attacker sends a seemingly innocent, but malicious, command to the RAV to change its mission speed to 0.2 m/s (the minimum *specified* speed is 0.2 m/s). After the turn, however, the operator will not be able to resume the 10 m/s mission speed by issuing speed-change commands. This attack exploits an cyber-physical bug in ArduPilot, illustrated in Listing 3.1.

- **Root Cause** Listing 3.1 presents the code that runs in the RAV when it receives a new speed-change input (denoted by `speed_cms`) during its mission. The specified minimum speed (in cm/s) is denoted by the `WPNAV_WP_SPEED_MIN` parameter (Line 1). We note that the *current* mission speed (denoted by `_wp_speed_cms`) is compared with the minimum mission speed (Line 5). This means that if (and only if) the current mission speed is equal to or higher than the minimum mission speed, it can be replaced by the new mission speed in the input command; If the current mission speed is lower than the minimum mission speed, it cannot be changed. Hence, this is the bug which can be exploited by the attacker, by sending a speed-change command with a value lower than the minimum mission speed while the current mission speed is higher than the minimum mission speed. This bug has been patched recently by the developers by correcting the value of the minimum mission speed (Line 2) and setting the comparison of the minimum mission speed with the input speed (Line 6).

- **Bug Discovery** This bug was discovered by RVFuzzer while performing one-dimensional mutation of the input mission speed parameter. For input mission values above 1 m/s, the RAV successfully changed its current mission speed. However, if the current mission speed dropped below 1 m/s, RVFuzzer can no longer change the current mission speed by setting the input mission speed parameter. The failure to change the current mission speed led to the incorrect execution of the mission, resulting in controller state deviation, simulated and detected by RVFuzzer. Hence, RVFuzzer reported this deviation-triggering parameter as an cyber-physical bug, which is confirmed by the related source code in Listing 3.1 (as ground truth of our evaluation).

Case Study II: Bug Causing "Oscillating Route and Crash" Discovered by Multi-Dimensional Mutation.

We consider an RAV that is assigned the same mission as in Case Study I. As shown in Figure 3.5, at Waypoint 2 of the mission, the attacker sends a malicious command to the RAV to change one of the four pitch control parameters: `MC_PITCH_P`, `MC_PITCHRATE_P`,

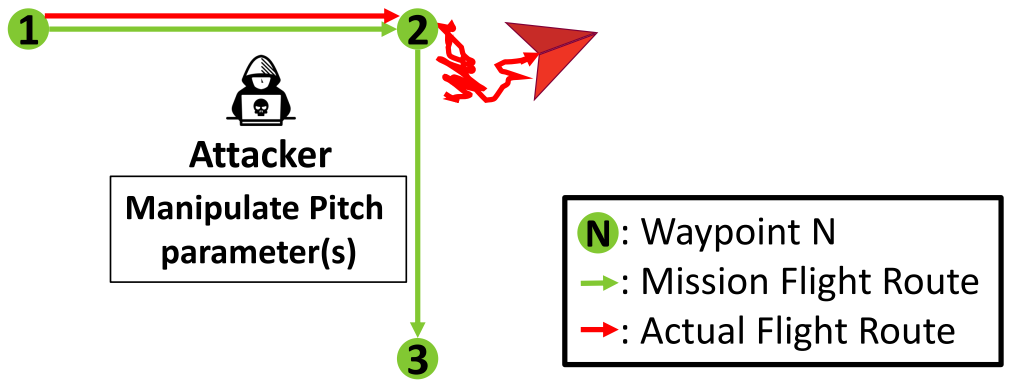


Figure 3.5. Illustration of Case Study II: The attack launched at Waypoint 2 causes an RAV to oscillate due to failing control of the pitch angle.

MC_PITCHRATE_P, and MC_PITCHRATE_FF. Because of the inter-dependency between these parameters, such a malicious command, which looks innocent, can cause the RAV to fail to stabilize its pitch angle, resulting in unrecoverable oscillation and deviation from its route.

- **Root Cause** The unrecoverable oscillation on the RAV's route is caused by the failure of its pitch controller to track the reference state of the pitch. The pitch controller utilizes four inter-dependent parameters: the P control gain of pitch angle (MC_PITCH_P), the P control gain of the pitch angular speed (MC_PITCHRATE_P), the maximum pitch rate (MC_PITCHRATE_MAX), and the feed-forward pitch rate (MC_PITCHRATE_FF). For example, a high value of MC_PITCHRATE_FF helps track the reference state of the pitch when MC_PITCH_P is low. When both MC_PITCHRATE_FF and MC_PITCH_P have high values, the RAV may perform overly aggressive stabilization operations. In that case, a low value of the maximum pitch rate (MC_PITCHRATE_MAX) is desirable to mitigate the impact of such operations.

  We point out that such dependencies can be exploited by an attacker to affect the RAV's operations by corrupting the value of just *one* parameter. Let us assume that the RAV is already configured with high values of MC_PITCHRATE_FF and MC_PITCH_P. If the attacker sets MC_PITCHRATE_MAX to a high value, the pitch controller will start to respond to the minuscule difference between the reference

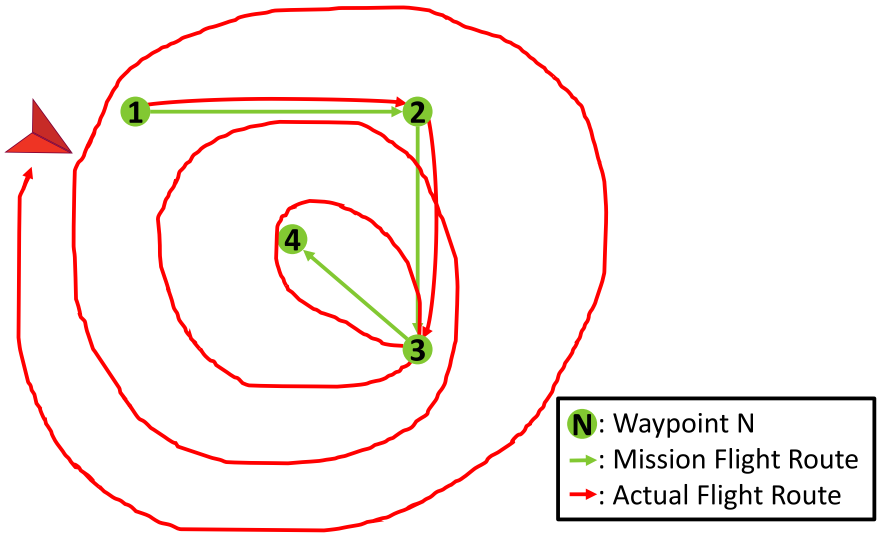


Figure 3.6. Illustration of Case Study III: An RAV fails to complete a simple mission from Waypoint 1 to Waypoint 4 due to the impact of environmental factors.

state and the vehicle state of the pitch angle with extreme sensitivity. As a result, the RAV will not be able to strictly follow its flight path. We note that this type of bug can only be discovered when the dependencies between multiple parameters are considered in the test.

- **Bug Discovery** This bug was found by RVFUZZER while performing multi-dimensional mutation (Algorithm 4) of the parameters related to the pitch controller. RVFUZZER mutated the target parameter (`MC_PITCHRATE_MAX`), while setting high values for `MC_PITCH_P` and `MC_PITCHRATE_FF` parameters. Unlike the one-dimensional mutation, which determined the parameter's valid range to be between 6.7 and 1800, the multi-dimensional mutation determined that the valid range of `MC_PITCHRATE_MAX` is to be between 6.7 and 220.1. RVFUZZER detected and reported the expanded invalid range of `MC_PITCHRATE_MAX` as an cyber-physical bug.

Case Study III: Bug Causing "Diverging Route" Discovered by Wind Force Mutation.

In this case study, we consider an RAV assigned a mission to deliver a food item to a customer via the path presented in Figure 3.6. The RAV is required to follow the path around tall buildings on a windy day with the wind direction towards the west. Since the item (e.g., soup) might spill if the RAV changes its attitude drastically, the operator tries to prevent sudden changes in the roll angle by limiting the maximum angular-change speed (`MC_ROLLRATE_MAX`) to a small value. When the vehicle is approaching Waypoint 2, the attacker sends a command to set the maximum tilting angle (`MPC_TILTMAX_AIR`) to a low value. We note that the RAV is supposed to make a 120-degree turn to avoid a tall building at Waypoint 3. However, the RAV fails to make the correct turn at Waypoint 3 and hence cannot reach the destination (Waypoint 4) after multiple attempts to correct the diverging path. We note that the value of the maximum tilting/roll angle parameter is accepted by the control program because it is within the *specified* valid range, yet the value causes controller state deviation due to the strong wind condition.

- **Root Cause** There are three causes that induce the vehicle's unexpected flight path divergence: (1) the mission route with sharp turns, (2) the roll controller's parameter value that is not responsive enough to change the direction in time, and (3) the strong wind that expands the invalid ranges of the roll controller's parameters. In this case study, the combination of these three factors disrupts the vehicle's maneuver and trajectory, resulting in a failed mission (and a hungry customer).

- **Bug Discovery** RVFuzzer discovered this bug in PX4 by mutating the wind condition during the AVC2013 mission (Section 3.4) which involves many sharp turns of the vehicle. As the input values of the roll controller parameters were mutated under a strong wind condition, RVFuzzer detected controller state deviation between the reference state and the mission (Figure 3.3). Hence, RVFuzzer reported this as an cyber-physical bug contingent upon the influence of an external factor (wind).

3.6 Discussion

Control Parameter Inter-dependencies. As revealed by multi-dimensional mutation, the control parameters may have dependencies on one another. A specific value of one parameter can increase or decrease the (in)valid value ranges of other parameters. The ground truth on such inter-parameter dependencies can only be derived from full knowledge about the underlying control model and the control program implementation, given the large number of control variables (including hundreds of parameters), the wide ranges of their values, and the influence from various environmental factors. As a result, it is challenging to fully and accurately capture the control parameter inter-dependencies, with only the binary of a control program. In this work, we consider the subject control program binary as a black box and take a pragmatic approach by only revealing *part of* such inter-dependencies. A more generic approach to control parameter dependency derivation – possibly based on source code and a formal control model – is left as future work.

Standard Safety Testing and Certification. For the safety of avionics software for airborne systems, there exist standard safety tests and software certifications such as DO-178B/C [105] and ISO/IEC 15408 [106]. To the best of our knowledge, however, there has been no standard safety testing framework created for RAVs. We believe that RVFuzzer's post-production, black-box-based (i.e., without source code) vetting will serve as a useful complement to standardized safety testing during RAV design and production.

4. DISPATCH: CONTROLLER SEMANTICS IDENTIFICATION AND INSTRUMENTATION FRAMEWORK FOR ROBOTIC AERIAL VEHICLE FIRMWARE

Thanks to the successful cyber-physical bugs detection through RVFuzzer and Mayday, we have reported a number of vulnerabilities to RAV control program developers. However, if RAV users/operators want to patch the legacy RAVs or closed-source binary firmware, there is a long-existing challenge: "How to patch the binary firmware without source code?" Patching cyber-physical bugs requires a mapping between the control model and the control (binary) firmware. Only with such a mapping, a binary patching framework can identify and patch the key vulnerable control variables (e.g., a z-axis velocity gain or a current altitude). However, there is no existing solution to identifying/locating controller components for binary firmware patching. To overcome the aforementioned challenge, we introduce an RAV binary firmware patching framework, DisPatch [20] in this chapter.

4.1 Background

RAV Control Model. DisPatch is a semantic decompiler and patch framework guided by a generic RAV control model. In general, RAV control models consist of the three following components: (1) sensor modules to measure the physical operations, (2) physical models depending on a number of factors such as shape, weight, sizes and motor powers, and (3) controller modules to actuate and adjust the propulsion engine. Components (1) and (2) are quantified via physical specification (e.g., motor specification, and shape of RAVs) and (3) are represented by control parameters (e.g., z-axis control gain) to properly adjust the controller states (e.g., z-axis speed and position of an RAV).

During a flight, sensor modules (e.g., a GPS, and a gyroscope) measure the controller states that are the physical movements along the six degrees of freedom (6DoF) as shown in Figure 2.1. These movements are continuously updated by the interaction between the physical world and the RAV. The RAV is defined and quantified by the physical RAV control model. Such a physical model encompasses multiple factors adapted by developers according to the environment factors and constraints. There are several *templates* of the RAV control models such as copter, plane, submarine, and helicopter. Those template models can be further customized, which introduces a variety of RAV control models. For example, a copter model encompasses a tricopter, quadcopter, and hexacopter with different configurations such as weight, propeller length, the number of motors, the combinations of motor rotation directions.

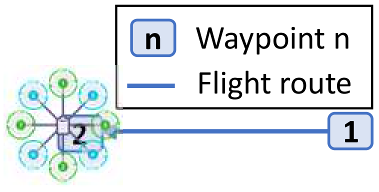
To control a physical model and its interaction with the physical world measured by the sensor modules, a controller algorithm works as the heart of the RAV control model orchestrating the rest of all the components. Among the multiple controller algorithms, the proportional-integral-derivative (PID) controller is the most commonly used one [107]–[109]. Multiple PID controllers are customized and integrated according to the physical specification of an RAV. To adapt the target RAV physical model, operators can configure control parameters via the (remote) ground control station (GCS) interface.

PID Controllers For RAVs. PID controllers play a vital role to control the vehicle. To guarantee stable physical operations, a set of PID controllers must be customized for each type of RAV. Figure 2.1 shows that an RAV controls physical movements along the 6DoF. Each *cascading controller* is responsible for controlling each degree of freedom such as the x-axis cascading controller as described in Figure 2.3. Zooming in the x-axis cascading controller, we can find three controllers (denoted as $c1(t)$, $c2(t)$, and $c3(t)$). Each of the three controllers is a *primitive* controller, and respectively computes the position, velocity, acceleration physical operations on the x-axis. Among primitive controllers, PID controllers are widely used in RAVs [22], [23], [110], mathematically expressed as follows:

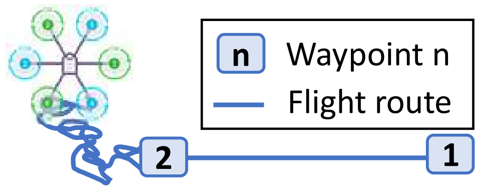
$$x(t) = P \cdot \mathrm{e}(t) + I \cdot \int_0^t \mathrm{e}(x)dx + D \cdot \frac{\mathrm{de}(t)}{\mathrm{d}t} + FF \cdot r(t) \tag{4.1}$$

$$\mathrm{e}(t) = r(t) - x(t) \tag{4.2}$$

In Equation 4.1 and 4.2, $x(t)$ denotes the *vehicle state*; $r(t)$ denotes the *reference* indicating the desired state; $\mathrm{e}(t)$ is thus the *control error*, which is the difference between the



(a) Stable flight with an octacopter.



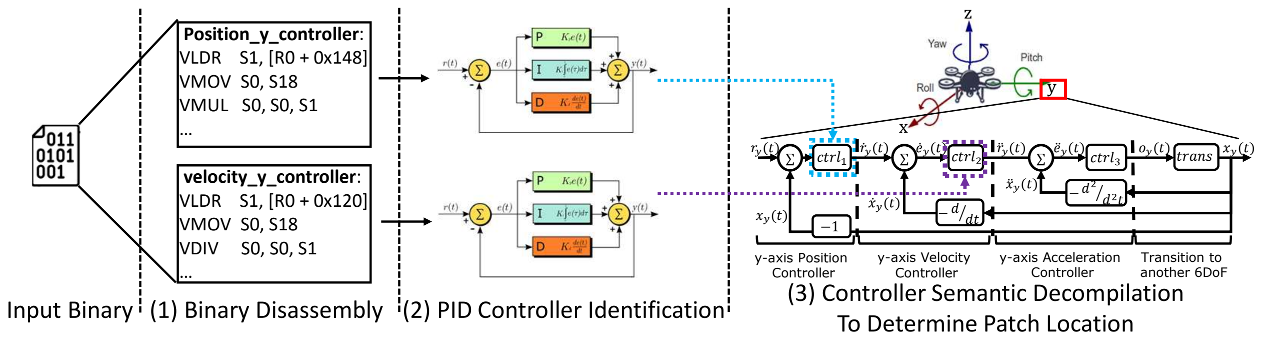
(b) Unstable flight with a hexacopter.

Figure 4.1. An example implication of x, y-axis velocity PID parameter configuration on the octacopter and hexacopter running on the identical copter firmware. Only hexacopter starts to show controller anomaly at Waypoint 2 within a few seconds after Waypoint 2 arrival.

vehicle state and the reference. In this mathematical expression, there are four predefined parameters (P, I, D, and FF) multiplying either e(t) or $r(t)$. The values of those *control parameters* are configured to properly control the target physical RAV model. Each controller generates the output as the input of another component connected. If it is connected to a primitive controller, $x(t)$ and $r(t)$ (generating e(t)) act as an input of another dependent primitive controller. We call such dependencies as *primitive controller dependencies*. Also, the output can be connected to a motor (e.g., $o_y(t)$) as a motor throttle or an input reference for another cascading controller (e.g., from the y-axis acceleration controller to the pitch angular controller). Such a connection between cascading controllers creates data dependencies in the RAV control model, which we call *cascading controller dependencies*. The entire controller structure consisting of six cascading controllers with their dependencies is described in Figure 2.2 and commonly used in RAVs [18].

4.2 Motivation and Threat Model

Control software developers have made RAVs more *robust* to overcome environmental changes. The most common defense mechanism is a filtering approach such as extended Kalman filter [44]–[46]. However, they are not designed for security or robust enough to defend against cyber-physical attacks in the real world as demonstrated by the previous research [18], [19]. As we mentioned earlier, the root cause of cyber-physical bugs within RAV firmware is that control program developers try to generalize the firmware implementation



```
Position_y_controller:
VLDR   S1, [R0 + 0x148]
VMOV  S0, S18
VMUL   S0, S0, S1
...
```

```
velocity_y_controller:
VLDR   S1, [R0 + 0x120]
VMOV  S0, S18
VDIV    S0, S0, S1
...
```

011
0101
001
Input Binary
(1) Binary Disassembly
(2) PID Controller Identification
P
I
D
z
Yaw
Pitch
Roll
x
y
ctrl1
ctrl2
ctrl3
trans
−1
y-axis Position Controller
y-axis Velocity Controller
y-axis Acceleration Controller
Transition to another 6DoF
(3) Controller Semantic Decompilation To Determine Patch Location

Figure 4.2. The necessity of identifying semantics of controller functions to solve **Challenge 1 and 2**. Given the binary, we show three example results: (1) an example of disassembly result, (2) an example of generic controller identification, and (3) patch payload and location determination by identifying semantics of controller functions.

to support as many control models as possible thus leaving room for attack in the control algorithm against specific RAV models, or that RAV operators need to replace a component with a different hardware specification as the previous one thus invalidating the previous setting of the control algorithm.

As an example, consider ArduPilot's support for hexacopter and octacopter. Despite their distinct differences in control models, they share the same control program code base and parameter specification with the same control parameters and value ranges [99]. Simply put, the RAV firmware is the exactly the same for hexacopter and octacopter in ArduPilot. We set the P (`PSC_VELXY_P`), I (`PSC_VELXY_I`), and D (`PSC_VELXY_D`) parameters of x, y-axis velocity controllers, and the P (`PSC_POSXY_P`) parameter of x, y-axis position controller respectively as 6.0, 0.02, 0.01, and 2.0 while keeping others default. Note that those values are valid according to the specification and they can be (remotely) set by either an attacker or an operator [18], [19] supported by major RAV firmware [22], [23], [110]. We then launch both the hexacopter and the octacopter to monitor the construction progress of a building. E.g., the RAV should move from Waypoint 1 to 2 (to the west direction) as described in Figure 4.1, and monitor the construction progress at Waypoint 2 standing still in the air. While the octacopter is able to finish the task accordingly, the hexacopter starts to show severe and persistent controller anomaly after a few seconds leading to a crash in the end.

The example above, in addition to typical unintentional incorrect implementation of the control algorithms, highlights the prevalence of cyber-physical bugs within RAV firmware implementations and the feasibility to exploit these bugs to attack RAVs. The appropriate fix might be as simple as adding an extra line in the source file to rule out those parameter values for the hexacopter if we have the source file. But the reality is that control program developers would reject this patch because it breaks the generality of the implementation or other RAV models[1], and we still need to take care of all other binary-only RAV firmware possibly stripped or running on the bare-metals. In this paper, we propose DisPatch, a systematic patching framework for RAV firmware focusing on cyber-physical bugs, by overcoming the challenges below:

Challenge 1: Identifying Different Variants of Controller Functions. We need to not only simply disassemble the binary code (as shown in Figure 4.2(1)) but also identify PID controller functions from the binary code (as illustrated in Figure 4.2(2)) as a first step before we could patch cyber-physical bugs within RAV firmware. Specifically, some controllers (1) do not have some of parameter variables if they are not needed (e.g., PID controller implemented only with one P parameter) or (2) contain additional parameter components. For (1), they are still mathematically valid PID controllers with some parameter value (e.g., I parameter) fixed at zero. For (2), some additional mechanisms or developer-customized parameters can be added. Therefore, DisPatch should have robust algorithms to identify PID controllers and their controller variables, decompiling the control algorithms.

Challenge 2: Finding Controller-Semantic Patch Locations. Given a high-level patch, we need to determine where to apply the patch within the firmware. This requires bridging the semantic gap between the abstract mathematical formulations and low-level binary instructions. Specifically, DisPatch must be able to identify the patch location and the semantics of each controller variable (e.g., z-axis velocity P parameter at address `0x7A22`). In fact, PID controllers are generic mathematical controller models that can be used to control any physical operations (e.g., any of 6DoF). In other words, even if we find a specific mathematical controller variable (e.g., P control parameter), we cannot figure out

[1]↑This is the link placeholder for developer's feedback anonymized for double-blinded submission

whether that is the z-axis velocity P parameter or roll angular P parameter. As a result, we cannot apply the patch customized for controller variables with specific semantics to others with different semantics but same names (as shown in Figure 4.2(2)). Only after identifying semantics of controller variables and functions (as described in Figure 4.2(3)), we can apply the controller-semantic patch to each controller components. -

Threat Model. We focus on cyber-physical bugs within RAV firmware that can be abused by attackers using the existing works [18], [19] or mathematical tool [89] with physical quantities of the target model. Considering the *model-specific* nature of RAVs, those threats can be caused by (1) parameter manipulation from attackers (either insiders or outsiders), and (2) physical specification change (e.g., RAV frame or weight changes). Insiders and outsiders can exploit remote configuration interfaces [18], [19], [78]–[80] to trigger cyber-physical bugs. Especially, insiders can manipulate control parameters whose impacts appear only on the certain environmental condition (e.g., certain flight paths with sharp turns) as shown in the previous works [18], [19]. Component replacement (e.g., RAV frame change caused by its vendor change) can also lead to control model deviation thus introducing additional cyber-physical bugs.

In this paper, we do not consider the orthogonal safety and security threats caused by either physical attacks (e.g., sensor spoofing) or traditional software bug exploitation (e.g., buffer overflows). There are existing works for RAVs to defend against those attacks [10], [12], [13], [29], [111]–[113].

4.3 Design

To patch cyber-physical bugs within RAV firmware, we propose DisPatch, an automatic patching framework supporting a number of RAV control models. Overall, DisPatch identifies not only PID controller functions but also their different variants (e.g., P, PID FF controllers). It further decompiles these controller functions to recover the controller variables and their control semantics (e.g., z-axis velocity P parameter or roll angular P parameter). End users can write a patch in a high-level DSL, which will be mapped into the corresponding controller functions variables and translated into binary-level instrumen-

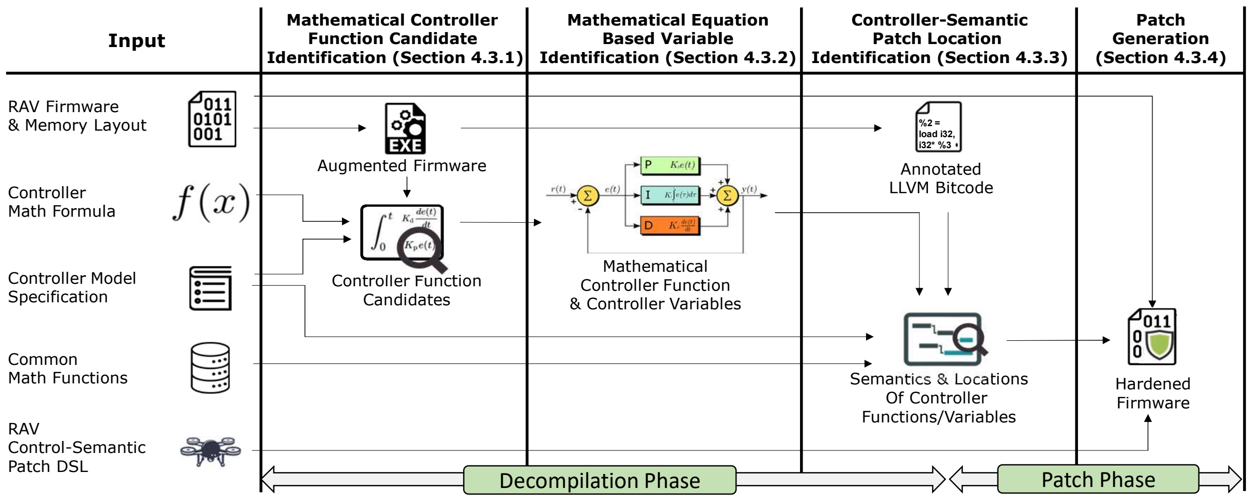

Input
Mathematical Controller Function Candidate Identification (Section 4.3.1)
Mathematical Equation Based Variable Identification (Section 4.3.2)
Controller-Semantic Patch Location Identification (Section 4.3.3)
Patch Generation (Section 4.3.4)
RAV Firmware & Memory Layout
Controller Math Formula
Controller Model Specification
Common Math Functions
RAV Control-Semantic Patch DSL
Augmented Firmware
Controller Function Candidates
Mathematical Controller Function & Controller Variables
Annotated LLVM Bitcode
Semantics & Locations Of Controller Functions/Variables
Hardened Firmware
Decompilation Phase
Patch Phase

Figure 4.3. The architecture of DisPatch.

tation to fix the corresponding cyber-physical bugs. Figure 4.3 shows the general workflow of DisPatch.

To perform those analyses and hardening steps, DisPatch takes the five following inputs: (1) an (stripped) RAV firmware with the memory layout of the platform board, (2) controller mathematical formulas (e.g., PID controller) used to control movements along the 6DoF, (3) controller model specification including the target controller structure (e.g., dependencies between controllers) and kinds of each controller (e.g., P or PID controller), (4) a signature list of common mathematical functions (e.g., `sin` and `cos`), and (5) a control-semantic patch written in the high-level DSL (e.g., enforcing values of controller variables to fall within the safe ranges). DisPatch then performs the following procedures to apply the patch to the input binary firmware.

1. **Mathematical Function Candidate Identification (Section 4.3.1)**: DisPatch starts with augmenting the binary firmware by annotating the memory layout, disassembles the binary firmware, and extracts a candidate list of the mathematical controller function using both static analysis and symbolic taint analysis.

2. **Mathematical Equation Based Controller Variable Identification (Section 4.3.2)**: To recognize customized PID controllers and their controller variables (a.k.a. **Challenge 1**), DisPatch captures the mathematical operations

of each candidate controller function using symbolic execution [114] and converts the outcome into an abstract syntax tree (AST). DisPatch then compares and matches the binary-derived ASTs with the domain knowledge-based AST templates to partially decompile these controller functions. The combination of symbolic execution techniques and mathematical expression templates ensures a robust semantic-based identification of controller functions and controller variables (e.g., P parameter of a PID controller).

3. **Controller-Semantic Patch Location Identification (Section 4.3.3)**: To further recover the semantics of controller functions (a.k.a. **Challenge 2**), we identify the *semantics* of controller variables (e.g., z-axis position or velocity P parameter) by matching dependencies at program, control, and mathematical model levels. Specifically, DisPatch performs data flow analysis to identify the program-level dependencies between primitive and cascading controllers guided by controller model domain knowledge, which contains a target controller structure described in the controller model specification. DisPatch then annotates every single controller variable by leveraging identified controller semantics and mathematical meanings of controller variables in the previous step (Section 4.3.2), achieving the semantic decompilation of these controller functions. DisPatch identifies instructions within controller functions accessing those annotated controller variables to denote both (1) controller semantics and (2) addresses of instructions accessing those variables as patch locations.

4. **Patch Generation (Section 4.3.4)**: With the knowledge of all the controller-semantic patch locations, DisPatch accepts a control-semantic patch written in a high-level DSL, finds the corresponding patch locations, translates and inject the patch at the binary level, and finally emits the hardended RAV firmware.

4.3.1 Mathematical Controller Function Candidate Identification

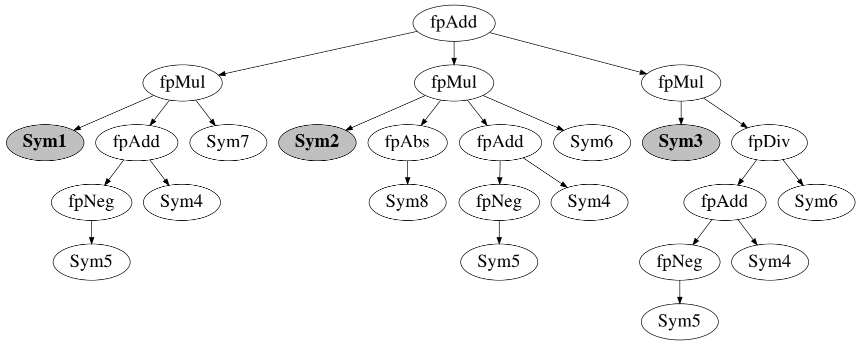
We identify the interested mathematical functions such as PID controllers within firmware using static analysis, starting with augmenting the binary firmware with the memory layout information of the target platform.

Firmware Augmentation. We first augment the binary firmware with its target board information such as memory layout (i.e., code and data memory addresses) in order to disassemble it correctly. The target board information such as memory layout is publicly available from the hardware board document such as system view description (SVD) file [115] written in the parsable Extensible Markup Language (XML) format or board-specific firmware development tools. Using this memory layout information, we augment binary firmware in the executable file format (e.g., Executable and Linkable Format (ELF)) using elftools [116]. The augmented firmware provides the base for the following analysis.

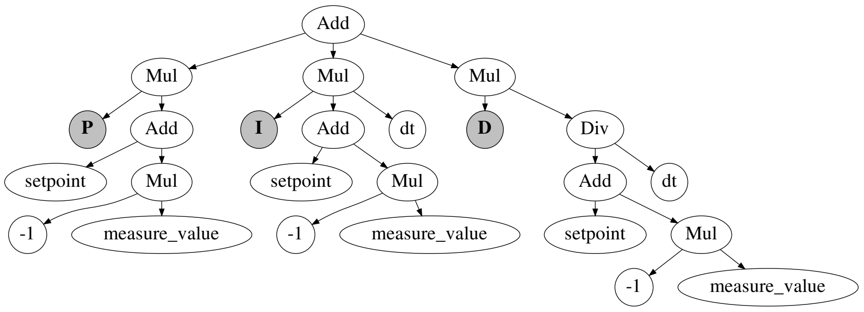
Shortlisting Controller Function Candidates. To filter out the functions that are not of our interests, we leverage the observation that controller functions using common control algorithms heavily use arithmetic instructions. For that, we customize MISMO [21]. Using the reference binary, we analyze each function in the assembly format and extract the features (e.g., number of floating point addition, subtraction, multiplication, and division instructions). We then use the extracted features as a baseline to define filtering rules, e.g., specifying the minimum/maximum value of each floating point instruction and the combination of floating point instructions in a controller function, and build a binary similarity checking tool to further prune the shortlisted functions.

4.3.2 Mathematical Controller Variable Identification

We compare the shortlisted functions after pruning against the mathematical function derived from the domain knowledge to identify individual controller functions (e.g., PID controller). We run symbolic execution [114] on the candidate functions and generate symbolic expression for the output of each candidate function. We convert the symbolic expression into an abstract syntax tree (AST), simplify the AST result, and compare with the AST



(a) Symbolic expression-derived AST for PID controller



(b) Domain knowledge-based AST template for PID controller

Figure 4.4. Controller semantic matching and variable identification.

templates of the mathematical functions from the domain knowledge to identify the exact controller.

Once the exact controller function is identified, we compare the vertices of the AST derived from the firmware with the domain knowledge-based AST templates to match the controller variables (e.g., PID gains, control error). As shown in Figure 4.4, we generated the ASTs for the identified PID function (Figure 4.4a) and the reference PID template (Figure 4.4b). We determine the semantic variable names for essential parameters (e.g., P, I, D gains) by comparing the subtree structures (e.g., number of child node) and individual node contents in terms of arithmetic opcodes. In that way, we can find all of the PID controllers and their variants.

We note that this process is built on top of MISMO [21] which can identify the PID controllers and its components. Furthermore, we also customized MISMO on top of angr [114] supporting symbolic taint analysis. This customization design enables DisPatch to find not only PID controllers but also their variants (e.g., PI and PID FF controllers). However, we skipped describing the detailed customized design of MISMO because this is not part of this dissertation.

4.3.3 Controller-Semantic Patch Location Identification

In this step, DisPatch decompiles controller functions and generates the patch location by identifying the semantics of controller functions and variables. DisPatch takes four inputs as shown in Figure 4.3: (1) the target binary firmware augmented with the memory layout (Section 4.3.1), (2) a controller model specification, (3) the list of common mathematical functions, (4) the list of the identified controllers and mathematical meaning of their controller variables (Section 4.3.2). We start with lifting the binary firmware to LLVM bitcode, find controller models and dependencies from the controller model specification, detect the *program-level* controller dependencies using data flow analysis, and annotate semantics of controller variables by comparing program-level dependencies with the controller model specification.

Annotated LLVM Bitcode Generation. DisPatch lifts the binary firmware augmented with the memory layout information (Section 4.3.1) into LLVM bitcode. Simultaneously, we annotate the LLVM-level instructions and functions with the addresses (in the binary firmware) of instructions, and tag them accordingly if they are common mathematical functions (e.g., *sin*, and *cos*) or controller functions and instructions accessing controller variables. We will show the detailed procedures in Section 4.4.

Controller Models and Dependencies. To identify the semantics of controllers, we need to know the exact controller models and controller dependencies. To identify the deployed controller models, DisPatch uses the domain knowledge described in the controller model specification, which contains the list of configurable controller parameters commonly available for operators [19], [99], [100], and tells the types of deployed variants of primitive controllers, and the number of primitive controllers inside of each cascading controller. For example, if there are a x-axis position P parameter, a x-axis velocity P parameter, and x-axis acceleration P, I, and D controller parameters mentioned in the list, it means that the x-axis cascading controller of the firmware has a P controller for the x-axis position,

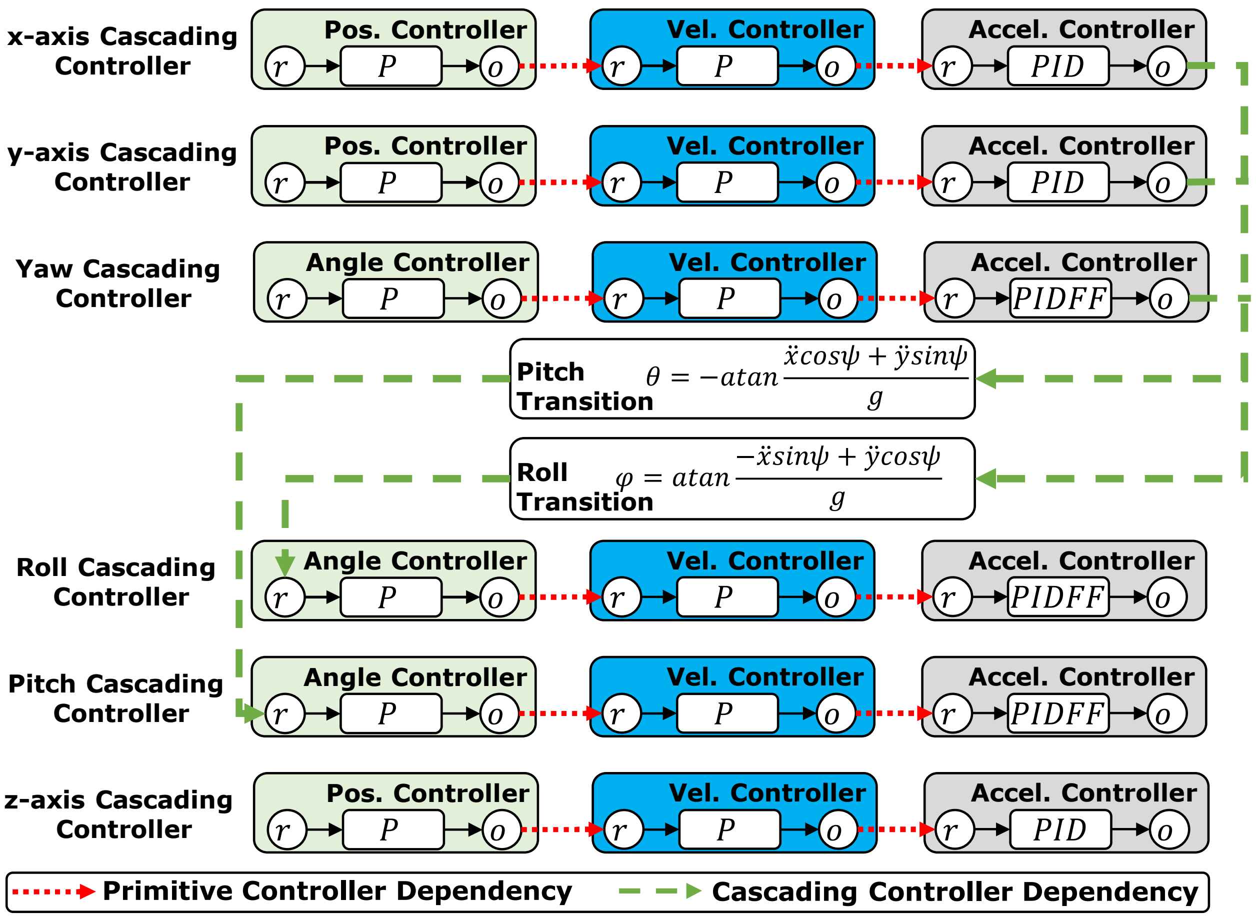

x-axis Cascading Controller
y-axis Cascading Controller
Yaw Cascading Controller
Roll Cascading Controller
Pitch Cascading Controller
z-axis Cascading Controller
Pos. Controller
Angle Controller
Vel. Controller
Accel. Controller
r
P
o
PID
PIDFF
Pitch Transition $\theta = -atan\frac{\ddot{x}cos\psi + \ddot{y}sin\psi}{g}$
Roll Transition $\varphi = atan\frac{-\ddot{x}sin\psi + \ddot{y}cos\psi}{g}$
Primitive Controller Dependency
Cascading Controller Dependency

Figure 4.5. Visualized controller dependencies inspired by CVDG Figure 2.7 and dependencies of 6DoF cascading controller structures Figure 2.2.

another P controller for the velocity, and a PID controller for the acceleration. In terms of the controller dependencies, we leverage the "control variable dependency graph (CVDG)" [18], which defines both primitive and cascading controller dependencies between controllers based on a generic RAV controller structure (Section 4.1).

Using the aforementioned domain knowledge, we can derive the controller dependency model as shown in Figure 4.5. Continuing on the x-axis example, its velocity primitive P controller has a dependency with its acceleration primitive PID controller (i.e., Vel. Controller → Accel. Controller in the x-axis cascading controller). Figure 4.5 also reflects controller variables (e.g., r for reference, P, I, D for controller parameters in Accel. Controller). We note that we did not include the *error* and *vehicle state* (although they exist) to focus on the

Algorithm 5 Semantics of Controller Variables Identification.

Input: Mathematical controller function data (C_{math}), The list of the common mathematical functions (MF), The list of all the functions (F), Controller model specification (CM)
Output: Semantics of controller variables (CVR)

```
1: function GetPrimitiveCtrlDeps($C_{math}, TC, F, CM$)
2:   Initialize $CD$;                                   ▷ $CD$: Primitive controller dependencies
3:   for $c \in C_{math}$ do                            ▷ Check dependencies of mathematical controllers
4:     $d \leftarrow$ GetDepFromCtrls($c, C_{math}, F, CM$)
5:     $CD[c] \leftarrow$ GetClosestDepCtrl($c, d, TC, F, CM$)
6:   return $CD$
7: function GetCascadingCtrls($CD, CM, TC$)
8:   Initialize $C_{cc}, chk$;                           ▷ $C_{cc}$: A set of cascading controllers
9:   for $pc \in CD.keys$ do                            ▷ Check every primitive controller ($pc$)
10:    if $pc \notin chk$ then                           ▷ Skip checked $pc$ having a dependency with another $pc$
11:      $cc \leftarrow$ RecursiveTrackDeps($pc, CD$)     ▷ Track data flows
12:      $C_{cc} \leftarrow C_{cc} \cup$ GetEachCascadingCtrl($cc, CM, TC$)
13:      $chk \leftarrow chk \cup$ GetPrimitiveCtrls($C_{cc}$)    ▷ Marked checked $pc$
14:  return $C_{cc}$
15: function AnnotateControllerVariableSemantics($C_{cc}, T, CM$)
16:  Initialize $CVR$;
17:  for $cc \in C_{cc}$ do                              ▷ Check each cascading controller ($cc$)
18:    $cr \leftarrow$ GetCascadingSemantic($cc, T, CM$)
19:    for $pc \in cc$ do                               ▷ Get semantics of primitive controllers in $cc$
20:      $r_{ctrl} \leftarrow$ GetPrimitiveCtrlSemantic($pc, cr, CM$)]
21:      for $cv \in pc$ do                             ▷ Iterate each controller variable ($cv$) in $pc$
22:        $CVR[cv] \leftarrow$ GetSemanticOfVar($cv, pc, CM$)
23:  return $CVR$
24: function IdentifySemantics($C_{math}, MF, F, CM$)    ▷ Main function
25:  $T \leftarrow$ GetTransFuncs($MF, F$)               ▷ $T$: Transition functions
26:  $CD \leftarrow$ GetPrimitiveCtrlDeps($C_{math}, T, F, CM$)   ▷ $Ctrl$: Controller
27:  $C_{cc} \leftarrow$ GetCascadingCtrls($CD, CM$)      ▷ $C_{cc}$: Cascading controllers
28:  $CVR \leftarrow$ AnnotateControllerVariableSemantics($C_{cc}, T, CM$)
29:  return $CVR$
```

controller model and dependency illustration. Finally, Figure 4.5 shows the dependencies between those cascading controllers via transition functions (e.g., Pitch Transition).

Controller Dependency Analysis. To identify the primitive and cascading controller dependencies in the LLVM bitcode, DisPatch performs inter-procedural data flow analysis as shown in Line 1-14, 25-26 in Algorithm 5 while comparing those data flows with the controller dependencies described in Figure 4.5.

To identify the primitive controller dependency (described in Line 1-6, 26), DisPatch first collects both input (i.e., error variable obtained by subtracting reference and vehicle state variables) and output controller variables of each identified controller (stored in C_{math}). This is represented in any of "controller" boxes in Figure 4.5. The result of primitive controller dependencies will be stored in CD (Line 26). DisPatch then chooses one primitive controller (Line 3), and performs data flow analysis in a backward manner from the chosen primitive controller. Specifically, DisPatch backtracks the variable update operations (e.g., multiplication) from an input variable of one controller to the output variables of any of the controllers (Line 3-5).

Meanwhile, DisPatch considers two features: (1) primitive controller dependency structure in a cascading controller and (2) alias update data flows. For (1), DisPatch checks (i) the number of primitive controllers (typically spanning from one to three in one of the six cascading controllers), and (ii) the dependency orders of those controllers. For example, as shown in the cascading controllers (e.g., for x-axis) in Figure 4.5, a P controller usually comes first (e.g., a position primitive controller) and a higher-order primitive controller (e.g., an acceleration primitive controller) typically comes later while having all of the P, I, and D terms. We do not consider 6DoF transition functions that are used to identify cascading controller dependency and will be described in more details (Line 5, 25).

To track alias update data flows, DisPatch keeps backtracking data flows from the aliases of the variables found in load or store instructions (e.g., to access a heap or global variable), To identify aliases, DisPatch performs inter-procedural context-sensitive points-to analysis [49] in advance. However, this procedure may miss some aliases in practice due to missing information from the compilation. To overcome this limitation, DisPatch collects instructions accessing either a global or heap variable from *this* pointer with a certain offset value, which is used as an alias identification key to heuristically identify alias memory accesses if the point-to analysis misses them. During this step, DisPatch prioritizes the data flows (only between the controller input and output) whose number of instructions is the smallest (Line 5). The intuition is that each controller's output is tightly coupled with the input of the dependent controller, hence, light computation involved. For instance, an output of a primitive controller is directly used as an input of its dependent controller in some RAVs (e.g., from a z-axis position to z-axis velocity primitive controller). If the data flows are complex, they may involve heavy computation that rarely happens in tightly coupled primitive controller dependencies.

We can distinctly differentiate the cascading controller dependency (Line 7-14, 27) from the primitive one because of the essential but unique computation to transit the degrees of freedom (Line 25). Specifically, as we described in Figure 4.5 and Section 4.1, the roll (ϕ) and pitch (θ) angles must be converted from the x, y-axis and yaw (ψ) angular acceleration controllers with different formulas. Whereas, z-axis cascading controller does not have transition. Thus, we leverage this unique computation pattern (Line 12, 25) to distinguish all the different cascading controllers (Line 9-13) by checking every identified primitive controller dependency result (Line 9, 26) from the previous step (Line 1-6). For example, the input of the roll and pitch angles are determined by the outputs of its x, y-axis and yaw controllers as defined in Equation 4.3 and 4.4.

$$\phi = atan((-\ddot{x}sin(\psi) + \ddot{y}cos(\psi))/g) \tag{4.3}$$

$$\theta = -atan((\ddot{x}cos(\psi) + \ddot{y}sin(\psi))/g) \tag{4.4}$$

In these equations, DisPatch can identify the transition computation by checking the usage of *sin*, *cos* and *atan* functions (from the list of common mathematical functions) and special constant such as g (the gravitational constant). DisPatch further identifies the x, y-axis, and yaw and roll and pitch controller variables by checking the difference in computation such as negation, *sin* and *cos* in both equations.

Semantic Annotation of Controller Functions and Variables. To achieve the semantic decompilation of controller functions and variables (Line 15-21, 28), DisPatch first identifies the possible semantics of the primitive controllers, through checking each identified cascading controller (cc of C_{cc} in Line 17). DisPatch then checks the program-level *primitive controller dependency* orders of controller function variants (e.g., P controller $\rightarrow$ P controller $\rightarrow$ PID FF controller) (Line 18), and checks whether there are the identical dependency combinations specified in the controller model specification (Line 19-20). For example, there are roll, pitch, yaw cascading controllers having the example ordered dependency above in Figure 4.5. Finally, DisPatch annotates each controller variable of each primitive controller (Line 21-22, 28-29) such as the z-axis P parameter and the z-axis reference variable.

```
1 z-axis.pos.ref   := [range,-50.0~50.0]
2 z-axis.vel.p     := [range,1.0~6.0]
3 ...
4 roll.vel.ff      := [equal,v?=0.0->v=1.0]
```

Listing 4.1 An example of an RAV control-semantic patch DSL following the EBNF-based grammar. This DSL indicates DisPatch will instrument z-axis position reference (Line 1), velocity P parameter (Line 2), and roll velocity (i.e., roll rate) FF parameter (Line 4) with the given enforcement rule.

4.3.4 Patch Generation

RAV Control-Semantic Patch DSL. Instead of mandating end users to deal with binary-level instrumentation directly, we design a high-level domain specific lanaguge (DSL) for users writing control-semantic patches. We show the DSL grammar in Figure 4.6.

Our DSL supports seven arithmetic operations to protect or regulate controller variables in all of the controllers to control 6DoF. For example in Listing 4.1, end users can write a control-semantic patch following the "*control variable := operation*" pattern. Each control variable is specified by (1) one of the 6DoFs (e.g., z-axis or roll), (2) primitive controller type (e.g., position (pos), velocity (vel)), and (3) mathematical controller variable (e.g., P, I, D parameter or reference). An operation could be any of the seven operations defined by our DSL. In this example, DisPatch will add range check on the P parameter gain of a z-axis velocity controller (denoted as `z-axis.vel.p`) with the value range from 1.0 to 5.6 (denoted as [`range, 1.0 5.6`]) in Line 2. We also prevent a controller variable from being a certain value such as [`equal, v?=0.0`→`v==1.0`] (Line 4). If the value (`roll.vel.ff`) is set to the target value (e.g., 0.0) in the firmware, DisPatch will set that value into the desired value (e.g., 1.0) according to the patch.

With the list of annotated semantics of controller variables retrieved from the previous step (Section 4.3.3) and a control-semantic patch written in a high-level DSL, DisPatch is able to apply the patch to the target binary firmware.

Binary Patching. Finally, DisPatch translates the control-semantic patch into the corresponding binary-level patch. From the patch, DisPatch has the list of the controller variables and controller codes accessing those variables (more details in Section 4.4). With

⟨*cascading*⟩ ::= ‘`x-axis`’ | ‘`y-axis`’ | ‘`z-axis`’ | ‘`xy-axis`’
    | ‘`roll`’ | ‘`pitch`’ | ‘`yaw`’;

⟨*primitive*⟩ ::= ‘`pos`’ | ‘`vel`’ | ‘`acc`’;

⟨*variable*⟩ ::= ‘`p`’ | ‘`i`’ | ‘`d`’ | ‘`ff`’ | ‘`r`’ | ‘`e`’ | ‘`x`’;

⟨*digit*⟩ ::= ‘`0`’ | ‘`1`’ | ‘`2`’ | ‘`3`’ | ‘`4`’ | ‘`5`’ | ‘`6`’ | ‘`7`’ | ‘`8`’ | ‘`9`’;

⟨*float*⟩ ::= ⟨*digit*⟩, ‘`.`’, {⟨*digit*⟩};

⟨*less*⟩ ::= ‘`less`’, ‘`v<`’;

⟨*greater*⟩ ::= ‘`greater`’, ‘`v>`’;

⟨*less-equal*⟩ ::= ‘`less-equal`’, ‘`v<=`’;

⟨*greater-equal*⟩ ::= ‘`greater-equal`’, ‘`v>=`’;

⟨*comp*⟩ ::= ⟨*less*⟩ | ⟨*greater*⟩ | ⟨*less-equal*⟩ | ⟨*greater-equal*⟩, ⟨*float*⟩;

⟨*comp-default*⟩ ::= ‘`equal`’ | ‘`not-equal`’, ‘`,`’, ‘`v?=`’, ⟨*float*⟩, ‘`->`’, ‘`v=`’, ⟨*float*⟩;

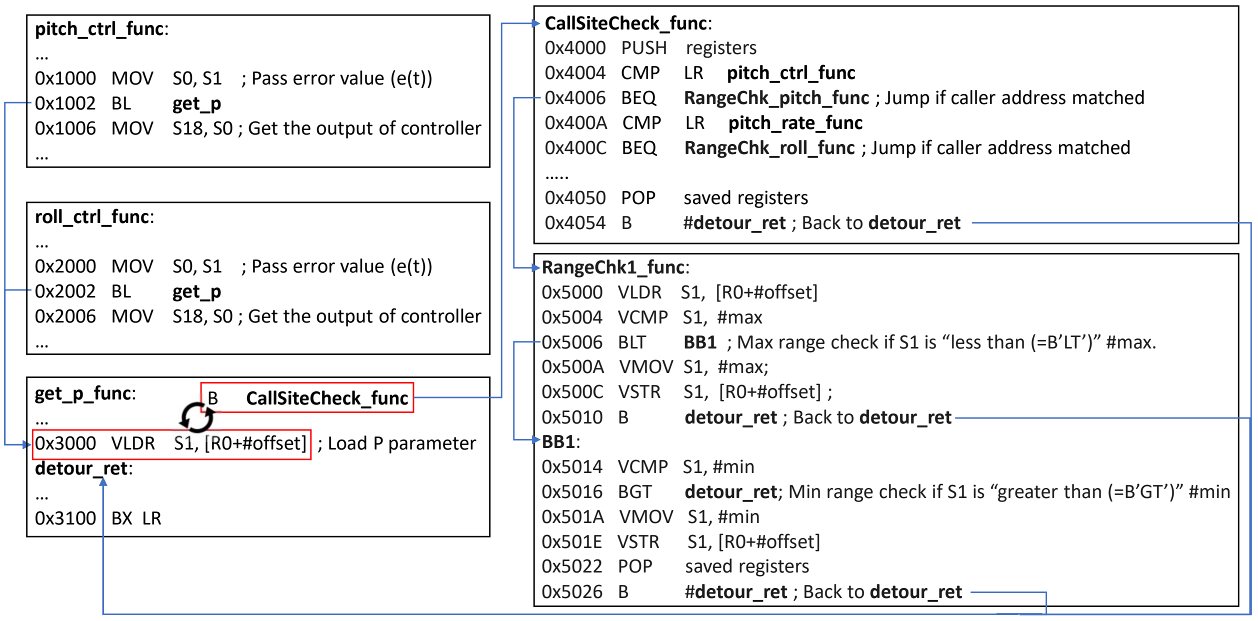
⟨*range*⟩ ::= ‘`range`’, ‘`,`’ ‘`v=`’, ⟨*float*⟩, ‘`~`’, ⟨*float*⟩;

⟨*policy*⟩ ::= ‘`[`’, ⟨*range*⟩ | ⟨*comp*⟩ | ⟨*comp-default*⟩, ‘`]`’;

⟨*patch*⟩ ::= { ⟨*cascading*⟩, ’.’, ⟨*primitive*⟩, ’.’, ⟨*variable*⟩,
    ’:=’, ⟨*policy*⟩ };

Figure 4.6. EBNF of DisPatch patch expression grammar.

these information, DisPatch patches the binary firmware using the detour approach [117] resilient to even partially incorrect disassembly results [117], [118]. One challenge is that controller code can access controller variables with different semantics (e.g., P parameter of either z-axis velocity or roll angular controller), and we will not know the semantics of these variables passed by until we check the callsite of each controller function. Such callsites can be differentiated from the different controller semantics (e.g., one controller function can be called by either z-axis velocity or roll angular codes). If there is only one callsite, DisPatch directly deploys the patch to the target controller variable. Otherwise, DisPatch performs *callsite-aware* instrumentation to handle the multiple semantics of the same controller variable.



```
pitch_ctrl_func:
...
0x1000  MOV   S0, S1   ; Pass error value (e(t))
0x1002  BL    get_p
0x1006  MOV   S18, S0 ; Get the output of controller
...
```

```
roll_ctrl_func:
...
0x2000  MOV   S0, S1   ; Pass error value (e(t))
0x2002  BL    get_p
0x2006  MOV   S18, S0 ; Get the output of controller
...
```

```
get_p_func:           B    CallSiteCheck_func
...
0x3000  VLDR   S1, [R0+#offset] ; Load P parameter
detour_ret:
...
0x3100  BX  LR
```

```
CallSiteCheck_func:
0x4000  PUSH   registers
0x4004  CMP    LR    pitch_ctrl_func
0x4006  BEQ    RangeChk_pitch_func ; Jump if caller address matched
0x400A  CMP    LR    pitch_rate_func
0x400C  BEQ    RangeChk_roll_func ; Jump if caller address matched
.....
0x4050  POP    saved registers
0x4054  B      #detour_ret ; Back to detour_ret
```

```
RangeChk1_func:
0x5000  VLDR   S1,  [R0+#offset]
0x5004  VCMP   S1,  #max
0x5006  BLT    BB1  ; Max range check if S1 is "less than (=B'LT')" #max.
0x500A  VMOV   S1,  #max;
0x500C  VSTR   S1,  [R0+#offset] ;
0x5010  B      detour_ret ; Back to detour_ret
BB1:
0x5014  VCMP   S1, #min
0x5016  BGT    detour_ret; Min range check if S1 is "greater than (=B'GT')" #min
0x501A  VMOV   S1, #min
0x501E  VSTR   S1, [R0+#offset]
0x5022  POP    saved registers
0x5026  B      #detour_ret ; Back to detour_ret
```

Figure 4.7. An example of an instrumented callsite-aware code payload to check the value range of P parameters of roll and pitch angular controllers.

For *callsite-aware* instrumentation, DisPatch checks the callsite address via either a return register or return address pointer stored in the stack, and places a trampoline to distinguish different semantics of the target control variable when it is loaded. For instance in Figure 4.7, we assume that both `roll_ctrl_func` and `pitch_ctrl_func` will call `get_p_func` to compute the PID controller output. P is loaded at the same code location (`VLDR` at `0x3000`) with different semantics depending on the caller. To apply the patch to the P parameter, we replace the `VLDR` with the jump instruction to `CallSiteCheck_func` using the detour-based approach, which checks for the callsite address. In our example, `LR` stores the callsite address and it is checked at `0x4004` and `0x400A` to determine the corresponding patch to apply. As an example of taking the `RangeChk1_func` branch, once the patch (from `0x5000` to `0x5026`) executed, we will move back to the next address of `0x3000` (i.e., the replaced instruction address) in `get_p_func` to continue the normal execution.

4.4 Implementation

We modified RetDec [119] to tag binary instruction addresses in the firmware to their corresponding LLVM instructions in the LLVM debugging symbol format, as well as common mathematical functions and identified controller functions and instructions accessing controller variables identified in Section 4.3.2 into the corresponding LLVM objects. Common mathematical functions (e.g., *sin* and *cos*) are well-defined transformation functions with fixed inputs and outputs unlike PID controllers with variants. Hence, we use the signature-based function matching technique [120] to identify those mathematical functions from the binary firmware without missing any of them.

RetDec does not fully support argument and return variable identification, which can cause DisPatch to miss inter-procedural data flows of controller dependencies. To complement such missing information, DisPatch tracks read and write operations on variables (e.g., registers or stack variable in the binary firmware) as typical binary analyses do [114], [121], [122]. If either an argument of a caller or a return variable of a callee is read before it is written, we consider such variables as passed from either a caller or callee. To reduce false positives, we choose those argument/return registers and stack variables based on the calling convention in the ARM architecture (e.g., the arguments and return values are held in lower registers `R0-R3`).

We implement the mathematical controller function candidate identification (Section 4.3.1) as a plugin for IDA Pro 7.2 [121], and a symbolic taint analyzer with angr [114] using IDA Pro as a disassembler and the function call tracing with gdb scripts to further shortlist candidate functions. We also implement a firmware augmentation script to add memory layout information on a binary blob using elftools [116] We also use the symbolic execution engine in angr to generate the symbolic expression for controller functions (Section 4.3.2) and develop the semantic matching module to perform AST generation, simplification, mathematical operator examination and merging. We use IDA Pro as a disassembler for this step. We implement the controller-semantic patch location identification module (Section 4.3.3) on top of the SVF 1.8 static analysis tool [49] for points-to analysis with LLVM 9.0 [123]. Finally, we implemented our binary patching (Section 4.3.4) based on detour-based binary rewriting technique as a plugin for IDA Pro 7.2. The total LoC of our whole system is 9,832 lines of code (5,327 lines of C++ and 4,505 lines of Python). DisPatch consists of four modules, and their lines of code is described in Table 4.1. We also provide which tools are used to developer our framework in the second column.

Table 4.1. Lines of codes for DisPatch's analysis modules.

Module	Used Tools	Lines of Codes
Mathematical Function Candidate Identification (Section 4.3.1)	IDA Pro [121], elftools [116], and angr [114]	1,823
Mathematical Equation Based Controller Variable Identification (Section 4.3.2)	IDA Pro [121], and angr [114]	1,425
Controller-Semantic Patch Location Identification (Section 4.3.3)	LLVM [123], SVF [49], FLIRT [120], and RetDec [119]	5,327
Patch Generation (Section 4.3.4)	IDA Pro [121]	1,245
Total	-	9,832

4.5 Evaluation

We validate DisPatch on real-world RAV firmware to answer the following key questions:

- **Q1**: How accurate is the controller function and variable identification/decompilation? (Section 4.5.1)
- **Q2**: How practical is DisPatch for patching real-world cyber-physical bugs? (Section 4.5.2)
- **Q3**: How much overhead is imposed by the patch of DisPatch on RAV operations? (Section 4.5.3)

Experimental Setup. We use ArduPilot 3.6 [22] based RAV firmware during the evaluation because it is the most popular control software [32], [124], [125], adapted and customized by many RAV venders [126] including BirdsEyeView, Walkera, and Traxxas [127]–[129], and supports a variety of RAV control models such as copter, plane, helicopter and submarine even with different physical quantities (e.g., RAV body size, weight and motor specification). ArduPilot 3.6 adopts ChibiOS [130] as the default real-time operating system. Accordingly, vendors often customize ArduPilot and provide their own firmware (binary) download or update services [131]–[133] for their products. We evaluate DisPatch using ArduPilot on Pixhawk [134] as our target hardware board. This board is equipped with a 192KB SRAM, a 2MB flash memory, and multiple sensors (a GPS, a magnetometer, an accelerometer, a barometer, and a gyroscope). We use the black magic probe [135] to enable execution tracing, retrieving, and firmware uploading/downloading to/from the Pixhawk board. Note that all evaluations have been done on four different RAV control models: copter, helicopter, plane, and submarine.

4.5.1 Controller Identification/Decompilation Accuracy

We had three steps to identify the mathematical controller functions: (1) mathematical function candidate identification with static analysis (Section 4.3.1), (2) mathematical function candidate identification with static analysis and symbolic taint analysis (Section 4.3.1), and (3) controller function recognition by checking AST similarities between a controller function and a mathematical template (Section 4.3.2). We show the results for each step in Table 4.2. In fact, our target firmware does not contain a complete PID controller function. Instead, we found P controller functions, sub-controller functions of PID, and PID FF controller functions as described in Table 4.2.[2] Here, we show on the copter, helicopter, plane, and submarine models.

In Step (1) (denoted as static analysis in Table 4.2), DisPatch's static analysis cannot distinguish among P, D, FF controllers due to their similar mathematical instruction signatures. Hence, DisPatch finds only 34-47 I and 27-34 non-I (i.e., P, D, FF) (sub-)controller functions in this step. The total number of (sub-)controller function candidates are 61-81 functions. In Step (2) (denoted as SS analysis in Table 4.2), DisPatch intersects the static analysis results with symbolic taint analysis results. Similar to Step (1), DisPatch still cannot distinguish between P, I, and FF (sub-)controller functions. Thus, there are 5-8 I and 13-16 non-I (sub-)controller functions. However, the number of candidates have significantly reduced from 65.6% to 75.6% for three control models compared to Step (1). In

[2]↑PID FF controller represents the PID controller with the feedforward term as well (FF in equation 4.1 in Section 4.1).

Table 4.2. Accuracy of the mathematical PID controller function identification of the copter, helicopter, plane, and submarine models in each of three steps. Ctrl.: Controller functions, SS: Static and Symbolic, FP: False positive rate.

Model	Processing Step	# of P Ctrl.	# of I Ctrl.	# of D Ctrl.	# of FF Ctrl.	# of Total Ctrl.	FP out of Total Ctrl.
Copter	Static Analysis	30	47	30	30	77	87.0%
	SS Analysis	14	5	14	14	19	47.4%
	AST Semantic Matching	5	2	2	1	10	**0.0%**
	Ground Truth	5	2	2	1	10	-
Helicopter	Static Analysis	31	47	31	31	78	87.2%
	SS Analysis	13	6	13	13	19	47.4%
	AST Semantic Matching	5	2	2	1	10	0.0%
	Ground Truth	5	2	2	1	10	-
Plane	Static Analysis	34	47	34	34	81	87.7%
	SS Analysis	13	8	13	13	21	52.4%
	AST Semantic Matching	5	2	2	1	10	0.0%
	Ground Truth	5	2	2	1	10	-
Submarine	Static Analysis	27	34	27	27	61	83.6%
	SS Analysis	16	5	16	16	21	52.4%
	AST Semantic Matching	5	2	2	1	10	0.0%
	Ground Truth	5	2	2	1	10	-

Step (3) (denoted as AST semantic matching in Table 4.2), DisPatch now can distinguish all of the different (sub-)controller candidate functions. In this step, DisPatch can find P(5), I(2), D(2), and FF(1) (sub-)controller function candidates for all of the control models These identified candidates are identical to the ground truth (number of controller functions identified manually). As demonstrated, DisPatch's incremental filtering gets more precise in each of the following step, leading to the zero false alarm in the end.

Using the result of mathematical PID (sub-)controller identification, DisPatch identifies controller-semantic patch locations (Section 4.3.3). Table 4.3 summarizes the results for (1) which variant of PID controllers are used for each controller semantic (e.g., z-axis velocity), and (2) the number of controller variables (including P, I, D, and FF control parameters, the reference, error, and vehicle state) of different semantics of controllers (Section 4.1). Specifically, in the case of the PID controllers, one PID controller is used to control z-axis acceleration with P, I and D control parameters while having reference, error and vehicle states. Hence, it has six controller variables. The other PID controller is customized to control two dimensional (x, y-axis) accelerations having respectively one more reference, error and vehicle states to control both x and y-axis simultaneously. Hence, it has nine con-

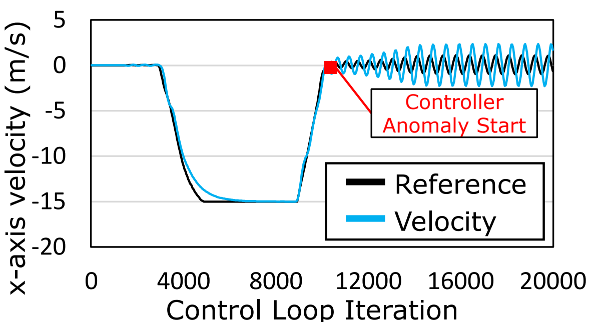
Table 4.3. The semantics of mathematical PID controller functions and their controller variables for all of the four control models. Their semantic identification rates of both functions and variables are 100% accurate Ctrl.: Controller function, ✓: semantic of a controller used, CV: controller variable.

Controller Semantic	P Ctrl.	PID Ctrl.	PID FF Ctrl.	# of CVs
x, y-axis position	✓			7
x, y-axis velocity		✓		9
z-axis position	✓			4
z-axis velocity	✓			4
z-axis acceleration		✓		6
Roll angle	✓			4
Roll angular rate			✓	7
Pitch angle	✓			4
Pitch angular rate			✓	7
Yaw angle	✓			4
Yaw angular rate			✓	7
Total # of CVs	-	-	-	63

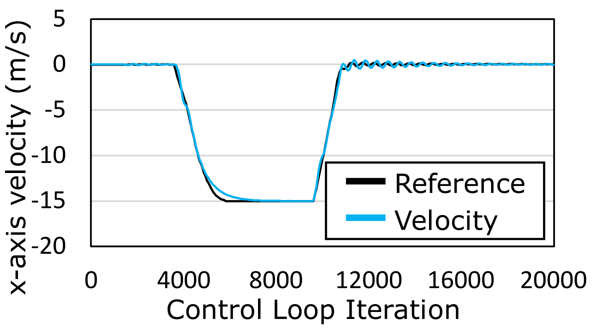
troller variables. As result, DisPatch finds total six P controllers, two PID controllers, and three PID FF controllers. DisPatch identifies all of those controllers and 63 common PID controller variables to be patched without missing any deployed controllers and controller variables to control 6DoF.

4.5.2 Case Studies

We show how DisPatch could be used to patch real-world cyber-physical bugs within RAV firmware with two case studies. In the consideration of safety regulations, we run these long distance testing flights using a software-in-the-loop (SITL) simulator [55], which has high accuracy and fidelity to emulate the physical world under the physics laws, hence is commonly used in RAV testing by researchers [18], [19], [22], [23] and RAV development companies [136]–[138]. Note that we tested our case studies on top of the real RAV firmware, but simulated aerial vehicles and physical environment *only to inspect the physical impacts with and without the patches.*



(a) Attack implication on the hexacopter model.



(b) Attack implication on the octacopter model.

Figure 4.8. Case 1: The implications of manipulating P, I, and D of the x, y-axis velocity controller.

Case Study I: Patching a cyber-physical bug unique to a certain RAV control model

Following up on the motivating example introduced in Section 4.2, we show the controller states of both the hexacopter and octacopter in Figure 4.8. In this example, the RAV is supposed to execute the moving command (i.e., moving in the west direction at a 15m/s) and hover at the destination. As shown in Figure 4.8b, the octacopter flies stably, while the hexacopter shows severe and persistent controller anomaly with the same control parameter setting, as shown in Figure 4.8a. Specifically, from 10,655 Iterations (i.e., destination arrival and hovering in the sky), both reference ($\dot{r}_x(t)$) and vehicle state ($\dot{x}_x(t)$) of the x, y-axis velocity controller cannot track with each other. We found that the RAV firmware for the hexacopter and octacopter are identical while having the same loose check for both the controller function and parameter specification for the following parameters: the P (`PSC_VELXY_P`), I (`PSC_VELXY_I`), and D (`PSC_VELXY_D`) of the x, y-axis velocity controller, and the P (`PSC_POSXY_P`) of the x, y-axis position controller. Therefore, we decided to apply the patch specialized to the hexacopter model for the above four parameters to fix this vulnerability.

To patch such a cyber-physical bug, we use (1) testing systems [19] or (2) mathematical tool [89] to determine the proper parameter range tightly coupled with the RAV control model. We write our control-semantic patch in the DSL shown in Listing 4.2. For example,

```
xy-axis.pos.p   := [range, 1.0~1.5]
xy-axis.vel.p   := [range, 1.5~4.5]
xy-axis.vel.i   := [range, 0.2~1.0]
xy-axis.vel.d   := [range, 0.2~1.0]
```

Listing 4.2 An RAV control-semantic patch DSL for the Case Study I. Four parameters are patched.

we decide to reduce the default value range of `PSC_VELXY_P` from 0.1 to 6.0[3] to from 1.5 to 4.5 described as "`xy-axis.vel.p` :=[`range, 1.5 4.5`]".

Given the patch, DisPatch first identifies the mathematical controller candidates by intersecting the results of static analysis and symbolic execution (Section 4.3.1); it finalizes the mathematical controllers and their mathematical expressions for each controller variable (Section 4.3.2); DisPatch then identifies all of the semantics of the mathematical controllers and their control parameters as we show in Table 4.3 of Section 4.5.1.

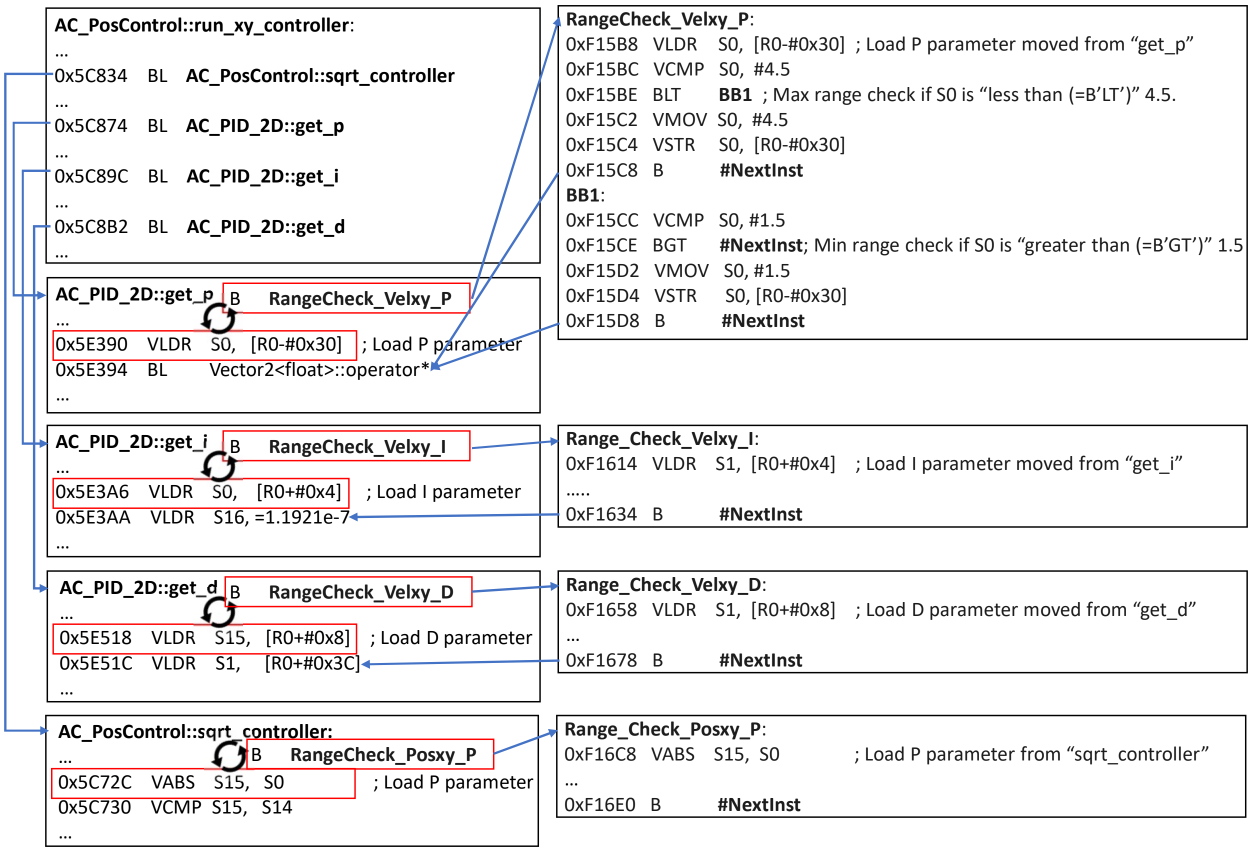
Finally, DisPatch deploys the patch at the binary level as shown in Figure 4.9, which enforces range checks on the four control parameters in the different (sub-)controller functions by making the firmware call the four protection functions (`RangeCheck_Velxy_P`, `RangeCheck_Velxy_I`, `RangeCheck_Velxy_D`, and `RangeCheck_Posxy_P`). For example, DisPatch reduces the value range of `PSC_VELXY_P` from 1.5 to 4.5 by making the firmware jump to the `RangeCheck_Velxy_P` detour function. We later ran the patched firmware and confirmed the stability of the hexacopter's operation in run-time.

Case Study II: Limiting flight distance to prevent communication loss

Operators utilize a remote control system to control the RAV, and this communication range is limited as illustrated in Figure 4.10. WiFi is usually limited to approximately 50 meters and radio controllers support ranges of few kilometers.[4] However, if a WiFi or radio module has been replaced or an RAV encounters adversarial environments causing radio communication range fluctuation and/or jamming attacks, it can result in unsafe states

[3]↑Recommended in the official document [99].

[4]↑The communication range of telemetry module such as FrSky DJT 2.4Ghz [139] spans from 1.5 to 2.5 kilometers.



```
AC_PosControl::run_xy_controller:
...
0x5C834   BL   AC_PosControl::sqrt_controller
...
0x5C874   BL   AC_PID_2D::get_p
...
0x5C89C   BL   AC_PID_2D::get_i
...
0x5C8B2   BL   AC_PID_2D::get_d
...

AC_PID_2D::get_p   B   RangeCheck_Velxy_P
...
0x5E390   VLDR   S0,   [R0-#0x30]   ; Load P parameter
0x5E394   BL     Vector2<float>::operator*
...

AC_PID_2D::get_i   B   RangeCheck_Velxy_I
...
0x5E3A6   VLDR   S0,   [R0+#0x4]   ; Load I parameter
0x5E3AA   VLDR   S16, =1.1921e-7
...

AC_PID_2D::get_d   B   RangeCheck_Velxy_D
...
0x5E518   VLDR   S15,   [R0+#0x8]   ; Load D parameter
0x5E51C   VLDR   S1,    [R0+#0x3C]
...

AC_PosControl::sqrt_controller:
...                B   RangeCheck_Posxy_P
0x5C72C   VABS   S15,  S0              ; Load P parameter
0x5C730   VCMP  S15,  S14
...

RangeCheck_Velxy_P:
0xF15B8  VLDR   S0,  [R0-#0x30]  ; Load P parameter moved from “get_p”
0xF15BC  VCMP   S0,  #4.5
0xF15BE  BLT    BB1  ; Max range check if S0 is “less than (=B’LT’)” 4.5.
0xF15C2  VMOV  S0,  #4.5
0xF15C4  VSTR   S0,  [R0-#0x30]
0xF15C8  B       #NextInst
BB1:
0xF15CC  VCMP  S0, #1.5
0xF15CE  BGT    #NextInst; Min range check if S0 is “greater than (=B’GT’)” 1.5
0xF15D2  VMOV   S0, #1.5
0xF15D4  VSTR    S0, [R0-#0x30]
0xF15D8  B       #NextInst

Range_Check_Velxy_I:
0xF1614  VLDR   S1,  [R0+#0x4]   ; Load I parameter moved from “get_i”
.....
0xF1634  B        #NextInst

Range_Check_Velxy_D:
0xF1658  VLDR   S1,  [R0+#0x8]   ; Load D parameter moved from “get_d”
...
0xF1678  B        #NextInst

Range_Check_Posxy_P:
0xF16C8  VABS   S15,  S0          ; Load P parameter from “sqrt_controller”
...
0xF16E0  B        #NextInst
```

Figure 4.9. An example of the protection logics for PSC_VELXY_P, PSC_VELXY_I, PSC_VELXY_D, and PSC_POSXY_P parameters.

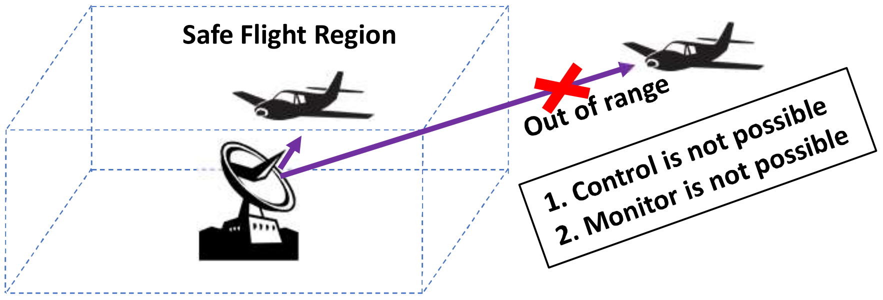
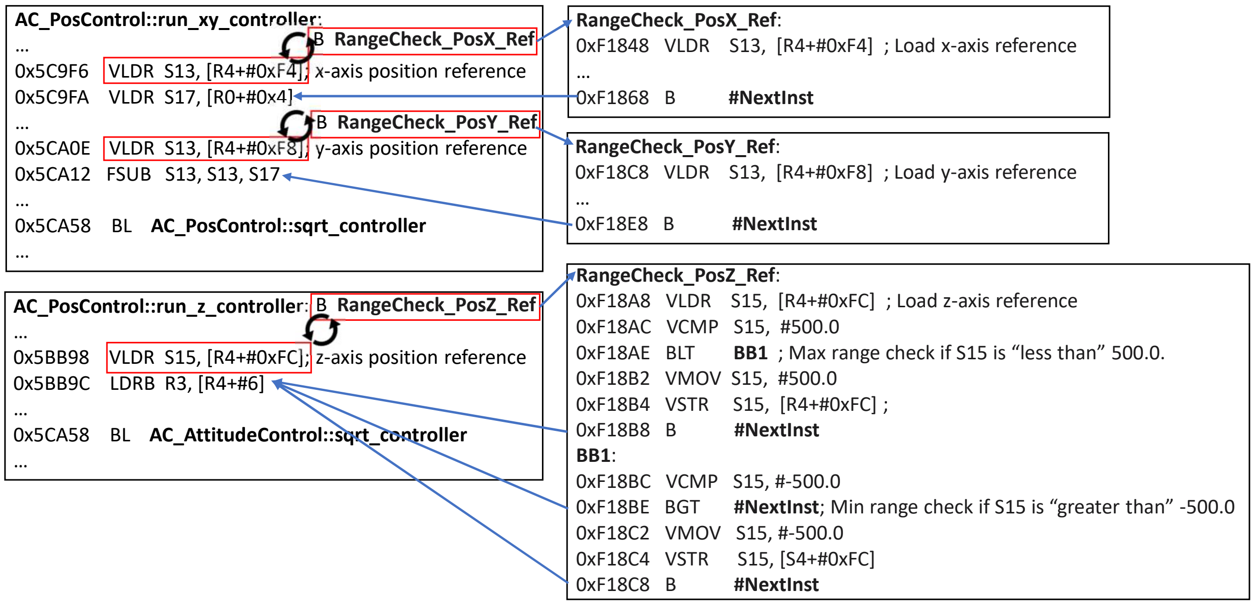


Figure 4.10. Case 2: Safe flight region enforcement by limiting the reference values of x, y-axis position controllers.

leading to task abortion or even crash. To launch this attack, attackers can modify the flight range by manipulating the allowed flight distance parameter (supported by advanced RAV software such as ArduPilot) or changing the missions via the remote interface [18], [78]–[80]. Therefore, operators would like to enforce static flight distance limitations to avoid

```
x−axis.pos.ref   := [range,  −500.0~500.0]
y−axis.pos.ref   := [range,  −500.0~500.0]
z−axis.pos.ref   := [range,  −500.0~500.0]
```

Listing 4.3 An RAV control-semantic patch DSL for the Case Study II. Three reference variables are patched.



```
AC_PosControl::run_xy_controller:
...                                   B RangeCheck_PosX_Ref
0x5C9F6   VLDR  S13, [R4+#0xF4]; x-axis position reference
0x5C9FA   VLDR  S17, [R0+#0x4]
...                                   B RangeCheck_PosY_Ref
0x5CA0E   VLDR  S13, [R4+#0xF8]; y-axis position reference
0x5CA12   FSUB  S13, S13, S17
...
0x5CA58   BL   AC_PosControl::sqrt_controller
...
```

```
AC_PosControl::run_z_controller: B RangeCheck_PosZ_Ref
...
0x5BB98   VLDR  S15, [R4+#0xFC]; z-axis position reference
0x5BB9C   LDRB  R3, [R4+#6]
...
0x5CA58   BL   AC_AttitudeControl::sqrt_controller
...
```

```
RangeCheck_PosX_Ref:
0xF1848   VLDR   S13,  [R4+#0xF4]  ; Load x-axis reference
...
0xF1868   B        #NextInst
```

```
RangeCheck_PosY_Ref:
0xF18C8   VLDR   S13,  [R4+#0xF8]  ; Load y-axis reference
...
0xF18E8   B        #NextInst
```

```
RangeCheck_PosZ_Ref:
0xF18A8   VLDR   S15,  [R4+#0xFC]  ; Load z-axis reference
0xF18AC   VCMP   S15,  #500.0
0xF18AE   BLT     BB1  ; Max range check if S15 is "less than" 500.0.
0xF18B2   VMOV  S15,  #500.0
0xF18B4   VSTR    S15,  [R4+#0xFC] ;
0xF18B8   B        #NextInst
BB1:
0xF18BC   VCMP   S15, #-500.0
0xF18BE   BGT     #NextInst; Min range check if S15 is "greater than" -500.0
0xF18C2   VMOV   S15, #-500.0
0xF18C4   VSTR     S15, [S4+#0xFC]
0xF18C8   B        #NextInst
```

Figure 4.11. An example of the restriction logics for references of x, y, z-axis position controllers.

such misbehaviors. With DisPatch, we could write a control-semantic patch as shown in Listing 4.3 to limit the flight distance within 500 meters.

Throughout the controller variable identification steps, DisPatch needs to identify the position reference variables in three axe (r_x, r_y, and r_z). Then, we can limit those values with the defined ranges (i.e., from -500.0 to 500.0) as described in the patch. Figure 4.11 shows how DisPatch deploys the flight distance patch in x, y, z-axe by replacing their reference values with `B` instructions jumping respectively to `RangeCheck_PosX_Ref`, `RangeCheck_PosY_Ref`, and `RangeCheck_PosZ_Ref`, all of which enforce the 500-meter limitation. Similarly, we were able to verify the 500-meter limitation enforced by the patch using our testing platform.

4.5.3 Performance Overhead

To ensure that the patched firmware can still fit within the flash ROM, and does not disturb the RAV's real-time operations at run-time, we measure both the space and run-time overhead caused by patching in worst case scenario, where we instrumented all of 63 controller variables we found in the firmware, and applied the min/max range checking patch to each control variable using DisPatch since this kind of patch imposes the largest overhead compared with other patches.

Space Overhead. We measured DisPatch's space overhead on four different RAV control models: copter, helicopter, plane, and submarine. The original firmware sizes of these models are 949KB, 936KB, 936KB, and 822KB, respectively. Each firmware image includes RTOS, control program codes, and read-only data (e.g., communication message strings) that must be loaded into a non-volatile flash memory during the RAV's boot up time. We show the measured space overheads following the worse case scenario. The patch increased the size of the firmware for different models to 954KB, 941KB, 941KB, and 827KB respectively, with overheads of 0.55%, 0.56%, 0.56%, and 0.64% (i.e., on average 0.58%). This space overhead is negligible and acceptable for real-world deployments, and our patched RAV firmware can easily fit within the RAV's size-constrained flash memory. We summarize the space overheads in Table 4.4.

Table 4.4. Space overhead introduced by DisPatch. We instrumented all of the identified controllers in our four different RAV control models.

RAV Control Model	Original Firmware Size	Instrumented Firmware Size	Space Overhead
Copter	949KB	954KB	0.55%
Helicopter	936KB	941KB	0.56%
Plane	936KB	941KB	0.56%
Submarine	822KB	827KB	0.64%
Average	911KB	916KB	0.58%

Run-time Overhead. We measured the run-time overhead on the four RAV models. We first measured the run-time overhead of 48 real-time tasks running within the firmware of the copter model. The execution frequency of each real-time task ranges from 0.1Hz to 400Hz.

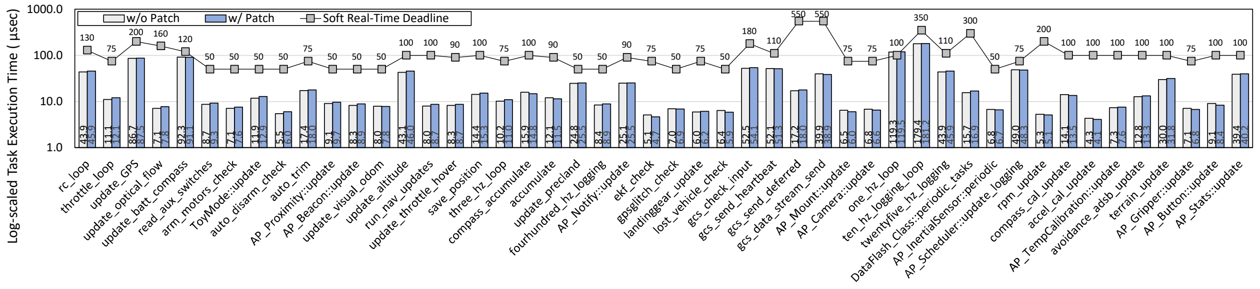


Figure 4.12. Run-time overhead of the copter firmware all of whose identified controller variables are instrumented: Average execution time of 48 soft real-time tasks in log scale with and without DisPatch's patch. Both unpatched and patched firmware meet the soft real-time deadlines except for the `one_hz_loop` task.

Figure 4.12 summarizes the (1) execution time without patches, (2) execution time with patches, and (3) soft real-time deadlines. Compared with the task execution time without patches, we observe 1.46% run-time overhead on average ranging from -8.1% to 9.7% in the copter model.[5]

The deviations in run-time overhead occur more frequently on the tasks with small execution time (e.g., 7.1 µs for `update_optical_flow`, and 7.1 µs for `arm_motors_check` on unpatched firmware) since they are more sensitive to the patching overhead. Among all the tasks, only `one_hz_loop` misses the soft real-time deadline. We found that task violates soft real-time deadline even without DisPatch's patches. Among all the tasks, we found that controller-related tasks such as `update_altitude` and `run_nav_updates` show relatively higher run-time overhead with 6.8% and 8.7% respectively. The higher run-time overhead is due to the presence of more number of controller variables which DisPatch instrumented. Overall, we observe 1.48% run-time overhead on average ranging from 1.41% to 1.55% on all of the four RAV control models with only one task violating the soft real-time deadline even without our patches.

For the helicopter model, we can observe the 1.41% run-time overhead on average ranging from -12.3% to 11.0% from 49 soft real-time tasks compared with the task execution time without patches. Those fluctuations appear more frequently on the tasks with small

[5]↑Note that non-deterministic inputs (e.g., GPS and GCS communication) cause fluctuation in run-time overhead.

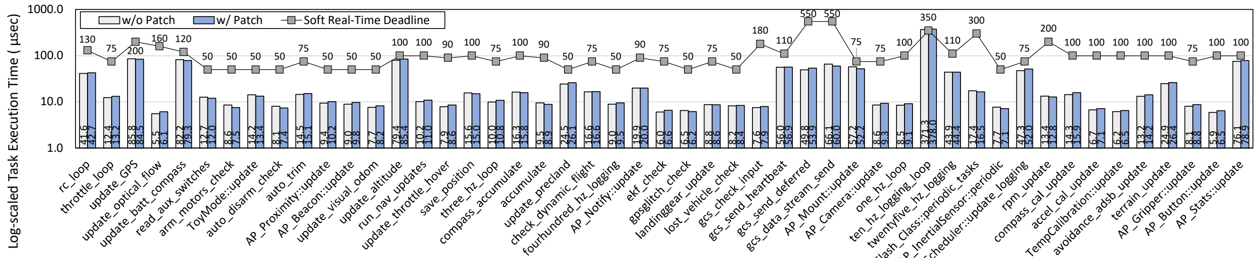


Figure 4.13. Run-time overhead of the helicopter firmware all of whose identified controller variables are instrumented: Average execution time of 49 soft real-time tasks in log scale with and without DisPatch's patch. Both unpatched and patched firmware meet the soft real-time deadlines except for the `ten_hz_logging_loop` task.

execution time (e.g., 8.6 µs for `arm_motors_check`, and 6.0 µs for `ekf_check` on unpatched firmware). Out of all of the tasks, only one task (`ten_hz_logging_loop`) misses the soft real-time deadline violation. However, this violation happens even with the firmware without patches. Out of tasks, we found that controller-related tasks (e.g., `update_altitude` and `run_nav_updates`) shows relatively higher run-time overhead (respectively 7.6% and 8.1%).

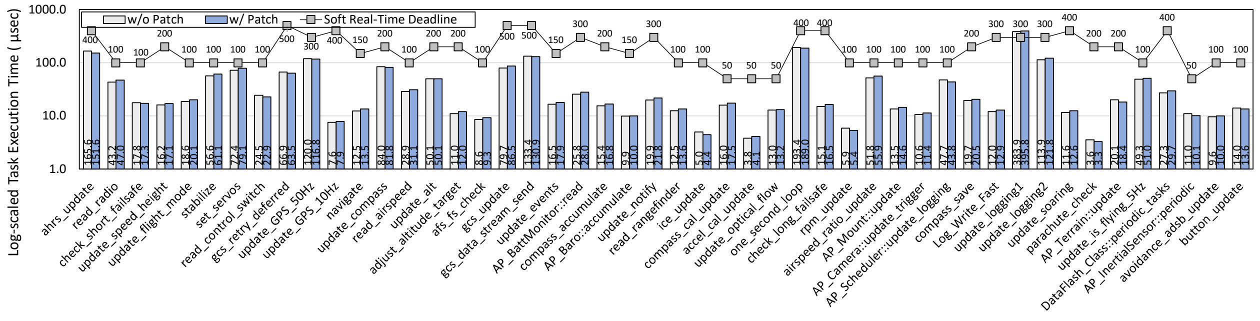


Figure 4.14. Run-time overhead of the plane firmware all of whose identified controller variables are instrumented: Average execution time of 50 soft real-time tasks in log scale with and without DisPatch's patch. Both unpatched and patched firmware meet the soft real-time deadlines except for the `update_logging1` task.

For the plane model, we can observe the 1.55% run-time overhead on average ranging from -11.4% to 9.7% from 50 soft real-times tasks of the plane firmware compared with the task execution time without patches. Those fluctuations appear more frequently on the tasks with small execution time (e.g., 3.6 µs for `parachute_check`, and 5.0 µs for `ice_update` on unpatched firmware). Out of all of the tasks, only one task (`update_logging1`) misses the soft real-time deadline violation. However, this violation happens even with the firmware without patches. Out of tasks, we found that controller-related tasks (e.g., `adjust_altitude_target` and `navigate`) shows relatively higher run-time overhead (respectively 8.9% and 8.4%).

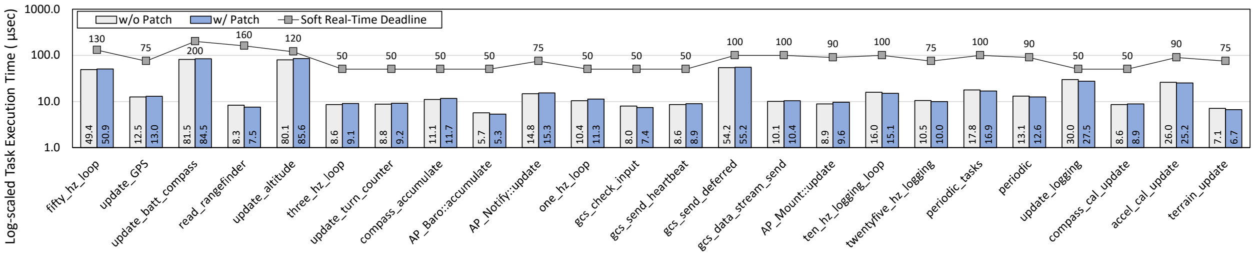


Figure 4.15. Run-time overhead of the submarine firmware all of whose identified controller variables are instrumented: Average task execution time of 24 soft real-time tasks in log scale with and without DisPatch's patch. Both unpatched and patched firmware meet the soft real-time deadlines.

For the submarine model, we can observe the 1.49% run-time overhead on average ranging from -9.6% to 8.5% from 24 soft real-times tasks of the submarine firmware compared with the task execution time without patches. Those fluctuations appear more frequently on the tasks with small execution time (e.g., 8.3 µs for `read_rangefinder`, and 8.9 µs for `AP_Mount::update` on unpatched firmware). Out of all of the tasks, both patched and unpatched firmware do not miss the soft real-time deadline violation. Out of tasks, we found that controller-related tasks (e.g., `update_altitude`) shows relatively higher run-time overhead (6.8%).

4.6 Discussion

Focused Control Variable Coverage. DisPatch focuses on critical PID controller variables required for operation such as inputs, outputs and parameters. These controller variables are the heart of the control system and applicable to any control system using PID controllers. The non-critical variables such as logging switch are not covered. We can cover additional mathematical components by adding their exact mathematical expressions to our mathematical expression template.

Generality of Primitive and Cascading Controller Dependencies. We inferred the primitive and cascading control dependencies from the RAV control model (Section 4.1), which are also shared by all the four control models that we tested. We also confirm that these dependencies hold true for other control systems (e.g., PX4 [23] and Crazyflies [110]). Although the internal details such as number, kind, and configuration of primitive controllers may vary in different control software, on the control model level they can be generalized to a generic controller dependency obtained from the control model (Section 4.1).

Multi-Architecture Support. Since the ARM architecture is the dominating in RAVs [22], [23], [110], DisPatch mainly targets the ARM architecture,and does not support RAV firmware running on other architecture such as x86 [75] out-of-the-box. However, our framework is built upon multi-architecture supported tools (e.g., angr and IDA Pro) and works on architecture-agnostic intermediate languages (e.g., LLVM bitcode). Therefore, DisPatch should be portable to other architectures with reasonable engineering effort.

Dynamically Generated, Obfuscated or Encrypted Firmware. We assume that the firmware can be disassembled before our technique is applied. This means we could not handle firmware whose actual binary instructions will only be available during the run-time due to packing, obfuscation, or encryption. Fortunately, such encodings are not popular in practice for RAV firmware [118], [140], [141] due to the limited computing resources available on RAVs.

Automatic Peripheral Path Discovery for Controller Function Identification. We assumed to have access to the locations of MMIO registers for sensors and actuators from publicly available information in order to identify the paths between sensors and actuators. Future work will leverage *peripheral models* [142], [143] to discover candidate MMIO addresses for peripherals in combination with the target board information.

5. RELATED WORK

5.1 Postmortem Robotic Aerial Vehicle Investigation

Mayday was inspired in part by the well-established aircraft accident investigation practices based on recorded flight data. We find it meaningful to establish a parallel practice of recording RAV flight data, in preparation for in-depth investigation of RAV accidents. Offline log analysis is an established method to investigate RAV operation problems. Based on flight logs recorded, existing analysis tools [33]–[35] can visualize sensor inputs, motor outputs, high-level controller states, and flight paths in the logs. The visualization helps investigators find the vehicle's physical and mechanical problems, such as sensor and motor failures and power problems. Some of these tools (e.g., LogAnalyzer [33]) also examine the correctness of some of the high-level controller states based on simple range checks (e.g., "from -45 to 45 degrees" for roll angle control), which can identify obvious problems without in-depth analysis. DROP [34] detects injected malicious commands based on the well-established DJI RAV framework. However, it focuses on finding a malicious command that appears only at the GCS or on-board the RAV, without performing cross-layer (i.e., from control and program) analysis. In comparison, Mayday performs cross-domain trace-back to RAV accident root causes by revealing the causality between physical impacts and control program bugs.

5.2 Program-Level Root Cause Analysis

Many root cause analysis techniques based on execution logs have been proposed to investigate program failures [37]–[40], security incidents [41]–[43], and for debugging [144]–[146].

Several solutions leverage program instrumentation to generate execution logs [37], [144]. On the other hand, there is a large number of works that record OS events during run-time and perform offline analyses to backtrack the provenance of Advanced Persistent Threat (APT) attacks [41]–[43]. These works leverage program execution partitioning [41], [43] and system event dependency models [41]–[43] to identify attack paths accurately in a large amount of log data from long-running systems. Another line of work records complete or partial execution until a program crashes and analyzes the logs to diagnose the root causes or reproduce the errors [37], [40]. Some of these works [38], [39] leverage hardware assistance [147] to log fine-grain program execution with high efficiency. Guided by RAV control model and control "model-to-program" mapping, MAYDAY achieves higher accuracy and efficiency for control program debugging.

Some debugging techniques such as statistical debugging techniques [145], [146] work by comparing the statistical code coverage patterns in "passing" and "failing" runs. However, bugs in control systems do not always induce obvious code coverage difference due to the iterative control-loop execution model, in which the same set of components (e.g., sensor reading sampling and control output generation) is periodically executed, with or without a controller digression. As such, for our target systems, they may not be as effective as for non-control programs.

5.3 Feedback-directed Testing and Fuzzing

RVFUZZER is inspired by many existing feedback-driven testing/fuzzing systems for conventional programs [148]–[162]. These solutions leverage different mutation strategies to increase the coverage of testing/fuzzing. Several systems [148]–[151] mutate input values with varying granularity (e.g., bit, byte-level) driven by the tested code's coverage achieved during each test run, using the code coverage as feedback. Another line of work [152], [153] adopts a hybrid approach to increase code coverage using both dynamic and symbolic execution. Finally, many efforts leverage taint analysis [154]–[158] or a combination of taint analysis and symbolic execution [159]–[162] for high testing coverage. Such approaches mutate inputs with awareness of the dependencies between program input and logic.

Testing techniques for conventional, non-cyber-physical programs rely on well-established mechanisms for (1) bug detection and (2) input mutation. Specifically, these testing techniques leverage generic, easy-to-detect symptoms of program failures (e.g., segmentation faults) as an indication of a triggered bug and mutate program input following information (e.g., code coverage) agnostic to domain semantics. Compared with conventional software testing, RVFuzzer addresses new problems and opportunities when finding cyber-physical bugs in RAV control. Many such bugs do not cause an immediate, easy-to-detect crash of the control program, especially when running with an RAV simulator. Meanwhile, control-theoretical properties offer hints to reduce the input value mutation space.

5.4 Disassembly and Function Identification

The capability of the traditional disassembly frameworks [121], [122], [163]–[166] cannot identify the *semantic meaning* of functions (e.g., z-axis velocity controller function). To identify a semantic meaning of a function, researchers have devised several approaches. Some identify such semantics of functions using static analysis information such as binary code signature pattern [120] or control flow graphs of functions [167]–[170]. Others take dynamic analysis approaches to identify semantic behavior of functions by leveraging execution traces [171]–[173]. There have been several works leveraging machine learning techniques [174]–[177]. However, none of them can find the target functions without reference binary function patterns. In addition, those binary patterns can be fragile if controller functions are customized.

There are three frameworks [12], [21], [178] targeting the controller function and variable identification. Hongjun et al. claim that they identify the controller variables. However, it cannot identify control parameters. Moreover, they leverage Valgrind [179] that requires general-purpose operating systems such as Linux and cannot run on both RTOSes and bare-metal firmware. ICSREF [178] identifies PID controllers using static binary signatures. However, it cannot handle customized PID controllers because it targets only PID controllers provided in libraries used only by specific industrial control system development tools. MISMO [21] can identify PID controllers using mathematical expressions with dynamic symbolic execution. However, MISMO is limited to finding only PID controller functions - not all of the different variants of the PID controller functions (e.g., P or PID FF controllers). Furthermore, it works only with x86 and PIN [180] which runs only on a general-purpose operating system. More importantly, both ICSREF and MISMO are limited to locating generic

PID controllers since they cannot identify their semantics (e.g., z-axis position controller) other than generic mathematical PID controller models.

5.5 Binary Rewriter

There have been a large body of works to patch the binary programs in general [117], [118], [179]–[187]. However, to patch the RAV binary programs, they fail to answer the following fundamental questions: (i) where the patch has to be applied, (ii) what patch has to be applied, and (iii) how easy it is for an operator to define and apply patches. In other words, they are limited to memory bugs and fail to identify and rewrite patches for cyber-physical bugs. They cannot patch RAV firmware without our controller-semantic patch location identifier. Furthermore, our framework provides a controller-specific DSL capable of patching semantic bugs in the controller functions and variables (e.g., z-axis velocity P parameter) with ease on top of the existing binary rewriting technique.

5.6 Run-time Control Semantics-Driven RAV Protection

There exists a body of work from attacks during flights and missions [12]–[14]. Blue-Box [13] detects abnormal behaviors of an RAV controller by running a shadow controller in a separate microprocessor that monitors the correctness of the primary controller, based on the same control model. CI [12] extracts control-level invariants of an RAV controller to detect physical attacks. Similarly, Heredia et al. [14] propose using a fault detection and isolation model extracted from a target RAV controller and enforces the model to detect anomalies during flights.

Another line of work focuses on deriving finite state models to detect abnormal controller behaviors [188], [189]. Orpheus [188] automatically derives state transition models using program analysis for run-time anomaly detection. Bruids [189] relies on a manual specification of RAV behaviors to derive a behavioral model to detect run-time anomalies.

Other approaches utilize machine learning techniques to derive benign behavioral models of an RAV controller. Abbaspour et al. [190] apply adaptive neural network techniques to detect fault data injection attacks during flight. Samy et al. [17] use neural network techniques to detect sensor faults. Two related efforts [15], [16] leverage a similar approach but detect both sensor and actuator faults.

Complementing the prior efforts, our frameworks have the following features: (1) MAYDAY is the control-model guided cyber-physical bug location system to *reactively* find cyber-physical bugs after its exploitation happens, (2) RVFUZZER leverages the control model and properties to *proactively* find cyber-physical bugs that may be exploited by RAV attackers, and (3) DISPATCH is the patching system to remove cyber-physical bugs after they are identified.

5.7 Defense against Cyber-Physical Bugs

Researchers have proposed a large number of solutions defending against the safety and security threats of RAVs. Some works apply filters such as the extended Kalman filters [44], [45] or low pass filters [46]. However, their capability is limited to mitigating occasionally missing or noisy sensor readings. Furthermore, they cannot handle threats that go beyond their filtering capabilities as shown in the previous works [12], [18], [19], [191].

Several efforts leverage an RAV control model to monitor the controller state anomaly [12], [13], [112], [191], [192]. Others leverage machine learning models to detect abnormal controller states [16], [17], [193] or incorrect computer-vision-based navigation [194], [195]. However, those efforts are mostly limited to either detecting controller state anomaly rather than controller state recovery [12], [13], [17], [192] or are targeting navigation systems [194], [195]A few recent research work [112], [191], [193] can defend against controller anomaly. However, they maintain the under-attack RAVs' controller state safety for a very limited time or show persistent controller anomaly; this makes RAVs vulnerable to sudden environmental changes (e.g., wind gust).

6. FUTURE WORK

My future research aims to tackle challenges in the security and safety of smart things and robotic aerial vehicles. Smart things (e.g., smart manufacturing, smart home devices, RAVs, and autonomous automotive) are physical objects connecting remotely with a wide range of smart devices to work together. This research direction will promote interdisciplinary collaboration with experts such as AI, control, and network experts to instantiate a new research paradigm.

6.1 Robotic Aerial Vehicle Security and Safety with AI

Artificial Intelligence (AI) is increasingly applied to various RAVs, including monitoring, testing, and navigating RAVs whose operations are determined by control software. In light of their close interaction, I plan to improve the security and safety of AI, control theory, and program techniques used in RAVs by complementing each other. For example, there is a well-known type of cyber attacks called advanced persistent threat (APT) attacks (e.g., Stuxnet). Such attacks are hard to detect because they gradually compromise a target system. Similarly, intelligent "cyber-physical" attackers will create APT-style stealthy attacks (which gradually corrupt a vehicle's control states). Unfortunately, the existing defense mechanisms, including even my current research work [18], [19], may miss those stealthy attacks. Hence, I plan to leverage AI models to prevent APT-style "cyber-physical" attacks at an early stage before the disruptive physical impact appears. By training AI models with various flight patterns, AI can derive a novel correlation between a vehicle's control states and sensors (e.g., a vehicle's control state inference from camera vision and approaching object detection from a pressure sensor). Such new findings can help detect stealthy attacks that existing approaches have missed. I also plan to improve AI's functionalities for RAVs by leveraging control and program knowledge. Specifically, control and program knowledge can help generate corner cases (e.g., fuzzing inputs (found by RVFuzzer [19]) making RAV controllers cause controller anomaly) to enhance AI training efficiency and fix latent buggy AI behavior. Such multi-disciplinary research will open new opportunities to collaborate with colleagues in AI, control, and programs to instantiate a new robotic aerial vehicle research paradigm.

6.2 Robotic Aerial Vehicle Swarm Security and Safety

RAV swarms have recently attracted significant attention from various fields such as military and urban applications (e.g., the drone show at the 2018 Winter Olympics). With the advent of vehicle-to-vehicle (V2V) communication with 5G, this trend will be accelerated. Considering the various applications of an RAV swarm and advances in communication technologies, I plan to research securing a group of RAVs and V2V communication, which is hard to be achieved by my previous work [10], [20] focusing on a single RAV's security and safety. One direction is that every RAV monitors the other RAVs in the swarm and fixes security and safety issues. For instance, if an RAV's sensor fails, adjacent RAVs can provide correct sensor values. Besides, if an RAV cannot communicate with its control station, a neighboring RAV fixes communication failure by acting as a communication repeater. Furthermore, I plan to work on lightweight, secure communication while preserving the real-time constraints. For example, network traffic reduction by de-duplicating messages in an RAV swarm network and lightweight encryption using special hardware are promising research directions.

6.3 Smart Things Security and Safety

The security and safety of emerging smart things have recently attracted significant attention. To secure smart things, I plan to generalize my "cyber-physical" analysis techniques (e.g., fuzzing and investigation). Specifically, smart things contain multiple controllers and their interaction performing physical operations as defined in the control model. For example, controllers in smart manufacturing systems (e.g., heating and molding metals) and autonomous automotive (e.g., accelerating and braking movements) are responsible for adjusting physical operations based on the underlying control models. I plan to improve those platforms' security and safety by adapting my "cyber-physical" analysis (e.g., CPS fuzzing and hardening) specialized to their respective control models.

7. CONCLUSION

It is challenging to prevent RAV accidents caused by cyber-physical bugs because we should secure RAVs considering the interplay between *"cyber"* and *"physical"* domains. However, none of the existing work secures the interplay between "cyber" and "physical" domains because they focus on only one of two domains. In this dissertation, we propose cyber-physical analysis to prevent attacks caused by cyber-physical bugs. More specifically, using program analysis and control modeling, I first developed novel techniques to (1) connect both cyber and physical domains and then (2) analyze their interplay. Based on our approach, I have developed MAYDAY for control program investigation, RVFUZZER for control program fuzzing, DISPATCH for control binary program patching. We summarize our three techniques as follows.

- MAYDAY is a cross-domain RAV accident investigation tool that localizes program-level root causes of accidents based on the RAV control model and enhanced in-flight logs. Guided by a generic RAV control model (CVDG), MAYDAY selectively instruments the control program to record its execution aligned with existing control-level logs. Using the control- and program-level logs, MAYDAY infers and maps the culprit control variable corruption from control domain to program domain and localizes the bug within a very small control program fragment.
- RVFUZZER is a control program testing system to proactively discover cyber-physical bugs in a control program binary. RVFUZZER can reveal illegitimate-yet-accepted value ranges of dynamically adjustable control parameters. Specifically, RVFUZZER adaptively mutates the input control parameter values to determine the (in)valid value ranges, driven by the detection of control state deviations in the simulated RAV. Furthermore, it considers the impact of external factors by mutating their values and presence.
- DISPATCH is the first end-to-end RAV firmware patching framework. DISPATCH allows end users to write human-readable patches and patch the cyber-physical

bugs within firmware directly with reasonable flexibility, high accuracy, and negligible overhead. The generality design of RAV firmware is applicable to different RAV models with different cyber-physical bugs that have been exploited to attack RAVs.

As a result, our work has discovered *91 new* cyber-physical bugs. Furthermore, 32 of cyber-physical bugs were confirmed, and 11 of them were patched by ArduPilot and PX4 developers. We believe our techniques will contribute to securely and safely using RAVs. This is because they cover the security aspects caused by the interplay between cyber and physical domains, which cannot be covered only with existing work.

REFERENCES

[1] *Usps drone delivery | cnbc*, 2015. [Online]. Available: https://www.youtube.com/watch?v=V9GXiXgaK34&list=PLL3t5xY2V44xOxvTIxS4AHuUhFE_bMwhz&index=36.

[2] *Dhl parcelcopter launches initial operations for research purposes*, 2014. [Online]. Available: http://www.dhl.com/en/press/releases/releases_2014/group/dhl_parcelcopter_launches_initial_operations_for_research_purposes.html.

[3] *Zipline's ambitious medical drone delivery in africa*, 2017. [Online]. Available: https://www.technologyreview.com/s/608034/blood-from-the-sky-ziplines-ambitious-medical-drone-delivery-in-africa.

[4] *How we're using drones to deliver blood and save lives*, 2017. [Online]. Available: https://www.youtube.com/watch?v=73rUjrow5pI.

[5] Y. M. Son, H. C. Shin, D. K. Kim, Y. S. Park, J. H. Noh, K. B. Choi, J. W. Choi, and Y. D. Kim, "Rocking drones with intentional sound noise on gyroscopic sensors," in *Proceedings of 24th USENIX Security symposium (Usenix Security)*, ser. Usenix Security '15, 2015.

[6] C. Yan, W. Xu, and J. Liu, "Can you trust autonomous vehicles: Contactless attacks against sensors of self-driving vehicle," *DEF CON*, vol. 24, 2016.

[7] T. Trippel, O. Weisse, W. Xu, P. Honeyman, and K. Fu, "Walnut: Waging doubt on the integrity of mems accelerometers with acoustic injection attacks," in *Proceedings of 2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, ser. EuroS&P '17, 2017.

[8] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (IEEE S&P)*, ser. IEEE S&P '13, 2013.

[9] T. Kim, V. Kumar, J. Rhee, J. Chen, K. Kim, C. H. Kim, D. Xu, and D. J. Tian, "Pasan: Detecting peripheral access concurrency bugs within bare-metal embedded applications," in *Proceedings of 30th USENIX Security Symposium (USENIX Security)*, 2021.

[10] C. H. Kim, T. Kim, H. Choi, Z. Gu, B. Lee, X. Zhang, and D. Xu, "Securing real-time microcontroller systems through customized memory view switching," in *Proceedings of the 27th Annual Symposium on Network and Distributed System Security (NDSS)*, 2018.

[11] T. Hardin, R. Scott, P. Proctor, J. Hester, J. Sorber, and D. Kotz, "Application memory isolation on ultra-low-power mcus," in *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, 2018.

[12] H. Choi, W.-C. Lee, Y. Aafer, F. Fei, Z. Tu, X. Zhang, D. Xu, and X. Deng, "Detecting Attacks Against Robotic Vehicles: A Control Invariant Approach," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[13] F. Fei, Z. Tu, R. Yu, T. Kim, X. Zhang, D. Xu, and X. Deng, "Cross-layer retrofitting of uavs against cyber-physical attacks," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2018.

[14] G. Heredia, A. Ollero, M. Bejar, and R. Mahtani, "Sensor and actuator fault detection in small autonomous helicopters," *Mechatronics*, vol. 18, no. 2, pp. 90–99, 2008.

[15] A. Abbaspour, P. Aboutalebi, K. K. Yen, and A. Sargolzaei, "Neural adaptive observer-based sensor and actuator fault detection in nonlinear systems: Application in uav," *ISA transactions*, vol. 67, pp. 317–329, 2017.

[16] H. A. Talebi, K. Khorasani, and S. Tafazoli, "A recurrent neural-network-based sensor and actuator fault detection and isolation for nonlinear systems with application to the satellite's attitude control subsystem," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 45–60, 2009.

[17] I. Samy, I. Postlethwaite, and D. Gu, "Neural network based sensor validation scheme demonstrated on an unmanned air vehicle (uav) model," in *Proceedings of 47th IEEE Conference on Decision and Control (CDC)*, 2008, pp. 1237–1242.

[18] T. Kim, C. H. Kim, A. Ozen, F. Fei, Z. Tu, X. Zhang, X. Deng, D. J. Tian, and D. Xu, "From control model to program: Investigating robotic aerial vehicle accidents with mayday," in *Proceedings of 29th USENIX Security Symposium (USENIX Security)*, 2020.

[19] T. Kim, C. H. Kim, J. Rhee, F. Fei, Z. Tu, G. Walkup, X. Zhang, X. Deng, and D. Xu, "Rvfuzzer: Finding input validation bugs in robotic vehicles through control-guided testing," in *Proceedings of 28th USENIX Security Symposium (USENIX Security)*, 2019.

[20] T. Kim, A. Ding, S. Etigowni, P. Sun, J. Chen, L. Garcia, S. Zonouz, D. Xu, and D. J. Tian, "Patching control-semantic bugs in rav firmware using dispatch," Planned to be submitted to one of the top-tier security conferences.

[21] P. Sun, L. Garcia, and S. Zonouz, "Tell me more than just assembly! reversing cyber-physical execution semantics of embedded iot controller software binaries," in *Proceedings of 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019.

[22] *Ardupilot*, 2020. [Online]. Available: http://ardupilot.org.

[23] *Px4 pro open source autopilot - open source for drones*, 2020. [Online]. Available: http://px4.io.

[24] *Paparazzi UAV - an open-source drone hardware and software project*, 2011. [Online]. Available: http://wiki.paparazziuav.org/wiki/Main_Page.

[25] Y. Li, J. M. McCune, and A. Perrig, "Viper: Verifying the integrity of peripherals' firmware," in *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, 2011.

[26] D. K. Nilsson, L. Sun, and T. Nakajima, "A framework for self-verification of firmware updates over the air in vehicle ecus," in *Proceedings of the 2008 IEEE GLOBECOM Workshops*, 2008.

[27] A. Ma, C. Dragga, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. K. Mckusick, "Ffsck: The fast file-system checker," *ACM Transactions on Storage (TOS)*, vol. 10, no. 1, 2014.

[28] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin, "Hafix: Hardware-assisted flow integrity extension," in *Proceedings of the 52nd Annual Design Automation Conference (DAC)*, 2015.

[29] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, "Hdfi: Hardware-assisted data-flow isolation," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*, 2016.

[30] *Exec shield*, 2005. [Online]. Available: https://static.redhat.com/legacy/f/pdf/rhel/WHP0006US_Execshield.pdf.

[31] M. Eagon, Z. Tu, F. Fei, D. Xu, and X. Deng, "Sensitivity-based dynamic control frequency scheduling of quadcopter mavs," in *Autonomous Systems: Sensors, Processing, and Security for Vehicles and Infrastructure 2019*, International Society for Optics and Photonics, vol. 11009, 2019, 110090A.

[32] *Ardupilot :: About*, 2020. [Online]. Available: https://ardupilot.org/about.

[33] *LogAnalyzer: Diagnosing problems using Logs for ArduPilot*, 2019. [Online]. Available: http://ardupilot.org/copter/docs/common-diagnosing-problems-using-logs.html.

[34] D. R. Clark, C. Meffert, I. Baggili, and F. Breitinger, "Drop (drone open source parser) your drone: Forensic analysis of the dji phantom iii," *Digital Investigation*, vol. 22, S3–S14, 2017.

[35] U. Jain, M. Rogers, and E. T. Matson, "Drone forensic framework: Sensor and data identification and verification," in *Proceedings of Sensors Applications Symposium (SAS)*, 2017.

[36] V. Sundaram, P. Eugster, and X. Zhang, "Efficient diagnostic tracing for wireless sensor networks," in *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2010.

[37] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "Sherlog: Error diagnosis by connecting clues from run-time logs," in *Proceedings of the 15th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[38] J. Xu, D. Mu, X. Xing, P. Liu, P. Chen, and B. Mao, "Postmortem program analysis with hardware-enhanced post-crash artifacts," in *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, 2017.

[39] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea, "Failure sketching: A technique for automated root cause diagnosis of in-production failures," in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, 2015.

[40] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu, "Pres: Probabilistic replay with execution sketching on multiprocessors," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, 2009.

[41] K. H. Lee, X. Zhang, and D. Xu, "High Accuracy Attack Provenance via Binary-based Execution Partition," in *Proceedings of the 20th Annual Symposium on Network and Distributed System Security (NDSS)*, 2016.

[42] Y. Ji, S. Lee, M. Fazzini, J. Allen, E. Downing, T. Kim, A. Orso, and W. Lee, "Enabling refinable cross-host attack investigation with efficient data flow tagging and tracking," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[43] S. Ma, X. Zhang, and D. Xu, "ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting," in *Proceedings of the 25th Annual Symposium on Network and Distributed System Security (NDSS)*, 2016.

[44] *Inertial Navigation Estimation Library*, 2016. [Online]. Available: https://github.com/priseborough/InertialNav.

[45] S. Habibi, "The smooth variable structure filter," *Proceedings of the IEEE*, vol. 95, no. 5, pp. 1026–1059, 2007.

[46] K. Ogata and Y. Yang, "Modern control engineering," 1970.

[47] *Amazon prime air*, 2017. [Online]. Available: https://www.amazon.com/Amazon-Prime-Air/b?node=8037720011.

[48] *MAVLink Micro Air Vehicle Communication Protocol*, 2020. [Online]. Available: https://mavlink.io.

[49] Y. Sui and J. Xue, "SVF: Interprocedural Static Value-flow Analysis in LLVM," in *Proceedings of the 25th International Conference on Compiler Construction (CC)*, 2016.

[50] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 26–60, 1990.

[51] T. Ball and J. R. Larus, "Efficient path profiling," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 1996.

[52] D. Graham and R. C. Lathrop, "The synthesis of optimum transient response: Criteria and standard forms," *Transactions of the American Institute of Electrical Engineers, Part II: Applications and Industry*, vol. 72, no. 5, pp. 273–288, 1953.

[53] F. Pukelsheim, "The three sigma rule," *The American Statistician*, vol. 48, no. 2, pp. 88–91, 1994.

[54] *Raspberry pi 3 model b*, 2021. [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-3-model-b.

[55] *SITL Simulator (ArduPilot Developer Team)*, 2014. [Online]. Available: http://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html.

[56] *Cppcheck - A tool for static C/C++ code analysis*, 2020. [Online]. Available: http://cppcheck.sourceforge.net.

[57] *Coverity Scan Static Analysis*, 2020. [Online]. Available: https://scan.coverity.com.

[58] K. H. Lee, X. Zhang, and D. Xu, "LogGC: Garbage Collecting Audit Log," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer Communications Security (CCS)*, 2013.

[59] Z. Liu, R. Sengupta, and A. Kurzhanskiy, "A power consumption model for multi-rotor small unmanned aircraft systems," in *Proceedings of the 2017 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2017.

[60] *Navio2*, 2021. [Online]. Available: https://emlid.com/navio.

[61] *Power Consumption of Raspberry Pi 3 Model B*, 2017. [Online]. Available: https://github.com/raspberrypi/documentation/blob/master/hardware/raspberrypi/power.

[62] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik, "A minimalist approach to remote attestation," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2014.

[63] *Tpm main specification*, 2018.

[64] S. Best, "Journaling file systems," *Linux Magazine*, vol. 4, pp. 24–31, 2002.

[65] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP)*, 2007.

[66] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing," in *Proceedings of International Workshop on Recent Advances in Intrusion Detection (RAID)*, 2008.

[67] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve, "Nested kernel: An operating system architecture for intra-kernel privilege separation," in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[68] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: Secure applications on an untrusted operating system," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[69] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "Trustlite: A security architecture for tiny embedded devices," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14, 2014.

[70] Y. Son, H. Shin, D. Kim, Y.-S. Park, J. Noh, K. Choi, J. Choi, and Y. Kim, "Rocking Drones with Intentional Sound Noise on Gyroscopic Sensors," in *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*, 2015.

[71] K. Jansen, M. Schäfer, D. Moser, V. Lenders, C. Pöpper, and J. Schmitt, "Crowd-gps-sec: Leveraging crowdsourcing to detect and localize gps spoofing attacks," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (SP)*, 2018.

[72] L. Heng, D. B. Work, and G. X. Gao, "Gps signal authentication from cooperative peers," *IEEE Transactions on Intelligent Transportation Systems*, vol. 16, no. 4, pp. 1794–1805, 2014.

[73] K. Pelechrinis, I. Broustis, S. V. Krishnamurthy, and C. Gkantsidis, "Ares: An anti-jamming reinforcement system for 802.11 networks," in *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2009.

[74] X. Liu, Y. Xu, L. Jia, Q. Wu, and A. Anpalagan, "Anti-jamming communications using spectrum waterfall: A deep reinforcement learning approach," *IEEE Communications Letters*, vol. 22, no. 5, pp. 998–1001, 2018.

[75] *Intel aero*, 2020. [Online]. Available: https://software.intel.com/en-us/aero.

[76] *3DR IRIS+*, 2018. [Online]. Available: https://3dr.com/support/articles/iris.

[77] *DJI Phantom 4 Advanced*, 2018. [Online]. Available: https://www.dji.com/phantom-4-adv.

[78] Y. Kwon, J. Yu, B. Cho, Y. Eun, and K. Park, "Empirical analysis of mavlink protocol vulnerability for attacking unmanned aerial vehicles," *IEEE Access*, vol. 6, pp. 43 203–43 212, 2018, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2863237.

[79] *Hijacking drones with a MAVLink exploit*, http://diydrones.com/profiles/blogs/hijacking-quadcopters-with-a-mavlink-exploit, 2015.

[80] N. Rodday, "Hacking a professional drone," *Blackhat ASIA '16*, 2016.

[81] N. M. Rodday, R. d. O. Schmidt, and A. Pras, "Exploring security vulnerabilities of unmanned aerial vehicles," in *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS)*, ser. NOMS '16, 2016.

[82] M. Vanhoef and F. Piessens, "Key reinstallation attacks: Forcing nonce reuse in wpa2," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, ser. CCS '17, 2017.

[83] A. A. Clements, N. S. Almakhdhub, S. Bagchi, and M. Payer, “ACES: Automatic compartments for embedded systems,” in *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[84] *Address space layout randomization*, 2001. [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt.

[85] F. Furrer, M. Burri, M. Achtelik, and R. Siegwart, “Rotors—a modular gazebo mav simulator framework,” in *Robot Operating System (ROS): The Complete Reference (Volume 1)*, 2016, pp. 595–625.

[86] F. L. Markley, J. Crassidis, and Y. Cheng, “Nonlinear attitude filtering methods,” in *Proceedings of the AIAA Guidance, Navigation, and Control Conference and Exhibit (AIAA)*, ser. AIAA ’05, 2005.

[87] D. Gebre-Egziabher, R. C. Hayward, and J. D. Powell, “Design of multi-sensor attitude determination systems,” *IEEE Transactions on aerospace and electronic systems*, vol. 40, no. 2, pp. 627–649, 2004.

[88] M. Jun, S. I. Roumeliotis, and G. S. Sukhatme, “State estimation of an autonomous helicopter using kalman filtering,” in *Proceedings of 1999 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, ser. IROS ’99, 1999.

[89] B. C. Kuo, *Automatic control systems.* Prentice Hall PTR, 1987.

[90] A. Nemati and M. Kumar, “Modeling and control of a single axis tilting quadcopter,” in *Proceedings of the American Control Conference (ACC)*, ser. ACC ’14, 2014.

[91] V. Praveen and S. Pillai, “A.,“modeling and simulation of quadcopter using pid controller”,” *International Journal of Control Theory and Applications (IJCTA)*, vol. 9, no. 15, pp. 7151–7158, 2016.

[92] Z. He, Y. Chen, Z. Shen, E. Huang, S. Li, Z. Shao, and Q. Wang, “Ard-mu-copter: A simple open source quadcopter platform,” in *Proceedings of the 2015 11th International Conference on Mobile Ad-hoc and Sensor Networks (MSN)*, 2015.

[93] N. P. Koenig and A. Howard, “Design and use paradigms for gazebo, an open-source multi-robot simulator,” in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2004.

[94] *QGroundControl - Intuitive and Powerful Ground Control Station for PX4 and ArduPilot UAVs*, 2018. [Online]. Available: http://qgroundcontrol.com.

[95] *MAVProxy - A UAV ground station software package for MAVLink based systems*, 2018. [Online]. Available: https://ardupilot.github.io/MAVProxy.

[96] *Pymavlink - A python implementation of the MAVLink protocol*, 2018. [Online]. Available: https://github.com/ArduPilot/pymavlink.

[97] *Librepilot*, 2020. [Online]. Available: https://www.librepilot.org.

[98] *SparkFun Autonomous Vehicle Competition 2013*, 2013. [Online]. Available: https://avc.sparkfun.com/2013.

[99] *ArduPilot Parameter List*, 2019. [Online]. Available: http://ardupilot.org/copter/docs/parameters.html.

[100] *PX4 Parameter List*, 2019. [Online]. Available: https://dev.px4.io/en/advanced/parameter_reference.html.

[101] Y. Son, J. Noh, J. Choi, and Y. Kim, "Gyrosfinger: Fingerprinting drones for location tracking based on the outputs of mems gyroscopes," *ACM Transactions on Privacy and Security (TOPS)*, vol. 21, no. 2, p. 10, 2018.

[102] M. Lothon, D. H. Lenschow, and S. D. Mayor, "Doppler lidar measurements of vertical velocity spectra in the convective planetary boundary layer," *Boundary-layer meteorology*, vol. 132, no. 2, pp. 205–226, 2009.

[103] R. Frehlich, Y. Meillier, M. L. Jensen, B. Balsley, and R. Sharman, "Measurements of boundary layer profiles in an urban environment," *Journal of applied meteorology and climatology*, vol. 45, no. 6, pp. 821–837, 2006.

[104] J. André, G. De Moor, P. Lacarrere, and R. Du Vachat, "Modeling the 24-hour evolution of the mean and turbulent structures of the planetary boundary layer," *Journal of the Atmospheric Sciences*, vol. 35, no. 10, pp. 1861–1883, 1978.

[105] *Rtca/do-178c*, Software Considerations in Airborne Systems and Equipment Certification, 2011.

[106] *Iso/iec 15408-1:2009*, https://www.iso.org/standard/50341.html, 2009.

[107] G. M. Qian, D. Pebrianti, L. Bayuaji, N. R. H. Abdullah, M. Mustafa, M. Syafrullah, and I. Riyanto, "Waypoint navigation of quad-rotor mav using fuzzy-pid control," in *Proceedings of Symposium on Intelligent Manufacturing & Mechatronics (IMM)*, 2018.

[108] X. Zhang, Y. Du, F. Chen, L. Qin, and Q. Ling, “Indoor position control of a quadrotor uav with monocular vision feedback,” in *Proceedings of the 37th Chinese Control Conference (CCC)*, 2018.

[109] E. Cano, R. Horton, C. Liljegren, and D. M. Bulanon, “Comparison of small unmanned aerial vehicles performance using image processing,” 2017.

[110] *Crazyflie 2.1*, 2020. [Online]. Available: https://www.bitcraze.io/products/crazyflie-2-1.

[111] M. Zhang and R. Sekar, “Control flow integrity for cots binaries,” in *Proceedings of the 22nd USENIX Security Symposium (USENIX Security)*, 2013.

[112] Z. Tu, F. Fei, M. Eagon, D. Xu, and X. Deng, “Flight recovery of mavs with compromised imu,” in *Proceedings of the 2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019.

[113] T. Abera, R. Bahmani, F. Brasser, A. Ibrahim, A.-R. Sadeghi, and M. Schunter, “Diat: Data integrity attestation for resilient collaboration of autonomous systems.,” in *Proceedings of the 28th Annual Symposium on Network and Distributed System Security (NDSS)*, 2019.

[114] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, *et al.*, “Sok:(state of) the art of war: Offensive techniques in binary analysis,” in *Proceedings of the 37th IEEE Symposium on Security and Privacy (IEEE S&P)*, IEEE, 2016.

[115] *Cmsis system view description*, 2020. [Online]. Available: http://www.keil.com/pack/doc/CMSIS/SVD/html/index.html.

[116] *Elftools: Engelmann’s libre fmri tools*, 2020. [Online]. Available: https://pypi.org/project/elftools.

[117] G. Hunt and D. Brubacher, “Detours: Binary interception of win32 functions,” in *Proceedings of the 3rd USENIX Windows NT Symposium*, 1999.

[118] T. Kim, C. H. Kim, H. Choi, Y. Kwon, B. Saltaformaggio, X. Zhang, and D. Xu, “Revarm: A platform-agnostic arm binary rewriter for security applications,” in *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC)*, 2017.

[119] *Retdec: A retargetable machine-code decompiler based on llvm*, 2020. [Online]. Available: https://github.com/avast/retdec.

[120] *Ida f.l.i.r.t. technology: In-depth — hex rays*, 2020. [Online]. Available: https://www.hex-rays.com/products/ida/tech/flirt/in_depth.

[121] *Hex-rays, ida pro disassembler*, 2016. [Online]. Available: http://www.hex-rays.com/products/ida.

[122] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2011.

[123] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, ser. CGO '04, 2004.

[124] L. Afonso, N. Souto, P. Sebastiao, M. Ribeiro, T. Tavares, and R. Marinheiro, "Cellular for the skies: Exploiting mobile network infrastructure for low altitude air-to-ground communications," *IEEE Aerospace and Electronic Systems Magazine*, vol. 31, no. 8, 2016.

[125] B. Gati, "Open source autopilot for academic research-the paparazzi system," in *Proceedings of the American Control Conference (ACC)*, 2013.

[126] *Ready-to-use & Easy-to-buy Vehicles — Copter Documentation 2020*, 2020. [Online]. Available: https://ardupilot.org/copter/docs/common-rtf.html.

[127] *Walkera technology co., ltd. (walkera)*, 2020. [Online]. Available: https://www.walkera.com.

[128] *Traxxas*, 2020. [Online]. Available: https://traxxas.com.

[129] *Birdseyeview aerobotics*, 2020. [Online]. Available: https://www.birdseyeview.aero.

[130] *Chibios — free embedded rtos*, 2020. [Online]. Available: http://chibios.org.

[131] *E386 – firmware download*, 2020. [Online]. Available: https://event38.com/knowledgebase/e386-firmware-download/?v=0a10a0b3e53b.

[132] *Aton and aton plus firmware update*, 2020. [Online]. Available: https://traxxas.com/products/models/heli/Aton?t=firmware.

[133] *Zoon by dronee — firmware update for dronee*, 2020. [Online]. Available: https://zoon.dronee.aero.

[134] *Pixhawk*, 2020. [Online]. Available: https://pixhawk.org/products.

[135] *Black magic probe*, 2020. [Online]. Available: https://github.com/blacksphere/blackmagic/wiki.

[136] *3dr*, 2020. [Online]. Available: https://3dr.com.

[137] *Auterion — the drone software platform built for enterprise*, 2020. [Online]. Available: https://auterion.com.

[138] *Parrot Bebop2*, 2018. [Online]. Available: https://www.parrot.com/global/drones/parrot-bebop-2.

[139] *Frsky djt2*, 2019. [Online]. Available: https://www.frsky-rc.com/product/djt-2.

[140] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware," in *Proceedings of the 22nd Annual Symposium on Network and Distributed System Security (NDSS)*, 2015.

[141] M. A. B. Khadra, D. Stoffel, and W. Kunz, "Speculative disassembly of binary code," in *Proceedings of the 2016 IEEE International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2016.

[142] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "Halucinator: Firmware re-hosting through abstraction layer emulation," in *29th USENIX Security Symposium (USENIX Sec)*, 2020, pp. 1–18.

[143] B. Feng, A. Mera, and L. Lu, "P 2 im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *Proceedings of the 29th USENIX Security Symposium*, 2020.

[144] P. Ohmann and B. Liblit, "Lightweight control-flow instrumentation and postmortem analysis in support of debugging," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013.

[145] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[146] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani, "Holmes: Effective statistical debugging via efficient path profiling," in *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, 2009.

[147] *Processor tracing*, 2013. [Online]. Available: https://software.intel.com/en-us/blogs/%202013/09/18/processor-tracing.

[148] *American fuzzy lop*, 2018. [Online]. Available: http://lcamtuf.coredump.cx/afl.

[149] *Libfuzzer*, 2018. [Online]. Available: https://llvm.org/docs/LibFuzzer.html.

[150] *Honggfuzz*, 2018. [Online]. Available: https://google.github.io/honggfuzz/.

[151] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "Kafl: Hardware-assisted feedback fuzzing for os kernels," in *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, ser. USENIX Security '17, 2017.

[152] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: Fuzzing by program transformation," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (IEEE S&P)*, ser. IEEE S&P '18, 2018.

[153] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proceedings of the 23rd Annual Symposium on Network and Distributed System Security (NDSS)*, ser. NDSS '16, 2016.

[154] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, ser. ICSE '09, 2009.

[155] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy (IEEE S&P)*, ser. IEEE S&P '10, 2010.

[156] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: Program-state based binary fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, ser. ESEC/FSE '17, 2017.

[157] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Angora: Efficient fuzzing by principled search," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, ser. ESEC/FSE '17, 2017.

[158] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing," in *Proceedings of the 25th Annual Symposium on Network and Distributed System Security (NDSS)*, ser. NDSS '18, 2018.

[159] V. Ganesh, T. Leek, and M. Rinard, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *Proceedings of the 22nd USENIX Security Symposium (USENIX Security)*, ser. USENIX Security '13, 2013.

[160] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (IEEE S&P)*, ser. IEEE S&P '12, 2012.

[161] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Proceedings of the 24th Annual Symposium on Network and Distributed System Security (NDSS)*, ser. NDSS '17, 2017.

[162] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (IEEE S&P)*, ser. IEEE S&P '15, 2015.

[163] E. Bauman, Z. Lin, and K. W. Hamlen, "Superset disassembly: Statically rewriting x86 binaries without heuristics.," in *Proceedings of the 27th Annual Symposium on Network and Distributed System Security (NDSS)*, 2018.

[164] *Capstone*, 2017. [Online]. Available: http://www.capstone-engine.org.

[165] S. Wang, P. Wang, and D. Wu, "Reassembleable disassembling," in *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*, 2015.

[166] K. Miller, Y. Kwon, Y. Sun, Z. Zhang, X. Zhang, and Z. Lin, "Probabilistic disassembly," in *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019.

[167] *Zynamics — bindiff*, 2020. [Online]. Available: https://www.zynamics.com/software.html.

[168] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, "Leveraging semantic signatures for bug search in binary programs," in *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC)*, 2014.

[169] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "Discovre: Efficient cross-architecture identification of bugs in binary code.," in *Proceedings of the 25th Annual Symposium on Network and Distributed System Security (NDSS)*, 2016.

[170] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 480–491.

[171] D. Kim, W. N. Sumner, X. Zhang, D. Xu, and H. Agrawal, “Reuse-oriented reverse engineering of functional components from x86 binaries,” in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014.

[172] D. Gao, M. K. Reiter, and D. Song, “Binhunt: Automatically finding semantic differences in binary programs,” in *International Conference on Information and Communications Security (ICICS)*, Springer, 2008.

[173] M. Egele, M. Woo, P. Chapman, and D. Brumley, “Blanket execution: Dynamic similarity testing for program binaries and components,” in *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*, 2014.

[174] P. Sun, L. Garcia, G. Salles-Loustau, and S. Zonouz, “Hybrid firmware analysis for known mobile and iot security vulnerabilities,” in *Proceedings of 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2020.

[175] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, “Neural network-based graph embedding for cross-platform binary code similarity detection,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 363–376.

[176] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, “Neural machine translation inspired binary code similarity comparison beyond function pairs,” in *Proceedings of the 29th Annual Symposium on Network and Distributed System Security (NDSS)*, 2020.

[177] S. H. H. Ding, B. C. M. Fung, and P. Charland, “Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization,” in *Proceedings of the 40th IEEE Symposium on Security and Privacy (IEEE S&P)*, 2019.

[178] A. Keliris and M. Maniatakos, “Icsref: A framework for automated reverse engineering of industrial control systems binaries,” in *Proceedings of the 27th Annual Symposium on Network and Distributed System Security (NDSS)*, 2018.

[179] N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation,” in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[180] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building customized program analysis tools with dynamic instrumentation,” in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, 2005.

[181] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Securing untrusted code via compiler-agnostic binary rewriting," in *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, 2012.

[182] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, "Pebil: Efficient static binary instrumentation for linux," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2010.

[183] B. Buck and J. K. Hollingsworth, "An api for runtime code patching," *Proceedings of the International Journal of High Performance Computing Applications (HPCA)*, 2000.

[184] Z. Deng, X. Zhang, and D. Xu, "Bistro: Binary component extraction and embedding for software security applications," in *Proceedings of the 18th European Symposium on Research in Computer Security (ESORICS)*, 2013.

[185] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (IEEE S&P)*, 2020.

[186] D. Williams-King, H. Kobayashi, K. Williams-King, G. Patterson, F. Spano, Y. J. Wu, J. Yang, and V. P. Kemerlis, "Egalito: Layout-agnostic binary recompilation," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[187] Wang, Shuai and Wang, Pei and Wu, Dinghao, "Uroboros: Instrumenting stripped binaries with static reassembling," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016.

[188] L. Cheng, K. Tian, and D. D. Yao, "Orpheus: Enforcing cyber-physical execution semantics to defend against data-oriented attacks," in *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC)*, ser. ACSAC '17, 2017.

[189] R. Mitchell and R. Chen, "Adaptive intrusion detection of malicious unmanned air vehicles using behavior rule specifications," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 44, no. 5, pp. 593–604, 2014.

[190] A. Abbaspour, K. K. Yen, S. Noei, and A. Sargolzaei, "Detection of fault data injection attack on uav using adaptive neural network," *Procedia computer science*, vol. 95, pp. 193–200, 2016.

[191] R. Quinonez, J. Giraldo, L. Salazar, and E. Bauman, "Savior: Securing autonomous vehicles with robust physical invariants," in *Proceedings of 29th USENIX Security Symposium (USENIX Security)*, 2020.

[192] G. Heredia, A. Ollero, M. Bejar, and R. Mahtani, “Sensor and actuator fault detection in small autonomous helicopters,” 2, vol. 18, Elsevier, 2008, pp. 90–99.

[193] F. Fei, Z. Tu, D. Xu, and X. Deng, “Learn-to-recover: Retrofitting uavs with reinforcement learning-assisted flight control under cyberphysical attacks,” in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2020.

[194] Y. Tian, K. Pei, S. Jana, and B. Ray, “Deeptest: Automated testing of deep-neural-network-driven autonomous cars,” in *Proceedings of the IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2018.

[195] K. Pei, Y. Cao, J. Yang, and S. Jana, “Deepxplore: Automated whitebox testing of deep learning systems,” in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.

VITA

Taegyu Kim earned a Ph.D. degree in the Department of Electrical and Computer Engineering at Purdue University, co-advised by Prof. Dongyan Xu and Prof. Dave (Jing) Tian in the Department of Computer Science. His main research interest lies in the security and safety of cyber-physical systems, with a focus on robotic aerial vehicles such as drones. Taegyu has published ten peer-reviewed papers including five papers at top security venues including USENIX Security, NDSS, and ACSAC. He received his M.S. at Korea Advanced Institute of Science and Technology (KAIST) and B.S. at Kwangwoon University in South Korea.