RVFUZZER: Finding Input Validation Bugs in Robotic Vehicles Through Control-Guided Testing

Taegyu Kim[†], Chung Hwan Kim[*], Junghwan Rhee[*], Fan Fei[†], Zhan Tu[†], Gregory Walkup[†]
Xiangyu Zhang[†], Xinyan Deng[†], Dongyan Xu[†]
[†]*Purdue University, {tgkim, feif, tu17, gwalkup, xyzhang, xdeng, dxu}@purdue.edu*
[*]*NEC Laboratories America, {chungkim, rhee}@nec-labs.com*

Abstract

Robotic vehicles (RVs) are being adopted in a variety of application domains. Despite their increasing deployment, many security issues with RVs have emerged, limiting their wider deployment. In this paper, we address a new type of vulnerability in RV control programs, called input validation bugs, which involve missing or incorrect validation checks on control parameter inputs. Such bugs can be exploited to cause physical disruptions to RVs which may result in mission failures and vehicle damages or crashes. Furthermore, attacks exploiting such bugs have a very small footprint: just one innocent-looking ground control command, requiring no code injection, control flow hijacking or sensor spoofing. To prevent such attacks, we propose RVFUZZER, a vetting system for finding input validation bugs in RV control programs through control-guided input mutation. The key insight behind RVFUZZER is that the RV control model, which is the generic theoretical model for a broad range of RVs, provides helpful semantic guidance to improve bug-discovery accuracy and efficiency. Specifically, RVFUZZER involves a control instability detector that detects control program misbehavior, by observing (simulated) physical operations of the RV based on the control model. In addition, RVFUZZER steers the input generation for finding input validation bugs more efficiently, by leveraging results from the control instability detector as feedback. In our evaluation of RVFUZZER on two popular RV control programs, a total of 89 input validation bugs are found, with 87 of them being zero-day bugs.

1 Introduction

Robotic vehicles (RVs), such as commodity drones, are a type of cyber-physical system for autonomous transportation. They are 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 to accomplish a given mission. RVs have emerged in various application domains such as commercial, industrial, entertainment, and law enforcement. For instance, logistics companies (e.g., USPS, DHL, and Amazon) have introduced drone delivery services to meet the rapidly growing demand in e-commerce [6, 10, 13, 27].

With their increasing adoption in real-world applications, RVs are facing threats of cyber and cyber-physical attacks that exploit their attack surface. More specifically, an RV's attack surface spans multiple aspects, such as (1) physical vulnerabilities of its sensors that enable external sensor spoofing attacks [72, 77, 80]; (2) traditional "syntactic" bugs in its control program (e.g., memory corruption bugs) that enable remote or trojaned exploits [75]; and (3) control-semantic bugs in its control program that enable attacks via remote control commands. For attacks exploiting (1) and (2), there have been research efforts in defending against them [30, 38, 40, 50, 52, 70, 76]; whereas those exploiting (3) have not received sufficient attention. As a result, the RV's attack surface in the aspects of (1) and (2) is expected to get smaller, which may prompt attackers to increasingly look at the control-semantic vulnerabilities for new exploits.

In this paper, we focus on an important type of control-semantic bugs in RV control programs, called *input validation bugs*. An input validation bug involves an incorrect or missing validity check on a control parameter-change input. Such an input is provided to the control program via a remote control command, which could trigger RV controller malfunction and ultimately lead to physical impacts on the vehicle, such as mission disruption, vehicle instability, or even vehicle damage/crash. Finding input validation bugs is a new research problem because they are largely orthogonal to the traditional "syntactic" bugs (e.g., buffer overflow and use-after-free bugs) which can be detected by existing software testing/fuzzing techniques.

Input validation bugs, on the other hand, are created semantically via incorrect setting of control parameters. In an RV, the control program can be configured through *control parameters*, which are adjustable numerical inputs that de-

termine certain aspects of the control function's behavior (e.g., controller gain and default flight speed). We can further categorize input validation bugs into two sub-categories: (1) Incorrect or missing parameter range checks in the control program, which would accept illegitimate setting of control parameter values, are called *range implementation bugs*. (2) Incorrect specification of control parameter ranges, even if correctly implemented, may cause RV controller malfunction. We call such specification-level errors (implemented in the control program) *range specification bugs*.

Most RVs have a remote control interface [21] for operators to set or adjust control parameters during a flight. Unfortunately, such an interface can be leveraged by attackers [9, 55, 67, 68] to exploit input validation bugs and deliberately mis-configure certain control parameters. As an example (details in Section 6.3.3), an RV control program allows operators to dynamically adjust a parameter for the vehicle's angular control and suggests a range of valid values in its specification. However, the range is erroneously determined and implemented. Knowing this bug, an attacker can issue a malicious command to reset the parameter using an illegitimate value that falls into the "valid" range, consequently crashing the vehicle.

Testing RV control programs to find input validation bugs is challenging. Popular RV control software (e.g., ArduPilot [15], PX4 [24], and Paparazzi [23]) supports many different RV models (e.g., quadcopters and ground rovers) with a large number of hardware, software and control configuration options. Generating accurate control parameter value ranges requires thorough testing of hundreds of control parameters for each RV model. With the growing number of RV models supported by control software, such testing is increasingly difficult to scale and automate. To overcome this challenge, especially without assuming source code access, one might think of leveraging automated black-box software testing methods, such as fuzzing [14, 17, 19, 71]. However, traditional software fuzzing techniques are not directly applicable to RV control programs because: (1) With hundreds of configurable parameters, the control program has an extremely large input space to explore and (2) there is no uniform and obvious condition to automatically decide that a control program is malfunctioning. Many input validation bugs do not exhibit system-level symptoms until certain control and physical conditions are met at run-time.

Our solution to finding input validation bugs – without control program source code – is motivated by the following ideas: (1) The impacts of attacks exploiting input validation bugs can be manifested by the victim vehicle's control state; and (2) such state can be efficiently reproduced by combining the RV control program and a high-fidelity RV simulation framework, which is readily available [7, 8].

Based on these ideas, we develop RVFUZZER, an automated RV control program testing system to find input validation bugs. RVFUZZER supports input-driven testing of a subject control program's binary, which runs in an RV simulator – for safety and efficiency. Unlike a traditional program bug (e.g., a memory corruption or divide-by-zero bug) that can result in an obvious program execution failure, automatically determining if the control program is ill-behaving based on the simulated vehicle's physical state is not straightforward. To address the problem, RVFUZZER involves a *control instability detector* based on a standard control stability measurement formula [47] to detect vehicle control malfunction. More importantly, RVFUZZER leverages this detector to quantify control (in)stability as feedback to guide input mutation, so that bugs can be found more efficiently by covering a large portion of the input space in a reasonable number of test runs. Our control-guided input mutation method is based on the following control property: When RV 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 such instability (Section 4.3.2). Finally, RVFUZZER mutates environmental factors such as trajectory curve or wind condition during testing, as attackers may leverage predictable environmental factors as probabilistic attack-triggering conditions.

We have implemented a prototype of RVFUZZER and applied it to ArduPilot [15] and PX4 [24], which are two popular RV control software suites used in many commodity RVs [32, 45, 58, 69]. RVFUZZER finds a total of 89 input validation bugs that can cause RV control malfunction: Two of them are known input validation bugs that were previously patched by developers; whereas the remaining 87 bugs are zero-day bugs which we have reported to the developers. In response to our report, eight bugs have been confirmed and seven of them have been patched. The contributions of our work are as follows:

- We introduce input validation bugs, a new type of RV control-semantic vulnerability that can be exploited by attackers.
- We develop RVFUZZER, a control-guided program vetting system to discover input validation bugs with safety, efficiency, and automation.
- We apply RVFUZZER to two popular RV control software suites and find 89 input validation bugs including 87 zero-day bugs.

2 Background

RV Control Model The RV 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 RV'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 (Fig. 1). The control model consists of multiple controllers, each for a specific degree of the 6DoF. For example, the x-axis

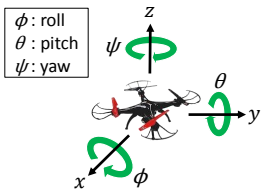

Figure 1: An RV's six degrees of freedom (6DoF).

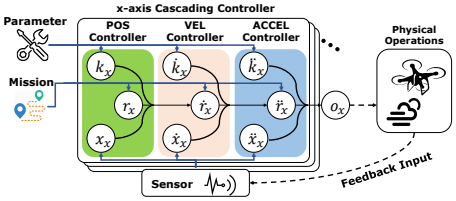

Figure 2: The x-axis controller (with three primitive controllers).

controller is shown in Fig. 2.

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 observed state ($x(t)$) reported by sensors. The goal of the controller is to keep the observed state close to the reference state, via its core function of *control 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 RV 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.

RV Control Program An RV control program implements the RV 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 RV communicates with a ground control station (GCS), which may issue a variety of GCS commands to the control program. Many of those commands allow RV 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 RV control programs have a run-time control state logging function, for record-keeping and troubleshooting purposes. Real-world commodity RVs (e.g., Intel Aero [18], 3DR IRIS+ [12], and DJI drone series [16]), as well as their simulators, log in-flight control states in persistent storage. RVFUZZER leverages such logs for automatic determination of controller malfunction.

Control Parameters Because of the complexity and generality of RV 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 [15], 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 Attack Model

Attack Model and Assumptions Attacks that exploit input validation 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 RV. Without correct input validation, the illegitimate parameter value will be accepted by the control program and cause at least one of the RV'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 6.3.3.

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 RV and GCS [9, 55, 67, 68, 78]. In the case of an insider threat, we assume that the insider is a rogue RV operator (not a developer), who does not have access to control program

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

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 [1,2,39,52]; (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 RV with minimum footprint[2]. This is because both the RV control program and control model already achieve a level of robustness for the RV 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 [11, 48]. Moreover, an internal attacker is also motivated to exploit illegitimate control parameter values that are erroneously considered normal in the RV's specification (i.e., range specification bugs), as the attacker could evade attack investigation by claiming that he/she was following RV control specification when issuing the command in question.

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

4 RVFuzzer Design

In this section, we present the design of RVFuzzer. We first give an overview of RVFuzzer's architecture (Section 4.1) and then present detailed design of two key components of RVFuzzer: (1) the control-guided instability detector that monitors the vehicle's control state to detect controller malfunction (Section 4.2) and (2) the control-guided input mutator that generates control program inputs for efficient program testing (Section 4.3).

4.1 Overview

RVFuzzer is designed to (1) detect physical instability of the RV during testing and (2) generate test inputs iteratively to achieve high testing efficiency and coverage. Fig. 3 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 RV control parameter-change commands; the subject control program, as the testing target, controls the operations of the (simulated) RV; and the simulator emulates the physical vehicle and its operating physical environment. We note that (1) the GCS and RV control programs are from real-world GCS and RV; and (2) our simulators [7, 8] are widely adopted for robotic vehicle design and testing.

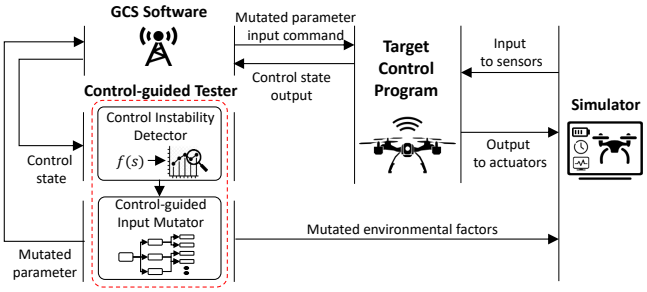

Figure 3: Overview of RVFuzzer.

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 physical disturbances of the target RV (e.g., crash and deviation), as indication of control program execution anomaly caused by an input validation 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 RV 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.

4.2 Control Instability Detector

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

We first define a rule to detect physical disturbances, which is tailored for input validation bugs. We then describe the mechanism to monitor the RVFuzzer's 6DoF control states for detecting such a disturbance.

Indication of Control State Deviation Exploitation of an input validation bug will cause an RV's failure to stabilize

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

its control states and/or complete its mission. To accurately detect bug-induced physical disturbance, RVFUZZER must be equipped with the capability of control state deviation detection. Among the possible physical disturbances experienced by an RV, there are two types of control state deviation: (1) observed state deviation and (2) reference state deviation. Accordingly, we define a detection rule to determine if one of the two types of control state deviation has occurred.

The first type – observed state deviation – is the case where a controller (e.g., the primitive x-axis velocity controller) fails to stabilize its observed 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 2). Consequently, if the difference between $x(t)$ and $r(t)$ keeps increasing and exceeds a certain threshold, the observed state will be considered deviating from the reference state. To quantify the observed state deviation, we leverage the integral absolute error (IAE) formula [47] which is widely used as a stability metric in control systems.

$$deviation(t) = \int_{t}^{t+w} \frac{|r(s) - x(s)|}{w} ds \tag{1}$$

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 control state deviation starting at t. We will describe how to experimentally determine w and τ for each 6DoF control state in Appendix A.

The second type – reference state deviation – is the case where an RV 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 control state deviation. An RV may experience transient control 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 [46, 51, 60].

Control Instability Detection We now apply our "observed-reference" and "reference-mission" deviation determination rule to detect control instability. During a test mission, the control program readily logs all its 6DoF control states (Section 2). 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). Note that the control 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 control state deviation at any primitive controller.

4.3 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 RV 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 (Section 4.3.1). We then present the feedback-driven input mutator which generates a reduced set of control parameter-change test inputs (Section 4.3.2). 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.

4.3.1 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 [15], PX4 [24], and Paparazzi [23]) all support a common parameter tuning interface defined in MAVLink [21], the de facto protocol for RV-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

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

in the input space search during mutation (Section 4.3.2). We note that the set of default values of control parameters is normally made available by RV vendors (e.g., 3DR, DJI, and Intel), as a guidance to RV users when tuning the control parameters.

4.3.2 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 input validation 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 [54], 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 disturbance, which will cause system response to degrade as the magnitude of the disturbance 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 4.2) 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 control state deviation by setting the values of all other parameters to their *default* values.

We present the one-dimensional mutation procedure in Algorithm 1. 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 1) 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 (Section 4.3.1); 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 control 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

Algorithm 1 One-dimensional Mutation.

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

1: **function** ONEDIMENSIONALMUTATION(M, P, E, τ) ▷ Main function
2: *Initialize R*
3: $R.max \leftarrow$ ONEMUTATION(M, P, E, τ, U) ▷ 'U': Upper-bound search
4: $R.min \leftarrow$ ONEMUTATION(M, P, E, τ, L) ▷ 'L': Lower-bound search
5: **return** R ▷ Return an invalid range of one parameter
6: **function** ONEMUTATION(M, P, E, τ, *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($test, test', bound, Dev$)

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.

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 2. In this algorithm, RVFUZZER utilizes the results from the one-dimensional mutation (Algorithm 1) 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 parameters to their lower/upper bound values leads to an extreme scenario which can potentially exacerbate the impact of the target parameter on the control state deviation.

After setting the values of the other parameters (Line 18), the mutator follows a procedure similar to the one-

Algorithm 2 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), control state deviation threshold set for all primitive controllers (τ)
Output: An invalid range for a target parameter (R)

1: **function** MULTIDIMENSIONALMUTATION(M, P_{target}, E, PS_{all}, τ) ▷ 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, τ)
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, τ)
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($R, Upper, Lower$)

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 control 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.

4.3.3 Environmental Factors

In real-world missions, the RV interacts with the physical environment with external factors such as physical obstacles and wind. Such factors influence RV's control state and performance. We note that an external factor (e.g., wind) could make an otherwise valid parameter value cause control 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 RV during its missions. The RV 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 control state deviation. To expose such input validation bugs, RVFUZZER defines and simulates RV missions in which the RV needs to avoid obstacles via sudden, sharp trajectory changes. An attack case triggered by obstacle avoidance will be presented in Section 6.3.3.

External disturbances such as wind and turbulence may also disrupt the RV'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 6.2.2. The attack case presented in Section 6.3.3 also exploits the wind condition.

5 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 RVs [49, 62, 64]. We point out that the implementation of RVFUZZER is not specific to a certain RV 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 RVs, 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 [12]. 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 [8] and Gazebo [7, 42, 53] for ArduPilot and PX4, respectively. We note that RVFUZZER's control instability detection and input mutation functions can easily inter-operate with these simulators via the interfaces between the simulators and the control and GCS programs.

GCS Program We used QGroundControl [26] and MAVProxy [22] 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 [25], which provides APIs to remotely control the RV via the MAVLink communication protocol [21]. 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 [23], DJI [16], and LibrePilot [20]. MAVLink supports a wide range of GCS commands (e.g., for mission assignment, run-time control 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* [5] mission which is an official mission provided by ArduPilot and used in autonomous vehicle competitions to test the control and mission execution capabilities of RVs. 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 [28, 29]. Such a list is usually provided in the Extensible Markup Language (XML) format in the source code and can be easily parsed.

6 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 input validation bugs (Section 6.1); (2) How do different input mutation schemes of RVFUZZER contribute to the discovery of input validation bugs (Section 6.2); and (3) How can RVFUZZER be applied to discover input validation bugs that would otherwise be exploited to launch stealthy attacks (Section 6.3).

6.1 Finding Input Validation Bugs

We present a summary of the input validation 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.

6.1.1 Classification of Input Validation 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 [28, 29]. Leveraging these public range specifications, RVFUZZER found a number of input validation bugs through the 8-hour testing session. We classify these input validation 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

Table 1: Summary of input validation 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

be maliciously provided and accepted by the program, causing control 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 control 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 control 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.

6.1.2 Detection of Input Validation Bugs

Table 1 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 B. For coherent presentation in Table 1, 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 1 shows that RVFUZZER detected a total of 89 input validation bugs (42 bugs in ArduPilot and 47 bugs in PX4). We note that some of the control parameters are associated with *both* range implementation and the range specification bugs. Hence, the total number of input validation 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 8 bugs and patched 7 of them. The remaining bugs are under review. 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 input validation bugs require more time and effort. This is because the exploitability of each input validation 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 input validation bugs.

6.1.3 Impact of Input Validation 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 control states. Appendix B 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 control 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 1). 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 control 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 input validation 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 observed 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 input validation bug in PX4 (Table 1). 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 observed state accordingly, the filter smooths out the sensor values and helps the controllers in robustly responding to physical interactions [73]. Hence, if an invalid value is assigned to a control parameter related to a sensor calibrator/-filter due to an input validation bug, the primitive controller that consumes the sensor values will compute a corrupted observed 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 1). 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 input validation 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.

6.2 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 input validation bugs.

6.2.1 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 Fig. 4, 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 (Fig. 4). 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 control 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 control state deviation under specific, non-default settings of the other parameters. In Fig. 4, 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 8).

6.2.2 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 the valid range of a pa-

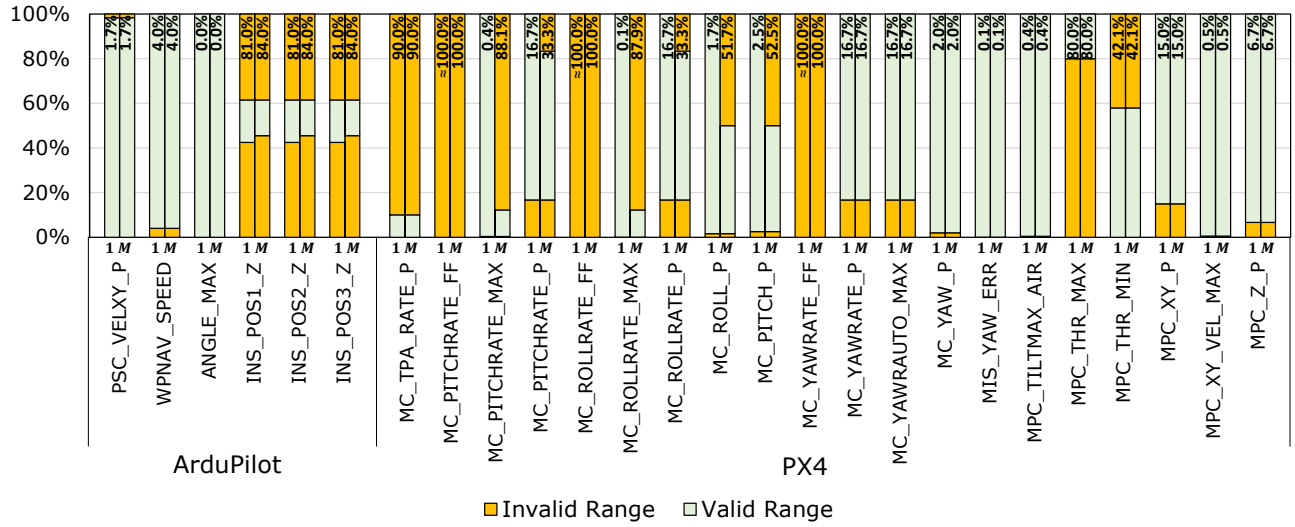

Figure 4: Invalid control parameter ranges discovered by RVFUZZER, normalized to the specified value ranges (**1**: One-dimensional mutation, ***M***: Multi-dimensional mutation). Percentage of invalid ranges (%) within the specified value ranges are noted at the top of the bars.

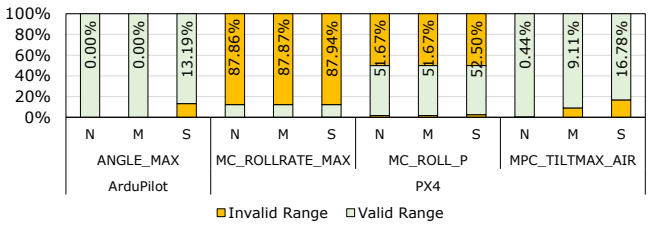

Figure 5: Normalized invalid ranges within the specified value ranges under different wind conditions (N: No wind, M: Medium wind, S: Strong wind).

rameter. 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 4.3.3. RVFUZZER found four cases which can be exploited with realistic environmental factors.

We performed tests based on existing wind analysis statistics [33, 41, 59] 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 5), which already reflects geographical factors as it emulates flight paths with sharp turns for obstacle avoidance.

Fig. 5 presents the impact of three different wind conditions on the four parameters which cause control state deviations. RVFUZZER discovered these four input validation 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 input validation 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.

6.3 Case Studies

We present three representative case studies of input validation 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 RV, and require different components of RVFUZZER's testing techniques to detect. Specifically, the bug discussed in Case I (Section 6.3.1) affects the x and

```
1 #define WPNAV_WP_SPEED_MIN 100  //Buggy code 2
2 #define WPNAV_WP_SPEED_MIN 20   //Patched code 2
3 ...
4 void AC_WPNav::set_speed_xy(float speed_cms){
5 -if(_wp_speed_cms>=WPNAV_WP_SPEED_MIN){  //Buggy code 1
6 +if(speed_cms>=WPNAV_WP_SPEED_MIN){     //Patched code 1
7     _wp_speed_cms = speed_cms;
8     _pos_control.set_speed_xy(_wp_speed_cms);
9     ...
```

Listing 1: Input validation 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.

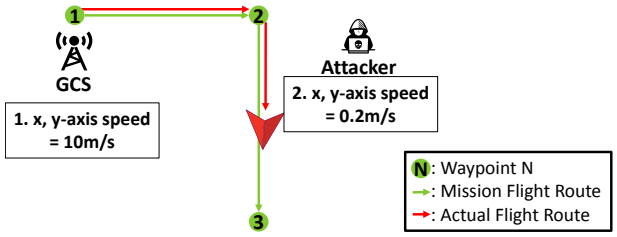

Figure 6: Illustration of Case Study I: An RV cannot recover its normal speed for the segment from Waypoint 2 to Waypoint 3.

y-axes controllers and causes unrecoverable slowdown, but can be discovered by RVFUZZER using the one-dimensional mutation technique. Case II (Section 6.3.2) 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 6.3.3) adversely affects the roll controller and causes significant deviation from the assigned mission, but can be discovered by mutating an environmental factor (wind force).

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

Attack We consider an RV that is assigned the mission of express package delivery (Fig. 6). Because of the urgency, the operator sets the RV's mission speed to 10 m/s at Waypoint 1. During the mission, while the RV slows down to make a turn at Waypoint 2, the attacker sends a seemingly innocent, but malicious, command to the RV 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 input validation bug in ArduPilot, illustrated in Listing 1.

Root Cause Listing 1 presents the code that runs in the RV 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

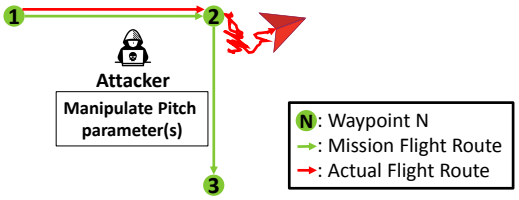

Figure 7: Illustration of Case Study II: The attack launched at Waypoint 2 causes an RV to oscillate due to failing control of the pitch angle.

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 RV 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 control state deviation, simulated and detected by RVFUZZER. Hence, RVFUZZER reported this deviation-triggering parameter as an input validation bug, which is confirmed by the related source code in Listing 1 (as ground truth of our evaluation).

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

Attack We consider an RV that is assigned the same mission as in Case Study I. As shown in Fig. 7, at Waypoint 2 of the mission, the attacker sends a malicious command to the RV to change one of the four pitch control parameters: `MC_PITCH_P`, `MC_PITCHRATE_P`, `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 RV to fail to stabilize its pitch angle, resulting in unrecoverable oscillation and deviation from its route.

Root Cause The unrecoverable oscillation on the RV'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 con-

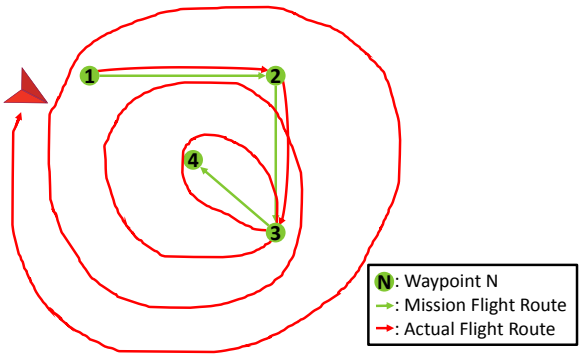

Figure 8: Illustration of Case Study III: An RV fails to complete a simple mission from Waypoint 1 to Waypoint 4 due to the impact of environmental factors.

trol 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 RV 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 RV's operations by corrupting the value of just *one* parameter. Let us assume that the RV 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 state and the observed state of the pitch angle with extreme sensitivity. As a result, the RV 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 2) 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 input validation bug.

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

Attack In this case study, we consider an RV assigned a mission to deliver a food item to a customer via the path presented in Fig. 8. The RV 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 RV 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 RV is supposed to make a 120-degree turn to avoid a tall building at Waypoint 3. However, the RV 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 control 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 5) 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 control state deviation between the reference state and the mission (Fig. 5). Hence, RVFUZZER reported this as an input validation bug contingent upon the influence of an external factor (wind).

7 Related Work

Control Semantics-Driven RV Protection There exists a body of work that leverages control semantics to protect RVs from attacks during flights and missions [38, 40, 50]. BlueBox [40] detects abnormal behaviors of an RV 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 [38] extracts control-level invariants of an RV controller to detect physical attacks. Similarly, Heredia et al. [50] propose using a fault detection and isolation model extracted from a target RV 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 [37, 61]. Orpheus [37] automatically derives state transition models using program analysis for run-time anomaly detection. Bruids [61] relies on

a manual specification of RV behaviors to derive a behavioral model to detect run-time anomalies.

Other approaches utilize machine learning techniques to derive benign behavioral models of an RV controller. Abbaspour et al. [31] apply adaptive neural network techniques to detect fault data injection attacks during flight. Samy et al. [70] use neural network techniques to detect sensor faults. Two related efforts [30, 76] leverage a similar approach but detect both sensor and actuator faults.

Complementing the prior efforts, RVFUZZER leverages control semantics to *proactively* find input validation bugs that may be exploited by RV attackers. Unlike the previous works that aim to detect abnormal behaviors *during* flights, our work focuses on identifying input validation bugs in RV control programs *before* flights via off-line RV simulation and program vetting. Control semantics are leveraged to reduce the input value mutation space and simulators are adopted to render the impacts of control parameter and external factor changes on control states.

Feedback-directed Testing RVFUZZER is inspired by many existing feedback-driven testing/fuzzing systems for conventional program testing [14, 17, 19, 34–36, 43, 44, 56, 57, 63, 66, 71, 74, 79]. These solutions leverage different mutation strategies to increase the coverage of testing/fuzzing. Several systems [14, 17, 19, 71] 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 [63, 74] adopts a hybrid approach to increase code coverage using both dynamic and symbolic execution. Finally, many efforts leverage taint analysis [36, 43, 56, 57, 79] or a combination of taint analysis and symbolic execution [34, 35, 44, 66] 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 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 (semantic) input validation bugs in RV control. Many such bugs do not cause an immediate, easy-to-detect crash of the control program, especially when running with an RV simulator. Meanwhile, control-theoretical properties offer hints to reduce the input value mutation space.

8 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 [4] and ISO/IEC 15408 [3]. To the best of our knowledge, however, there has been no standard safety testing framework created for RVs. 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 RV design and production.

9 Conclusion

Robotic vehicles (RVs) are facing cyber and cyber-physical attacks launched via various attack vectors. In this paper, we identify a new, small-footprint attack against RVs, where an attacker remotely issues a control parameter-change command with an illegitimate parameter value to disrupt the RV's control and mission. Such a value is erroneously accepted by the RV control program because of an input validation bug associated with the control parameter. The attack requires no code injection, control flow hijacking, or sensor spoofing hence cannot be defeated by existing RV security solutions. To proactively discover input validation bugs in a control program binary, we develop RVFUZZER, a control program testing system that reveals illegitimate-yet-accepted value ranges of dynamically adjustable control parameters. 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 RV. Furthermore, it considers the impact of external factors by mutating their values and presence. RVFUZZER has discovered 89 real input validation bugs in two of the most popular RV control software suites, with mutation efficiency and automation.

10 Acknowledgment

We thank our shepherd, Nolen Scaife and the anonymous reviewers for their valuable comments and suggestions. We also thank Vireshwar Kumar for his detailed feedback and edits which have improved the quality of the paper. This work was supported in part by ONR under Grant N00014-17-1-2045. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the ONR.

References

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

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

[3] *ISO/IEC 15408-1:2009*, 2009. https://www.iso.org/standard/50341.html.

[4] *RTCA/DO-178C*, 2011. Software Considerations in Airborne Systems and Equipment Certification.

[5] SparkFun Autonomous Vehicle Competition 2013, 2013. https://avc.sparkfun.com/2013.

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

[7] *Gazebo - A dynamic multi-robot simulator*, 2014. http://gazebosim.org.

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

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

[10] *USPS Drone Delivery | CNBC*, 2015. https://www.youtube.com/watch?v=V9GXiXgaK34&list=PLL3t5xY2V44xOxvTIxS4AHuUhFE_bMwhz&index=36.

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

[12] *3DR iris+*, 2018. https://3dr.com/support/articles/iris.

[13] *Amazone Prime Air*, 2018. https://www.amazon.com/b?node=8037720011.

[14] *American Fuzzy Lop*, 2018. http://lcamtuf.coredump.cx/afl.

[15] *ArduPilot*, 2018. http://ardupilot.org.

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

[17] *Honggfuzz*, 2018. https://google.github.io/honggfuzz/.

[18] *Intel Aero*, 2018. https://software.intel.com/en-us/aero.

[19] *libFuzzer*, 2018. https://llvm.org/docs/LibFuzzer.html.

[20] *LibrePilot*, 2018. https://www.librepilot.org.

[21] *MAVLink — Micro Air Vehicle Communication Protocol*, 2018. https://mavlink.io.

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

[23] *Paparazzi UAV - an open-source drone hardware and software project*, 2018. http://wiki.paparazziuav.org/wiki/Main_Page.

[24] *PX4 Pro Open Source Autopilot - Open Source for Drones*, 2018. http://px4.io.

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

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

[27] *Wing - Google X*, 2018. https://x.company/projects/wing.

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

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

[30] Alireza Abbaspour, Payam Aboutalebi, Kang K Yen, and Arman Sargolzaei. Neural adaptive observer-based sensor and actuator fault detection in nonlinear systems: Application in uav. *ISA transactions*, 67:317–329, 2017.

[31] Alireza Abbaspour, Kang K Yen, Shirin Noei, and Arman Sargolzaei. Detection of fault data injection attack on uav using adaptive neural network. *Procedia computer science*, 95:193–200, 2016.

[32] Luis Afonso, Nuno Souto, Pedro Sebastiao, Marco Ribeiro, Tiago Tavares, and Rui Marinheiro. Cellular for the skies: Exploiting mobile network infrastructure for low altitude air-to-ground communications. *IEEE Aerospace and Electronic Systems Magazine*, 31(8), 2016.

[33] JC 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*, 35(10):1861–1883, 1978.

[34] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (IEEE S&P)*, IEEE S&P '12, 2012.

[35] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (IEEE S&P)*, IEEE S&P '15, 2015.

[36] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing.

[37] Long Cheng, Ke Tian, and Danfeng Daphne Yao. Orpheus: Enforcing cyber-physical execution semantics to defend against data-oriented attacks. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC)*, ACSAC '17, 2017.

[38] Hongjun Choi, Wen-Chuan Lee, Yousra Aafer, Fan Fei, Zhan Tu, Xiangyu Zhang, Dongyan Xu, and Xinyan Deng. Detecting attacks against robotic vehicles: A control invariant approach. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, CCS '18, 2018.

[39] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. Aces: Automatic compartments for embedded systems. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[40] Fan Fei, Zhan Tu, Ruikun Yu, Taegyu Kim, Xiangyu Zhang, Dongyan Xu, and Xinyan Deng. Cross-layer retrofitting of uavs against cyber-physical attacks. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, ICRA '18, 2018.

[41] Rod Frehlich, Yannick Meillier, Michael L Jensen, Ben Balsley, and Robert Sharman. Measurements of boundary layer profiles in an urban environment. *Journal of applied meteorology and climatology*, 45(6):821–837, 2006.

[42] Fadri Furrer, Michael Burri, Markus Achtelik, and Roland Siegwart. Rotors—a modular gazebo mav simulator framework. In *Robot Operating System (ROS): The Complete Reference (Volume 1)*, pages 595–625. 2016.

[43] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, ICSE '09, 2009.

[44] Vijay Ganesh, Tim Leek, and Martin Rinard. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security)*, USENIX Security '13, 2013.

[45] Balazs Gati. Open source autopilot for academic research-the paparazzi system. In *Proceedings of the American Control Conference (ACC)*, ACC '13, 2013.

[46] Demoz Gebre-Egziabher, Roger C Hayward, and J David Powell. Design of multi-sensor attitude determination systems. *IEEE Transactions on aerospace and electronic systems*, 40(2):627–649, 2004.

[47] Dunstan Graham and Richard 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*, 72(5):273–288, 1953.

[48] Saeid Habibi. The smooth variable structure filter. *Proceedings of the IEEE*, 95(5):1026–1059, 2007.

[49] Zhijian He, Yanming Chen, Zhaoyan Shen, Enyan Huang, Shuai Li, Zili Shao, and Qixin 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)*, MSN '15, 2015.

[50] G Heredia, A Ollero, M Bejar, and R Mahtani. Sensor and actuator fault detection in small autonomous helicopters. volume 18, pages 90–99. Elsevier, 2008.

[51] Myungsoo Jun, Stergios I Roumeliotis, and Gaurav 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)*, IROS '99, 1999.

[52] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan 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.

[53] Nathan P Koenig and Andrew 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.

[54] Benjamin C Kuo. *Automatic control systems*. Prentice Hall PTR, 1987.

[55] Y. Kwon, J. Yu, B. Cho, Y. Eun, and K. Park. Empirical analysis of mavlink protocol vulnerability for attacking unmanned aerial vehicles. *IEEE Access*, 6:43203–43212, 2018.

[56] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Angora: Efficient fuzzing by principled search. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, ESEC/FSE '17, 2017.

[57] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, ESEC/FSE '17, 2017.

[58] Renju Liu and Mani Srivastava. Protc: Protecting drone's peripherals through arm trustzone. In *Proceedings of the 3rd Workshop on Micro Aerial Vehicle Networks, Systems, and Applications (DroNet)*, DroNet '17, 2017.

[59] Marie Lothon, Donald H Lenschow, and Shane D Mayor. Doppler lidar measurements of vertical velocity spectra in the convective planetary boundary layer. *Boundary-layer meteorology*, 132(2):205–226, 2009.

[60] F Landis Markley, John Crassidis, and Yang Cheng. Nonlinear attitude filtering methods. In *Proceedings of the AIAA Guidance, Navigation, and Control Conference and Exhibit (AIAA)*, AIAA '05, 2005.

[61] Robert Mitchell and Ray Chen. Adaptive intrusion detection of malicious unmanned air vehicles using behavior rule specifications. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 44(5):593–604, 2014.

[62] A. Nemati and M. Kumar. Modeling and control of a single axis tilting quadcopter. In *Proceedings of the American Control Conference (ACC)*, ACC '14, 2014.

[63] 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)*, IEEE S&P '18, 2018.

[64] Viswanadhapalli Praveen and S Pillai. A.,"modeling and simulation of quadcopter using pid controller". *International Journal of Control Theory and Applications (IJCTA)*, 9(15):7151–7158, 2016.

[65] Friedrich Pukelsheim. The three sigma rule. *The American Statistician*, 48(2):88–91, 1994.

[66] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the 24th Annual Symposium on Network and Distributed System Security (NDSS)*, NDSS '17, 2017.

[67] Nils Rodday. Hacking a professional drone. 2016.

[68] Nils Miro Rodday, Ricardo de O Schmidt, and Aiko Pras. Exploring security vulnerabilities of unmanned aerial vehicles. In *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS)*, NOMS '16, 2016.

[69] S Sabikan and SW Nawawi. Open-source project (osps) platform for outdoor quadcopter. *Journal of Advanced Research Design*, 24:13–27, 2016.

[70] Ihab Samy, Ian Postlethwaite, and Dawei 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)*, pages 1237–1242, 2008.

[71] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kafl: Hardware-assisted feedback fuzzing for os kernels. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, USENIX Security '17, 2017.

[72] Yun Mok Son, Ho Cheol Shin, Dong Kwan Kim, Young Seok Park, Ju Hwan Noh, Ki Bum Choi, Jung Woo Choi, and Yong Dae Kim. Rocking drones with intentional sound noise on gyroscopic sensors. In *Proceedings of 24th USENIX Security symposium (Usenix Security)*, Usenix Security '15, 2015.

[73] Yunmok Son, Juhwan Noh, Jaeyeong Choi, and Yongdae Kim. Gyrosfinger: Fingerprinting drones for location tracking based on the outputs of mems gyroscopes. *ACM Transactions on Privacy and Security (TOPS)*, 21(2):10, 2018.

[74] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 23rd Annual Symposium on Network and Distributed System Security (NDSS)*, NDSS '16, 2016.

[75] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (IEEE S&P)*, IEEE S&P '13, 2013.

[76] Heidar A Talebi, Khashayar Khorasani, and Siamak 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*, 20(1):45–60, 2009.

[77] 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)*, EuroS&P '17, 2017.

[78] Mathy Vanhoef and Frank Piessens. Key reinstallation attacks: Forcing nonce reuse in wpa2. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, CCS '17, 2017.

[79] 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)*, IEEE S&P '10, 2010.

[80] Chen Yan, Wenyuan Xu, and Jianhao Liu. Can you trust autonomous vehicles: Contactless attacks against sensors of self-driving vehicle. *DEF CON*, 24, 2016.

A Thresholds for Control State Deviation

We present how to determine the threshold values used by our control instability detector to detect control state deviation (Section 4.2). We use the AVC2013 mission (Section 5) and thirty other experimental missions in our experiments, similar to existing work [38]. Specifically, the thresholds are determined by applying the three-sigma rule [65] 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 control state is presented in Table 2.

We note that we do not monitor control state deviation in the second derivative states of the 6DoF (i.e., acceleration of any of the 6DoF). This is because, if their observed 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 control state deviation via the indirect impacts on the *dependent* states. The control 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.

Table 2: List of threshold values for each control 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	0.12 m/s
Roll	2.66 deg	1.98 deg
Roll Rate	2.83 deg/s	3.68 deg/s
Pitch	4.64 deg	3.94 deg
Pitch Rate	10.67 deg/s	15.35 deg/s
Yaw	4.13 deg	6.16 deg
Yaw Rate	16.24 deg/s	14.69 deg/s

Table 3: Input validation 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	✓			✓

B Physical Impacts Caused by Input Validation Bug Exploitation

We present more details about the input validation bugs found by RVFUZZER and the implications of the attacks that exploit them in Tables 3 (for ArduPilot) and Table 4 (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 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.

Table 4: Input validation 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				✓