From Control Model to Program: Investigating Robotic Aerial Vehicle Accidents with MAYDAY

Taegyu Kim[†], Chung Hwan Kim[*], Altay Ozen[†], Fan Fei[†], Zhan Tu[†]
Xiangyu Zhang[†], Xinyan Deng[†], Dave (Jing) Tian[†], Dongyan Xu[†]
[†]*Purdue University, {tgkim, aozen, feif, tu17, xyzhang, xdeng, daveti, dxu}@purdue.edu*
[*]*University of Texas at Dallas, chungkim@utdallas.edu*

Abstract

With wide adoption of robotic aerial vehicles (RAVs), their accidents increasingly occur, calling for in-depth investigation of such accidents. Unfortunately, an inquiry to "why did my drone crash" often ends up with nowhere, if the root cause lies in the RAV's control program, due to the key challenges in *evidence* and *methodology*: (1) Current RAVs' flight log only records high-level vehicle control states and events, without recording control program execution; (2) The capability of "connecting the dots" – from controller anomaly to program variable corruption to program bug location – is lacking. To address these challenges, we develop MAYDAY, a cross-domain post-accident investigation framework by mapping control model to control program, enabling (1) in-flight logging of control program execution, and (2) traceback to the *control-semantic bug* that led to an accident, based on control- and program-level logs. We have applied MAYDAY to ArduPilot, a popular open-source RAV control program that runs on a wide range of commodity RAVs. Our investigation of 10 RAV accidents caused by real ArduPilot bugs demonstrates that MAYDAY is able to pinpoint the root causes of these accidents within the program with high accuracy and minimum runtime and storage overhead. We also found 4 recently patched bugs still vulnerable and alerted the ArduPilot team.

1 Introduction

Robotic aerial vehicles (RAVs) such as quadrotors have been increasingly adopted in commercial and industrial applications – for example, package delivery by RAVs for Amazon Prime Air service [8]. Meanwhile, RAV accidents are increasingly reported, with undesirable consequences such as vehicle malfunction, instability or even crash [5, 6], calling for in-depth investigation of such accidents.

The causes of RAV accidents vary widely, including (but not limited to) (1) "physical" causes such as physical component failures, environmental disturbances, and sensor hardware limitations [4, 7, 32, 42, 70]; (2) generic bugs, such as buffer overflows, in the control program [26, 50]; (3) domain-specific *control-semantic bugs*, which arise from programming errors in implementing the underlying RAV control model in the control program. The first two categories involve hardware or software issues separately, while the third category entangles both the cyber and physical aspects of an RAV system. Meanwhile, current RAV's flight data recording can provide information to help trace back to physical causes of RAV accidents, but becomes much less informative when inspecting the control program internals. As a result, an inquiry to "why did my drone crash" often ends up with nowhere, if its root cause is a control-semantic bug.

As a motivating case (detailed investigation in Section 3 and 8.1.1), an RAV had been cruising normally at a constant speed, until it made a scheduled 90-degree turn when the vehicle suddenly became unstable and crashed afterward. It turns out that the root cause of the accident is a control-semantic bug – the control program's failure to check the validity of a control parameter (control gain), set either by the operator via the ground control station (GCS) or by an attacker via a remote parameter-changing command [51] during normal cruising – *long before* the crash. Such an accident is difficult to investigate. First, the physical impact on the vehicle did not happen immediately after the triggering event (i.e., parameter change), making it hard to establish the causality between them. Second, it is challenging to spot the triggering event among numerous states/events in the RAV's flight log. Third, it is non-trivial to locate the bug in the control program, which should be fixed to avoid future accidents.

The fundamental challenges preventing a successful root cause analysis for these accidents lie in *evidence* and *methodology*: (1) current RAV's flight log – generated by most control programs – records high-level controller states (e.g., position, attitude and velocity), without recording control program execution; (2) the capability of "connecting the dots" – from controller anomaly to program variable corruption to program bug location – is lacking.

To address these challenges, we develop MAYDAY, a cross-domain post-accident investigation framework by mapping control model to control program, enabling (1) in-flight logging of control program execution, and (2) traceback to the control-semantic bug that led to an accident, based on control-

and program-level logs.

More specifically, to enrich investigation evidence, MAYDAY first analyzes the control program and instruments it with selective program execution logging, guided by a *control variable dependency model*. This establishes a mapping between the control model and the program. To investigate an accident, MAYDAY performs a two-step investigation: (1) In *control-level investigation*, MAYDAY analyzes control-level log to identify (i) the controller (among the RAV's multiple controllers) that first went wrong and (ii) the sequence of control variable corruptions that had led to that controller's malfunction. (2) In *program-level investigation*, MAYDAY uses the output of (1) and the control model-program mapping to narrow the scope of program-level log to be analyzed, resulting in a very small subset of control program basic blocks where the root cause (bug) of the accident is located.

We have applied MAYDAY to ArduPilot [12], a popular open-source RAV control program [20,36] that runs in a wide range of commodity RAVs, such as Intel Aero, 3DR IRIS+, the Bebop series, and Navio2. Our investigation of 10 RAV accidents caused by real ArduPilot bugs demonstrates that MAYDAY is able to accurately localize the bugs for all the cases. Our evaluation results show that MAYDAY incurs low control task execution latency (3.32% on average), relative to the tasks' soft real-time deadlines. The volume of log generated by MAYDAY is moderate: 1.3GB in 30 minutes, which can easily be supported by lightweight commodity storage devices.

The contributions of MAYDAY lie in the *awareness and integration of RAV control model* for control program instrumentation, tracing, and debugging, which *specializes* these generic program analysis capabilities for more effective discovery of control-semantic bugs.

- For control program instrumentation (offline), we formalize the control model as a Control Variable Dependency Graph (CVDG). By establishing a *mapping* between the control model and the control program, we bridge the semantic gap between control- and program-level variables and data flows. We then develop an automatic instrumentation to enable CVDG-guided program execution logging.
- For control program tracing (runtime), we leverage the intrinsically low controller frequency for the cyber-physical RAV (tens/hundreds of Hz; versus its MHz/GHz processor), making it practical to employ the fine-grain control program execution logging at runtime. Such an approach would not be feasible for "cyber-only" systems due to the high relative overhead.
- For control program debugging (post-accident), We develop a *two-stage* accident investigation process, where the control-level investigation identifies the first malfunctioning controller and infers its control variable corruption path; and the program-level investigation will backtrack the program trace along that corruption path for bug localization, guided by the control "model-program" mapping.
- Our investigation of 10 RAV accidents caused by real ArduPilot bugs demonstrates that MAYDAY is able to accurately localize the bugs for all the cases with reasonable runtime and storage overhead. We also found 4 recently patched bugs that are still vulnerable and alerted the ArduPilot team.

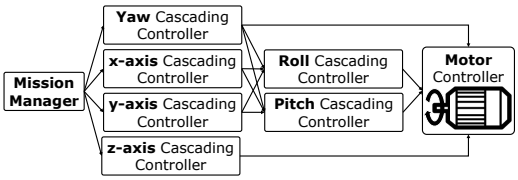


Figure 1: Dependencies of an RAV's Six degrees of freedom (6DoF) cascading controllers.

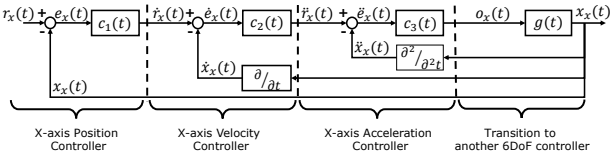


Figure 2: Primitive controllers in x-axis cascading controller.

We structured the rest of this paper as follows: Section 2 illustrates background on RAV control model and program and security model; Section 3 describes the motivating example of MAYDAY; Section 4 shows the overview of MAYDAY framework; Section 5 introduces CVDG and CVDG-guided program analysis and instrumentation techniques enabling cross-domain investigations; Section 6 illustrates our two-stage post-accident investigation process; Section 7 shows the detailed implementation of MAYDAY; Section 8 evaluates MAYDAY; Section 9 discusses the limitations and future work of MAYDAY; Section 10 summarizes the related work; Section 11 concludes MAYDAY.

2 Background and Models

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

Inside each 6DoF controller, a cascade of *primitive controllers* controls the position, velocity, and acceleration of that "degree", respectively. The control variables of these primitive controllers have dependencies induced by physical laws. Fig. 2 shows such dependencies using the x-axis controller as an example. For the x-axis position controller ($c_1(t)$, left-most), $x_x(t)$ is the *vehicle state* (i.e., position). $r_x(t)$ is the *reference* which indicates the desired position.

$e_x(t) = r_x(t) - x_x(t)$ is the *error*, namely difference between the state and reference. Intuitively, the goal of the controller is to minimize $e_x(t)$.

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

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

Trust Model and Assumptions MAYDAY is subject to the following assumptions: (1) We assume the soundness of the underlying RAV control model. (2) We assume that the RAV control program already generates high-level control log, which at least includes each primitive controller's reference, state and input. This is confirmed by popular RAV control programs ArduPilot [12], PX4 [16] and Paparazzi [15]. (3) We assume the integrity of logs and log generation logic in the control program, which can be enforced by existing code and data integrity techniques [55, 59, 61]. After a crash, we assume that the logs are fully recoverable from the vehicle's "black box". (4) We assume the control flow integrity of control program execution. Hence traditional program vulnerabilities/exploits, such as buffer overflow, memory corruption, and return-oriented programming, are outside the scope of MAYDAY. There exists a wide range of software security techniques to defend against such attacks [1, 31, 50, 71].

Soundness of Control Model To justify Assumption (1) of the trust model, we show that the underlying RAV control model adopted by ArduPilot is theoretically sound. For the model's *vehicle dynamics*, prior work [34] has analytically proved its correctness by modeling a standard rigid body system using Newton-Euler equations. For the model's *control algorithm* and *controller organization*, every primitive controller (e.g., those in Fig. 2) instantiates the classic PID (proportional-integral-derivative) algorithm; whereas all the controllers are organized in a dependency graph (CVDG, to be presented in Section 5.1), which reflects the classic RAV controller organization for controlling the vehicle's 6DoF.

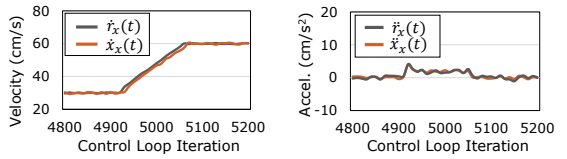
Based on the sound control model elements, the model's stability has been proved in prior work [33]. Furthermore, the control model – by design – tolerates vehicle dynamics changes (e.g., payload change) and disturbances (e.g., strong wind) to a *bounded* extent. We note that MAYDAY investigates accidents/attacks when the vehicle is operating *within* such bounds; and the triggering of the control-semantic bug will make an *originally sound* control model unsound, by corrupting its control/mission parameter(s), leading to instability of the system. Finally, theoretical soundness of the RAV control model is also testified to by its wide adoption by RAV vendors such as 3D Robotics, jDrone, and AgEagle for millions of robotic vehicles [20].

Threat and Safety Model MAYDAY addresses safety and security threats faced by RAVs, with a focus on finding control-semantic bugs in RAV control programs after accidents. These accidents may be caused by either safety issues (e.g., buggy control code execution or operator errors) or attacks (e.g., deliberate *negligence* or *exploitation* by a malicious insider). We assume that attackers know the existence of a control-semantic bug and its triggering condition. Then, an attacker may (1) continue to launch flight missions under the bug-triggering condition (e.g., strong wind) or (2) adjust vehicle control/mission parameters to create the bug-triggering condition (demonstrated in [51]). Action (1) requires the operator to simply "do nothing"; whereas action (2) will only leave a minimum bug-triggering footprint which could gradually corrupt controller states over a long period of time (Section 3). Such small footprint and long "trigger-to-impact" time gap make investigation harder. Furthermore, attacks exploiting control-semantic bugs do not require code injection, sensor/GPS spoofing, or blatantly self-sabotaging commands. As such, the security threat posed by control-semantic bugs is real to RAV operations and "exploit-worthy" to adversaries. All accident cases in our evaluation (Section 8) can happen in either accidental (i.e., safety) or malicious (e.g., security) context, reflecting the broad applicability of MAYDAY for RAV safety and security.

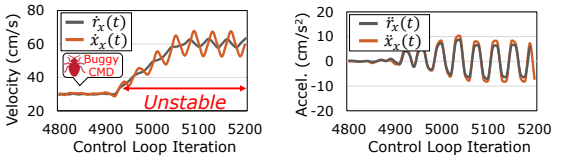
Meanwhile, accidents caused by either *physical* failures/attacks or generic software bugs are out of scope, as they have been addressed by existing efforts. For example, built-in logs can provide information for investigating either suspicious operator commands or physical attacks/accidents [19, 29, 45], without cross-layer (i.e., from control model to program) analysis (Section 9); and there has been a large body of solutions targeting generic software vulnerabilities [47, 49, 54, 60, 64, 73, 76, 77].

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

Figure 3: Motivating example flight. An RAV first flies to the north east with 60 cm/s (only in east, 30 cm/s) and then flies to east with 60 cm/s speed.



(a) Velocity without a bug. (b) Acceleration without a bug.



(c) Velocity with a bug. (d) Acceleration with a bug.

Figure 4: Controller states with and without the x-axis velocity parameter manipulation. The control loop iterates at a 10 Hz interval.

3 Motivating Example

Modern control programs are robust systems that operate while addressing and minimizing the impact of not only various physical non-deterministic factors (e.g., inertia and noise) but also control anomaly and security attacks [38, 48, 62]. However, we have found that such robustness is not enough to tackle all safety and security issues. Specifically, combined impacts of (i) operational inputs (e.g., mission, parameter changes) with (ii) particular altered physical conditions may go beyond the protection capability of a control system, which is an implication of a control-semantic bug. As a result, such impact starts to appear in a control variable of an exploited controller and will be propagated to its dependent controllers and can be *signified over the multiple control loop iterations*. To illustrate this, we introduce the following intuitive motivating accident case (more cases are discussed in Section 8) *only with* high-level control logs recorded by a built-in flight recorder.

In this example, we assume that our target RAV loads an item to deliver (as performed by real RAVs [8, 9, 11]) and flies to the north east with 60 cm/s (only in east, 30 cm/s) as described in Figure 3. At Iteration 4,850, the RAV operator increases Parameter P of x-axis velocity controller to make up for the weight gain. In the next 80 iterations of the control loop, the RAV continues to operate normally (i.e., the x-axis controller maintains a stable state). At a scheduled turn (i.e., flying east in Figure 3), the RAV is supposed to drastically decrease its x-axis velocity and to exhibit a behavior similar to that of the velocity and acceleration references depicted in Fig. 4a and Fig. 4b, respectively. However, at the junction, the changed parameter P unexpectedly leads to a corrupt state; the x-axis velocity started showing digression (Fig. 4c) and generating a corrupt x-axis acceleration reference. Consequently, the RAV completely failed to stabilize, ultimately resulting in a crash due to intensified digression over the multiple control loop iterations. We note that our example case is realistic because this accident can be triggered via a remote operational interface (e.g., MAVLink [13]).

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

4 MAYDAY Framework

MAYDAY spans different phases of an RAV's life cycle, shown in Fig. 5. In the offline phase, MAYDAY defines a formal description of the RAV control model, and uses it to enable CVDG-guided program-level logging during the control program execution via automatic instrumentation (Section 5). Then the RAV goes back into service with the instrumented control program, which will generate both control- and program-level logs during flights. In the case of an accident or attack, MAYDAY retrieves the logs and performs a two-stage forensic analysis, including control- and program-level investigations (Section 6). The investigations will lead to the localization of the control-semantic bug in the control program – the root cause of the crash.

5 Control-Guided Control Program Analysis and Instrumentation

This offline phase of MAYDAY formalizes a generic RAV control model using a Control Variable Dependency Graph (CVDG) (Section 5.1), which will guide the analysis (Section 5.2) and instrumentation (Section 5.3) of the control program, in preparation for the runtime program execution logging and the post-accident investigation (Section 6).

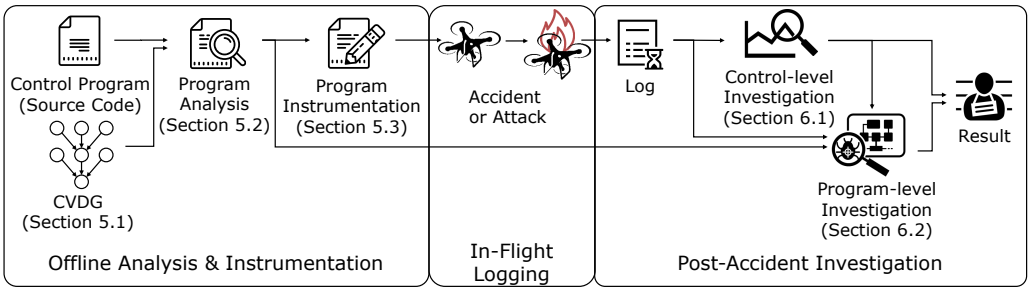


Figure 5: MAYDAY Framework.

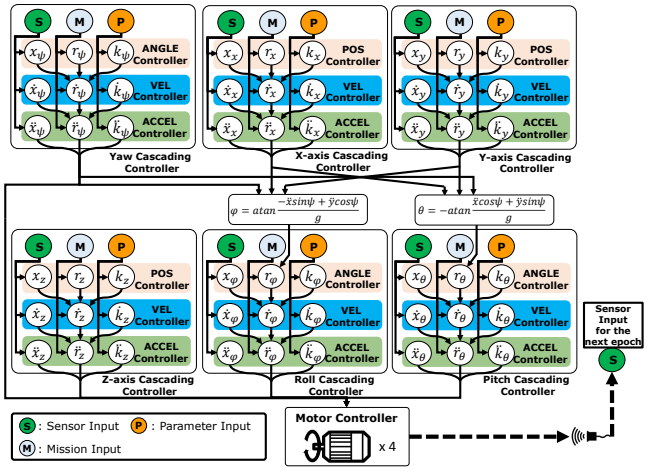


Figure 6: Control Variable Dependency Graph (CVDG).

5.1 Control Variable Dependency Graph

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

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

5.2 Mapping CVDG to Control Program

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

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

Our analysis is performed by Algorithm 1 at LLVM bitcode level. It is inter-procedural and considers pointer aliases of the control variables as well as other intermediate variables for completeness. It first performs a path-insensitive and flow-sensitive points-to analysis [72] to identify all aliases of the control variables (Line 2-3). For each alias identified, the algo-

Algorithm 1 Mapping CVDG edges to program code.

Input: Control variable set in the CVDG (CV)
Output: Mapping control variables to backward sliced instructions (M)

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

rithm performs backward slicing [44] to identify the program code that may influence the value of the control variable (Line 4, 10-23). As a result, each slice contains all the instructions that directly read or write the control variable and those that indirectly affect its value through some intermediate variables. Since the intermediate variables may have aliases not covered in the previous steps, Algorithm 1 recursively performs both points-to analysis and backward slicing on those variables to identify additional instructions that may affect the value of the control variable (Line 16-22). As new intermediate variables may be found in the identified slices during a recursion, this process will continue until no more affecting variable or alias exists.

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

5.3 Control Program Instrumentation

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

Efficient Logging of Program Execution A key requirement of control program execution logging is high (time and space) efficiency. MAYDAY meets this requirement via two methods. The first method is *selective basic block logging*. MAYDAY only instruments the basic blocks of the CVDG-mapped program code. For example, in ArduPilot, the CVDG-mapped basic blocks are about 40.08% of all basic blocks. The second method is *execution path encoding*, which involves inserting logging functions at proper locations to record *encoded* program execution paths. We adopt Ball-Larus (BL) algorithm [24] – an efficient execution path profiling technique with path encoding. Under BL algorithm, each execution path is associated with a path ID, which efficiently represents its multiple basic blocks in the order of their execution.

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

6 Post-Accident Investigation

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

6.1 Control-Level Investigation

The control-level investigation has two main steps: (1) identify which controller, among all the primitive controllers in the CVDG, was the *first* to go wrong during the accident (Section 6.1.1); (2) infer the possible sequence of control variable corruption, represented as a corruption path in the CVDG, that led to that controller's malfunction.

6.1.1 Initial Digressing Controller Identification

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

More formally, the malfunction of a controller manifests itself in two perceivable ways [51]: (1) non-transient digression between the control *state and reference* and (2) non-transient digression between the control *reference and mission input*. (1) means that the real state of the vehicle cannot "track" (i.e., converge to) the reference (i.e., desired state) generated by the controller; whereas (2) means that the reference cannot approach the target state set for the flight mission. As such, we

call the first controller that exhibited (1) or (2) the *initial digressing controller*; and we call the time when the digression started the *initial digressing time*.

To identify the initial digressing controller and time, MAYDAY examines the control log. Similar to [51], a sliding window-based digression check is performed on each primitive controller (1) between state and reference and (2) between reference and mission input. Unlike the previous work, MAYDAY uses the Integral Absolute Error (IAE) formula [37] in a distinct way to identify the initial digression in a *reverse* temporal order (details are discussed in Appendix B). By performing the digression check with the sliding window from the crash point backward, we identify the first digression window (hence time) of that controller, from which the digression persists toward the end of the log. The controller with the *earliest* first-digression window is the initial digressing controller.

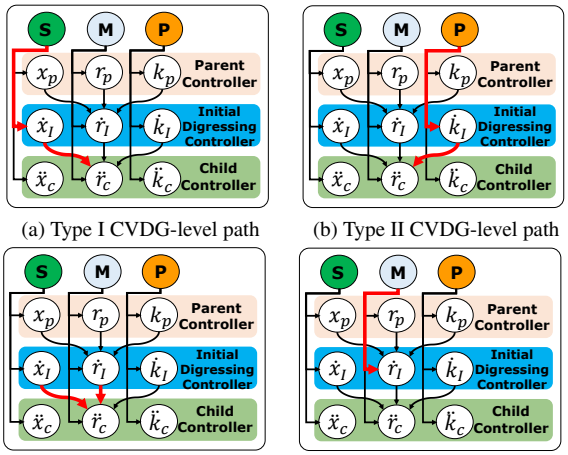
6.1.2 CVDG-Level Corruption Path Inference

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

We first define several terms. Each primitive controller has three inputs: sensor input S, flight mission M, and control parameter P, with M and P coming from ground control (GCS). x_I, r_I, and k_I denote the control state, reference, and parameter (a vector) of the initial digressing controller – denoted as C_I. x_c, r_c, and k_c denote the control state, reference, and parameter of C_I's child (i.e., immediate downstream) controller – denoted as C_c, respectively. Now we present the inference of CVDG-level corruption path as summarized in Figure 7.

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

- **Type I**: x_I was corrupted "locally" during the sensor input data processing (e.g., filtering). In the CVDG, such corruption corresponds to path $S \rightarrow x_I \rightarrow r_c$ as described in Figure 7a.
- **Type II**: x_I was corrupted indirectly via the control feedback loop. In this case, the control parameter k_I was first corrupted via GCS input (e.g., a parameter-changing command), which then corrupted r_c, the output of C_I. In C_c's effort to track the corrupted r_c, it generated the corrupted reference for its own child controller, and so on so forth. Finally, the RAV motors physically changed the vehicle's state, leading to the anomalous change of x_I. In the CVDG, such corruption corresponds to path $P \rightarrow k_I \rightarrow r_c$ as described in Figure 7b.
- **Type III**: x_I was similarly (to Type II) corrupted via the control feedback loop, due to the corruption of r_c. Unlike Type II, r_c's corruption was not triggered by external input. Instead, it was caused by some execution anomaly along CVDG edge $x_I \rightarrow r_c$ or $r_I \rightarrow r_c$ as described in Figure 7c.



(a) Type I CVDG-level path (b) Type II CVDG-level path

(c) Type III CVDG-level path (d) Type IV CVDG-level path

Figure 7: Summary of CVDG-level corruption paths according to different corruption types.

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

To determine if an accident is caused by Type I or II/III corruption path, MAYDAY needs to check if x_I is corrupted locally or indirectly. This is done by checking the *state consistency* between C_I and C_c (i.e., between x_I and x_c). Intuitively, the state consistency is an indication that C_c makes control decisions following the "guidance" – either right or wrong – of C_I; and the observation of C_c is consistent – according to physics laws – with that of C_I. For example, if C_I is a velocity controller and C_c is an acceleration controller, then x_I (velocity) is consistent with x_c (acceleration), provided that the observed velocity x_I closely matches the velocity computed using the actual acceleration x_c (via integration) in each iteration. [1] Since x_c did not digress from r_c when x_I digressed from r_I (by C_I's definition), if x_I and x_c are consistent, then we can infer that x_I is not locally corrupted and the CVDG-level path for x_I's corruption should be of Type II or III. Otherwise, the corruption path for x_I's corruption should be of Type I.

If the initial digression is between r_I and mission input M

[1] The formal definition of state consistency is given in Appendix A.

(Type IV), we can infer that a mission input (e.g., a GCS command to change trajectory or velocity) must have led to the change of r_I; and the new r_I value made C_I malfunction. In the CVDG, the corruption of r_I happened on path $M \rightarrow r_I$, as described in Figure 7d. Similar to Types I-III, we can prove that the parent controller of C_I cannot initially corrupt r_I.

In summary, Table 1 shows all four types of CVDG-level corruption paths and their determination conditions, to be applied during the investigation. Notice that the four types fully cover the CVDG edges in the initial digressing controller.

Table 1: Four types of CVDG-level corruption paths.

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

6.2 Program-Level Investigation

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

6.2.1 Transition to Program-Level Investigation

MAYDAY first makes the following preparations: (1) mapping the control variables on the CVDG-level corruption path to program variables, based on the control model $\rightarrow$ program mapping established during the offline analysis (Section 5); (2) locating the program trace for the initial digressing iteration – recall that the log has been indexed by control loop iteration number – as the starting point for (backward) log analysis; and (3) restoring the LLVM instruction trace from the encoded log for LLVM bitcode-level data flow analysis.

6.2.2 CVDG-Guided Program Trace Analysis

MAYDAY first identifies the data flows of program-level variable corruptions representing the CVDG-level corruption path. It runs Algorithm 2 to identify such data flows, starting from the initial digressing iteration and going backward. There are four inputs to Algorithm 2: (1) the restored LLVM bitcode-level program trace, indexed by control loop iteration number; (2) the initial digressing iteration number ($i_{digress}$); (3) the source and sink program variables that correspond to the start and ending nodes on the CVDG-level corruption path; [2] (4) the mapping between instructions in the trace and the program basic blocks they belong to. The output of Algorithm 2 is a small subset of control program basic blocks that may have been involved in the CVDG-level corruption path.

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

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

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

```
1:  P_prog ← BACKTRACK(P_cvdg, 0, i_digress)            ▷ Get program-level data flows
2:  i_trigger ← P_prog.i_start                          ▷ Control loop iteration with the triggering input
3:  while i_trigger ≤ i_digress do                      ▷ Find additional data flows
4:      i_digress ← P_prog.i_end − 1
5:      P_prog ← P_prog ∪ BACKTRACK(P_cvdg, i_trigger, i_digress)
6:  return GETBB(P_prog)
7:  function BACKTRACK(P_cvdg, i_start, i_end)
8:      P_prog ← ∅
9:      for e ∈ P_cvdg do
10:         P_prog ← P_prog ∪ BACKTRACKSRCSINK(e.src, e.sink, i_start, i_end)
11:     return P_prog
12: function BACKTRACKSRCSINK(src, sink, i_start, i_end)
13:     if src = sink then
14:         return ∅
15:     P_prog ← ∅
16:     for i ∈ {i_end ... i_start} do                  ▷ Backtrack the executed paths at every iteration
17:         P_i ← G.GETDATAFLOWPATHS(L[i], src, sink)   ▷ Between source and sink
18:         for p ∈ P_i do
19:             for sink_p ∈ p.sinks do                 ▷ Consider intermediate variables
20:                 P_prog ← P_prog ∪ BACKTRACKSRCSINK(src, sink_p, i_start, i)
21:     return P_prog
```

To explain Algorithm 2, we show a simple example in Fig. 8: The initial digressing controller is the x-axis velocity controller, and the CVDG-level corruption path is $P \rightarrow \dot{k}_x \rightarrow \ddot{r}_x$. The initial digressing time is Iteration 4930. P, $\dot{k}_x$, and $\ddot{r}_x$ are mapped to program variables (`msg`, `_pi_vel_xy._kp`, and `_accel_target.x`). Algorithm 2 starts from the sink variable (`_accel_target.x`) in Iteration $i_{digress}$ (4930) and finds a variable-corruption data flow from source variable `msg`, through intermediate variable `_pi_vel_xy._kp` (Line 1, 7-21), to sink variable `_accel_target.x`. Data flows that go through the intermediate variables (e.g., `_pi_vel_xy._kp`) are reconstructed using the additional sink information (Line 19-20). This information is retrieved via backward slicing (Line 17) as described in Section 5.2. In Fig. 8, the data flow is $P_{4850} \rightarrow \dot{k}_{x,4850} \rightarrow V_{4,4850} \rightarrow V_{8,4929} \rightarrow \ddot{r}_{x,4930}$, which realizes CVDG-level path $P \rightarrow \dot{k}_x \rightarrow \ddot{r}_x$. In particular, Iteration 4850 is the starting iteration of control variable corruption with the triggering input (P). We denote this iteration as $i_{trigger}$.

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

Once Algorithm 2 finds all the data flows of program-level

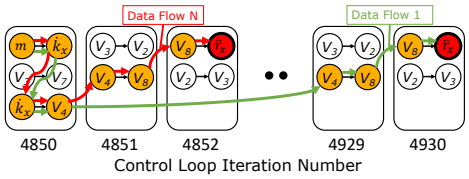


Figure 8: An example showing the working of Algorithm 2.

variable corruption, it can identify the corresponding basic blocks that implement each of the corrupting data flows (Line 6). In most cases, the multiple data flows will be mapped to the *same* set of program basic blocks, because of the iterative nature of control program execution. For example, the two corruption data flows in Fig. 8 share the common segment $V_4 \rightarrow V_8 \rightarrow \ddot{r}_x$ implemented by the same set of basic blocks. This helps keep the number of basic blocks reported by Algorithm 2 small, making it easy for investigators to examine the source code of those basic blocks to finally pinpoint the bug that caused the accident.

7 Implementation

We have implemented MAYDAY for an IRIS+ quadrotor with a Raspberry Pi 3 Model B (RPi) [17] as the main processor board powered by a 1.2 GHz 64 bit quadcore ARM Cortex-A53 CPU with 1 GB SDRAM. Attached to the RPi are a Navio2 sensor board and a 64 GB SD card. The sensor board has a number of sensors (GPS, gyroscope, barometer, etc.) and is equipped with four actuators and a telemetry radio signal receiver. The control program is the popular ArduPilot 3.4 on Linux 4.9.45, with the main control loop running at a default frequency of 400 Hz.

For MAYDAY's control program analysis (Section 5.2), we leverage the SVF 1.4 static analysis tool [72] for the points-to analysis. We modified SVF to support our inter-procedural backward slicing and control program instrumentation on LLVM 4.0. MAYDAY's control- and program-level investigation functions (Section 6) are implemented in Python 2.7.6. The entire MAYDAY system contains 10,239 lines of C++ code and 7,574 lines of Python code.

8 Evaluation

We evaluate MAYDAY's effectiveness with respect to RAV accident investigation (Section 8.1) and bug localization (Section 8.2); and MAYDAY's efficiency with respect to runtime, storage, and energy overhead (Section 8.3).

8.1 Effectiveness of Accident Investigation

Summary of Cases We investigated 10 RAV accidents based on *real* control-semantic bugs in ArduPilot 3.4. Table 2 summarizes the nature of the 10 accidents, with respect to categorization, physical impact, triggering condition, nature of control program bug, patching status, and vulnerability status. We chose these cases by the following criteria: (1) their root causes are real control-semantic bugs; (2) the specific nature of the bugs should be representative (e.g., invalid control/mission parameter values, integer overflow, and divide-by-zero); (3) the initial digressing controllers in these cases should cover all six degrees of 6DoF; and (4) the CVDG corruption paths in these cases should show diversity.

Specifically, Cases 1-4 are caused by controller parameter corruption, which corresponds to Type II CVDG-level path in Table 1 (Section 6.1) and results in unrecoverable vehicle instability, deviation, or even crashes. Cases 5-7 are caused by corruption of flight missions (e.g., location, velocity), which corresponds to Type IV CVDG-level path in Table 1. Cases 8-10 are caused by data (e.g., sensor or GCS input) processing errors such as divide-by-zero, which corresponds to either Type I (Case 10) or Type II (Cases 8-9) CVDG-level path in Table 1.

The root causes of these accidents are real control-semantic bugs that exist in ArduPilot 3.4 or earlier. The ones in Cases 5-10 are known bugs that have since been patched; whereas the bugs in Cases 1-4 still exist in the later version of ArduPilot 3.5. Our code review shows that the patches for those four bugs only fix the RAV's pre-flight parameter-check code, but not the in-flight parameter adjustment code. We alerted the ArduPilot team that the bugs in Cases 1-4 are not fully patched. Their reply was that, the four bugs were recently reported and confirmed along with other "invalid parameter range check" bugs. However, if ArduPilot fixes every parameter check, the firmware size may not fit in the memory of some resource-constrained micro-controllers supported by ArduPilot [3].

The "Patch Commit Number" column in Table 2 shows the patch commit numbers for all cases. Detailed ArduPilot bug-patching history, including the code snippets involved, can be accessed at: *https://github.com/ArduPilot/ardupilot/commit/[commit number]*.

Note that these accidents are not easy to reproduce or investigate. Their occurrences depend on vehicle-, control-, and program-level conditions. For example, the control program bugs may be triggered only when the vehicle takes a certain trajectory (Cases 1-4) and/or accepts a certain controller parameter or flight mission (Cases 1-9). Or they can only be triggered by a certain environment factor (e.g., wind speed in Case 10). Such accidents abound in real-world RAV operations [13]. Due to their hazardous nature and in compliance with safety regulations, we run these realistic accidents using a software-in-the-loop (SITL) RAV simulator [3], with a real control program and logs but simulated vehicle and external environment. Widely used in drone industry, the SITL simulator provides high-fidelity simulation of the vehicle as well as the physical environment it operates in (including aerodynamics and disturbances). We leverage MAVLink [13]

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

Table 2: List of accident cases caused by control-semantic bugs.

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

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

to trigger control-semantic bugs by issuing GCS commands to adjust control/mission parameters. MAVLink is able to communicate with both real and simulated RAVs.

Investigation Results Table 3 presents the results of our investigations using MAYDAY. For each case, MAYDAY first performs the control-level investigation, which identifies the initial digressing controller and infers the CVDG-level corruption path(s) by analyzing the control-level log. MAYDAY then performs then program-level investigation, which identifies the portion of control program code that implements the CVDG-level paths. We clarify that the final output of MAYDAY is not the specific buggy line of code per se. Instead, it is a small subset of program code (basic blocks) which the investigator will further inspect to pinpoint and confirm the bug.

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

Table 3: Investigation results of accident cases in Table 2. SLoC: Source lines of code.

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

Program-Level Investigation: The 5th and 6th columns of Table 3 show respectively the number of control program basic blocks and lines of source code identified by MAYDAY for each case. Notice that the numbers are fairly small (from 7 to 50 basic blocks, or 22 to 137 lines of code), indicating a low-effort manual program inspection. We confirm that the actual bug behind each case is indeed located in the code identified by MAYDAY.

Bug Detection Capability Comparison We have also conducted a comparative evaluation with (1) two off-the-shelf bug-finding tools: Cppcheck 1.9 [22] and Coverity [21], and (2) RVFuzzer [51], to detect the bugs behind the 10 accident cases. We used the most recent stable version of Cppcheck with all its available analysis options to leverage Cppcheck's full capability. For Coverity, we used its online service version. For RVFuzzer, we used its latest version. The results are shown in Table 4.

Comparison with Cppcheck and Coverity Neither Cppcheck nor Coverity reported any of the bugs behind the 10 cases. For Cases 1-6, without knowledge about the control model, it is impossible for Cppcheck and Coverity to check the validity of control/mission parameter input, or to determine if the RAV controller state – manifested by program state – is semantically valid or corrupted. For Case 7, the overflow of an integer program variable was not detected by either Cppcheck or Coverity. This was also confirmed by a Cppcheck developer[4]. For Cases 8-10, accurate detection of divide-by-zero bugs

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

Table 4: Bug detection capability comparison results. ✓: bug triggered and located in source code, Δ: bug triggered and faulty input constructed, and ✗: bug not detected.

Case ID	Nature of Bug	MAYDAY	Cppcheck [22]	Coverity [21]	RVFuzzer [51]
1	Missing controller parameter range check	✓	✗	✗	Δ
2	Missing controller parameter range check	✓	✗	✗	Δ
3	Missing controller parameter range check	✓	✗	✗	Δ
4	Missing controller parameter range check	✓	✗	✗	Δ
5	Comparison with a wrong variable	✓	✗	✗	Δ
6	Wrong waypoint computation based on non-existent coordinate	✓	✗	✗	✗*
7	Integer overflow on a mission variable	✓	✗	✗	Δ
8	Divide-by-zero caused by invalid controller parameter	✓	✗	✗	Δ
9	Divide-by-zero caused by invalid controller parameter	✓	✗	✗	Δ
10	(Probabilistic) Divide-by-zero caused by sensor input	✓	✗	✗	✗

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

is hard for static analysis-based tools such as Cppcheck and Coverity. Without a concrete execution confirming a divide-by-zero instance, they cannot detect such bugs with low false positive and false negative rates.

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

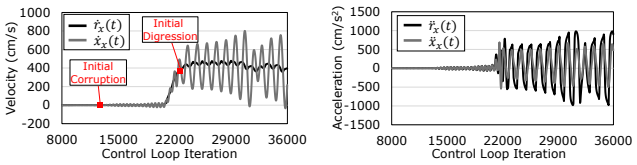
Even if a static analysis tool is aware of value ranges of control parameters, MAYDAY is still necessary because (1) there is no existing static analysis tool that comes with or generates a parameter-range specification; (2) static analysis is prone to high false positives/negatives when detecting divide-by-zero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

Comparison with RVFuzzer Among the 10 cases, RVFuzzer was able to trigger eight cases caused by GCS input validation bugs (i.e., lack of valid range check for runtime-adjustable control or mission parameters, as defined in [51]). RVFuzzer did not trigger Cases 6 and 10 for different reasons: (1) For Case 6, the reason is insufficient flight simulation time under RVFuzzer's default configuration. In this case, given an invalid input, RVFuzzer's simulation run terminated *before* controller anomaly could occur. However, RVFuzzer would have detected the bug in Case 6, if the simulation had run longer (for hours instead of minutes by default) for each input value. We note that RVFuzzer limits the simulation time to achieve high fuzzing throughput; and Case 6 manifests the trade-off between fuzzing coverage and throughput. (2) Case 10 cannot be detected by RVFuzzer because the bug is not a GCS input validation bug. Instead, it is triggered *probabilistically* by the wind speed sensor input.

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

More fundamentally, MAYDAY and RVFuzzer differ in two aspects: (1) MAYDAY reactively performs investigation to localize the bug in the *source code* that had led to an accident. MAYDAY involves CVDG-guided source code analysis and instrumentation to bridge the RAV control model and control program. RVFuzzer proactively discovers vulnerable inputs that cause controller anomalies, by treating the control *binary code* as a blackbox. (2) RVFuzzer automatically mutates values of control parameters that can be dynamically adjusted via GCS commands, to uncover vulnerable value ranges of those control parameters – namely *input validation bugs*. On the other hand, MAYDAY aims to trace back and pinpoint *control semantic bugs*, which include not only input validation bugs (e.g., Cases 1-4) but also other types of bugs such as flight mission corruption (e.g., Cases 6) and data processing error (e.g., Case 10).

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



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

Figure 9: Case 1: History of x-axis velocity and acceleration controllers – the former is the initial digressing controller.

8.1.1 Case Study: "Unexpected Crash after Turn"

We now present the investigations of Cases 1 and 5 as detailed case studies. In Case 1, the quadrotor's mission was to first stop at waypoint A to pick up a package, then fly straight north (along the y-axis) to waypoint B, where it would make a 90-degree turn to fly east (along the x-axis) to the destination. After the pickup, to maintain the y-axis speed (5 m/s) with the increased payload, the operator issued a parameter-changing command via GCS to increase the k_P parameter, shared by both x- and y-axis velocity controllers. The flight from A to B was normal. Unexpectedly, when the vehicle made the scheduled turn at B, it became very unstable and soon lost control and crashed.

MAYDAY first performs the control-level investigation. By analyzing the control-level log, MAYDAY finds that the initial digressing controllers are the x- and y-axis velocity controllers, both with digression between the vehicle velocity state ($\dot{x}_{xy}$) and reference ($\dot{r}_{xy}$) starting at around Iteration 23267 (after the scheduled turn at Iteration 20858). Fig. 9a shows the x-axis velocity state and reference. [5] Next, MAYDAY checks their child controllers (i.e., the x, y-axis acceleration controllers) and confirms that the child controllers did not exhibit any digression (i.e., $\ddot{x}_{xy}$ always tracked $\ddot{r}_{xy}$), even after the velocity controllers' digression. Fig. 9b shows the x-axis acceleration state and reference. Based on Table 1, MAYDAY infers that the CVDG-level corruption path is $P \rightarrow \dot{k}_{xy} \rightarrow \ddot{r}_{xy}$ (Type II).

MAYDAY then performs the program-level investigation. It runs Algorithm 2 on the program execution log, starting from Iteration 23267 and going backward, to find data flows that correspond to the CVDG-level corruption path. The multiple data flows found by the algorithm reveal that they all started from the parameter-changing GCS command (P), which led to the modification of k_P (which is part of $\dot{k}_{xy}$) during Iteration 13938 – much earlier than the digression (23267). k_P remained unchanged after Iteration 13938. Finally, MAYDAY maps the data flows to 34 basic blocks, among which we (as investigator) find the actual bug.

Listing 1 shows the code snippets with the bug. When a parameter-changing command is received, `set_and_save` saves the new parameter value. The value is later retrieved by `get_p`, when `rate_to_accel_xy` is called by the x, y-axis velocity controller. The code indicates that the controller

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

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

Listing 1: Control-semantic bug behind Case 1. The range check patch can be applied in Line 7.

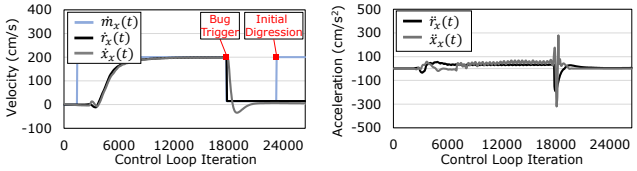
would accept any k_P value from the GCS without a range check! (A range check should be added at Line 7.) The relevant log also shows that, despite the improper k_P value, the vehicle remained stable from A to B. This is because the x- and y-axis velocity controllers are not sensitive to k_P under *constant*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

8.1.2 Case Study: "'Frozen' Velocity after Slowdown"

While Case 1 was caused by corruption of control parameters (Type II), Case 5 was triggered by corruption of flight mission (Type IV). We note that this case was first discussed by [51] as an attack scenario; and the corresponding vulnerability was found but *without* exact reasoning of the root cause (bug) at source code level. Here, we demonstrate how MAYDAY can locate the bug via post-accident/attack investigation.

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

MAYDAY first performs the control-level investigation. From the control-level log, it finds that the initial digressing controller is the x-axis velocity controller, with the digression between the velocity reference $\dot{r}_x$ and the operator-set velocity (which is part of mission M), starting from Iteration 23629 (Fig 10a). Different from Case 1, there is no digression between the x-axis velocity state ($\dot{x}_x$) and reference ($\dot{r}_x$), hence the vehicle did not lose control during the entire flight, despite the "frozen" speed. MAYDAY also confirms that the child controller (i.e., the x-axis acceleration controller) did not exhibit any digression (Fig 10b). In other words, both velocity and acceleration states correctly tracked their respective references



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

Figure 10: Case 5: History of x-axis velocity and acceleration controllers – the former is the initial digressing controller.

```
1  class AC_PosControl {
2  public:
3    float get_max_speed_xy() const { return _speed_cms; }
4  ...
5  void AC_WPNav::set_speed_xy(float speed_cms) {
6    // range check new target speed
7  - if(_pos_control.get_max_speed_xy() >=
8  -      WPNAV_WP_SPEED_MIN){ //Buggy code
9  + if(speed_cms >= WPNAV_WP_SPEED_MIN){ //Patched code
10     _pos_control.set_max_speed_xy(_wp_speed_cms);
11     // flag that wp leash must be recalculated
12     _flags.recalc_wp_leash = true;
```

Listing 2: Control-semantic bug behind Case 5.

and hence are consistent. Based on Table 1, MAYDAY infers that the CVDG-level corruption path is $M \rightarrow \dot{r}_x$ (Type IV).

Next, MAYDAY performs the program-level investigation. Starting from the program execution log at Iteration 23629 and moving backward. Algorithm 2 finds the data flow that corresponds to the CVDG-level corruption path: It started from the velocity-changing (from 2 m/s to 15 cm/s) command at Iteration 17736, which led to the modification of x-axis velocity reference ($\dot{r}_x$) at Iteration 17742. MAYDAY reports 12 basic blocks that may be involved in the data flow.

From the 12 basic blocks, we pinpoint the bug as shown in Listing 2. The code *intends* to enforce a minimum mission velocity (`WPNAV_WP_SPEED_MIN`, which is 20 cm/s in ArduPilot) through a range check on the *flight mission* velocity input (`speed_cms`) (Line 9, which is the patch). But the code, by mistake, compares the minimum mission velocity with the *current* velocity `_pos_control.get_max_speed_xy()`, not with the *set* velocity `speed_cms` (Line 7)! This bug caused the control program to accept the 15cm/s velocity, which is lower than the minimum mission velocity. Even worse, after this velocity change, the x-axis velocity controller will refuse to accept *any other* velocity change, because the result of the (buggy) comparison will always be FALSE. The 12 basic blocks identified by MAYDAY cover the buggy statement with the wrong variable name, which RVFuzzer [51] cannot report.

8.2 Scope Reduction for Bug Localization

As shown in Section 8.1, MAYDAY can significantly narrow down the scope of control program code for manual inspection to pinpoint a bug, thanks to 1) control model (CVDG)-guided corruption inference and 2) program execution logging. In this section, we define and implement a baseline investigation method *without* adopting these two ideas. We then compare

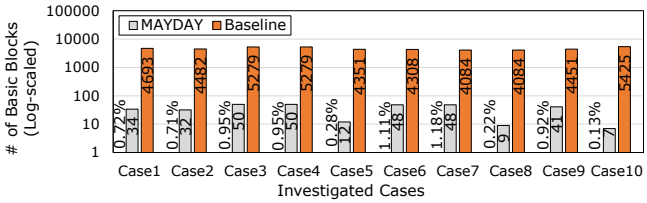


Figure 11: Number of basic blocks reported by the baseline investigation method and by MAYDAY.

MAYDAY with the baseline, with respect to the number of basic blocks they identify for bug localization.

The baseline model only analyzes the control program source code and control-level log. To its favor, we assume that the baseline method is able to identify at least one corrupted control variable based on the control-level log. From the corrupted variable, it performs static analysis (i.e., point-to analysis and backward slicing) to identify the corresponding basic blocks that implement the slice. Fig. 11 shows, in log scale, the number of basic blocks reported by the baseline method for each of the 10 cases in Section 8.1, comparing with MAYDAY. For each case, the baseline method reports thousands of basic blocks for bug localization; whereas MAYDAY reports tens of them. This comparison highlights the benefit (and novelty) of MAYDAY's control model guidance and program-level logging, which mitigates the long-existing problem of state explosion [53, 54] faced by generic program attack provenance.

8.3 Runtime, Storage and Energy Overhead

By identifying the basic blocks that implement the data flows in the CVDG (Section 5.3), we instrumented and logged 40.08% of the basic blocks in ArduPilot, introducing runtime, storage, and energy overheads. We measure these overheads using a *real quadrotor RAV*.

Runtime Overhead We measure the execution time of the 40 soft real-time tasks in ArduPilot during 30-minute flights with twenty random and different flight operations, with and without MAYDAY. The execution frequencies of the ArduPilot tasks vary, from 0.1 Hz to 400 Hz. The results are shown in Fig. 12, with each task's average execution time and its soft real-time deadline (defined in ArduPilot) in log scale.

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

We further breakdown the logging overhead between log generation (e.g., program path encoding) and I/O (writing to

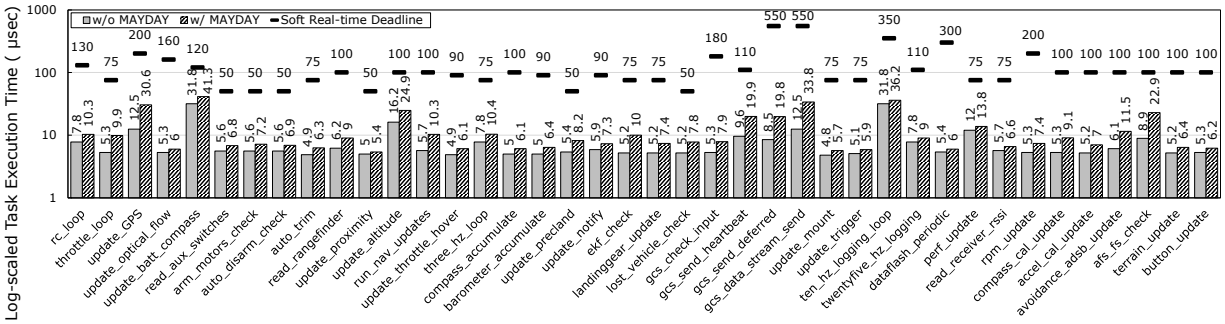


Figure 12: Runtime overhead of MAYDAY: average execution time of soft real-time tasks with and without MAYDAY in log scale. While MAYDAY introduces runtime overhead, it still meets the real-time requirement without missing deadlines.

SD card), as shown in Table 5. With a 400 Hz control loop frequency, MAYDAY's logging takes 7.6% of the time in one iteration – 190.72 μs in total. We note that such runtime fine-grain program tracing is feasible, thanks to the intrinsically low control frequency of cyber-physical systems, relative to that of their controller CPUs.

Table 5: Logging overhead breakdown.

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

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

Battery Consumption MAYDAY consumes fairly small amount of battery power, compared with the RAV motors. Our quadrotor is equipped with four motors whose total power consumption is approximately 147.5-177.5 Watts [58] excluding the computing board's power consumption (2.5 Watts). According to specifications, our sensor board consumes no more than 0.65 Watt [14], and its main processor board consumes a maximum of 5.0 Watts (less than 3.69% of the overall power consumption), with other attached devices (e.g., SD card) powered via the main processor board [10]. MAYDAY's power consumption is covered by the main processor board and therefore an even smaller fraction of the overall power consumption.

9 Discussion

Code and Log Protection We assume code integrity after instrumentation, log integrity, and log recover-ability in MAYDAY. To achieve code integrity, we can apply content-based integrity checking [55, 61] via remote attestation [18, 35]. We can also apply disk content integrity techniques [59] for log integrity. To recover from log corruption, special file system techniques (e.g., journaling file systems [25]) may be applied.

To protect kernel and flight data recording (FDR) modules at runtime, we could apply kernel hardening (e.g., SecVisor [69], NICKLE [68], and nested kernel [30]) and persistent data protection (e.g., InkTag [43]) techniques. However, many of those techniques are not suitable for resource-constrained RAV micro-controller platforms. Fortunately, there exist lightweight memory isolation techniques [39, 50, 52] that can protect security-critical modules (e.g., kernel and FDR) with low overhead. In particular, MINION [50] can be readily deployed with ArduPilot for memory access protection, even on low-end micro-controllers with only an MPU (memory protection unit). Additionally, we could consider Date Execution Prevention (DEP) [1] for thwarting code injection.

Log Volume Reduction We assume that the subject RAV has enough storage space to store logs in light-weight, low-cost devices such as commodity SD cards. However, future control programs may generate a larger volume of logs due to the complexity of their control algorithms and the fact that MAYDAY must record fine-grain, reproducible program execution paths/traces. Existing techniques reduce log size by (1) compressing the entire log [73] or (2) identifying and removing redundant log entries [53, 60]. Similar to (2), we plan to leverage control- and program-level dependencies to further reduce the log volume.

Scope of Applicability We clarify that, rather than being a generic bug-finding tool, MAYDAY specializes in finding RAV control-semantics bugs, which involve incomplete or incorrect *implementation* of the underlying control theoretical model. As acknowledged in Section 2, there exist other types of vulnerabilities in RAV systems, such as traditional program vulnerabilities and vulnerabilities in physical components (e.g., sensors). For physical attacks (e.g., sensor and GPS spoofing), MAYDAY is fundamentally not suitable, as the root cause of those attacks lies in the physical component (e.g., vulnerable sensing mechanism of a gyroscope device [70]), not in the control program. Hence MAYDAY's program exe-

cution trace analysis would not be necessary for detecting or investigating physical attacks.

Fortunately, defenses against physical attacks exist and can be deployed alongside with MAYDAY. Many sensor attacks can be detected by checking the RAV control log [19] for anomaly and inconsistency among sensors [70]. Physical sensor spoofing attacks can be detected by cross-checking the observed and expected controller states [28,34]. GPS spoofing attacks can be detected by commodity hardware (e.g., `u-blox M8`) and advanced techniques [41,46]. Jamming attacks can be defended against via existing solutions [57,66].

More Robust Control Models We acknowledge that more robust control models are technically possible and can make the RAV more tolerant of disturbances and changes. For example, a "self-examining" control algorithm can be designed to dynamically compute and verify the system's stability properties, in response to every GCS command. As another example, the PID control algorithm can be replaced by more advanced ones such as the Linear-quadratic regulator controllers [62] to better mitigate disturbances. However, such advanced control models are not yet widely adopted in commodity control programs (e.g., ArduPilot and PX4).

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

10 Related Work

Postmortem Robotic Vehicle Investigation MAYDAY was inspired in part by the well-established aircraft accident investigation practices based on recorded flight data. We find it meaningful to establish a parallel practice of recording RAV flight data, in preparation for in-depth investigation of RAV accidents. Offline log analysis is an established method to investigate RAV operation problems. Based on flight logs recorded, existing analysis tools [19, 29, 45] can visualize sensor inputs, motor outputs, high-level controller states, and flight paths in the logs. The visualization helps investigators find the vehicle's physical and mechanical problems, such as sensor and motor failures and power problems. Some of these tools (e.g., LogAnalyzer [19]) also examine the correctness of some of the high-level controller states based on simple range checks (e.g., "from -45 to 45 degrees" for roll angle control), which can identify obvious problems without in-depth analysis. DROP [29] detects injected malicious commands based on the well-established DJI RAV framework. However, it focuses on finding a malicious command that appears only at the GCS or on-board the RAV, without performing cross-layer (i.e., from control and program) analysis. In comparison, MAYDAY performs cross-domain trace-back to RAV accident root causes by revealing the causality between physical impacts and control program bugs.

Program-Level Root Cause Analysis Many root cause analysis techniques based on execution logs have been proposed to investigate program failures [49,64,76,77], security incidents [47,54,60] and for debugging [27,56,63].

Several solutions leverage program instrumentation to generate execution logs [63,77]. On the other hand, there is a large number of works that record OS events during runtime and perform offline analyses to backtrack the provenance of Advanced Persistent Threat (APT) attacks [47,54,60]. These works leverage program execution partitioning [54,60] and system event dependency models [47,54,60] to identify attack paths accurately in a large amount of log data from long-running systems. Another line of work records complete or partial execution until a program crashes and analyzes the logs to diagnose the root causes or reproduce the errors [64,77]. Some of these works [49,76] leverage hardware assistance [2] to log fine-grain program execution with high efficiency. Guided by RAV control model and control "model-to-program" mapping, MAYDAY achieves higher accuracy and efficiency for control program debugging.

Some debugging techniques such as statistical debugging techniques [27,56] work by comparing the statistical code coverage patterns in "passing" and "failing" runs. However, bugs in control systems do not always induce obvious code coverage difference, due to the iterative control-loop execution model, in which the same set of components (e.g., sensor reading sampling and control output generation) is periodically executed, with or without a controller digression. As such, for our target systems, they may not be as effective as for non-control programs.

Runtime Assurance and Testing for Robotic Vehicle Safety There have been significant advances in ensuring robotic vehicle operation reliability and safety to monitor controller state digression [28,34], violation of safety constraints [75] and memory safety [50]. Meanwhile, there have been many software testing efforts that aim at bug detection [23,40,51,65,74]. Several techniques are proposed to find erroneous behaviors of deep neural networks [65] and violation of safety constraints [23] for autonomous cars. Timperley et al. [74] and RVFuzzer [51] introduced new testing methods to characterize existing bugs and find control-semantic vulnerabilities in robotic vehicles, respectively. Compared to these runtime defense and off-line testing techniques, MAYDAY focuses on post-accident trace-back of control-semantic bugs, based on off-line source code instrumentation, runtime logging, and post-accident log analysis. He et al. [40] proposed a debugging system based on heuristics and an approximate model generated by a system identification technique. Unlike MAYDAY, which is designed for post-accident investigation based on *production* runtime logs, the debugging system

in [40] is effective only in scenarios where one can interactively monitor multiple program runs hence is more applicable during program development.

11 Conclusion

It is challenging to investigate RAV accidents caused by control-semantic bugs. We have presented MAYDAY, a cross-domain RAV accident investigation tool that localizes program-level root causes of accidents, based on RAV control model and enhanced in-flight logs. Guided by a generic RAV control model (CVDG), MAYDAY selectively instruments the control program to record its execution aligned with existing control-level logs. Using the control- and program-level logs, MAYDAY infers and maps the culprit control variable corruption from control domain to program domain, and localizes the bug within a very small fragment of the control program. Our investigation of 10 accident cases caused by real control-semantic bugs demonstrates the effectiveness of MAYDAY. Moreover, MAYDAY incurs low runtime and storage overhead.

Acknowledgment

We thank our shepherd, Nathan Dautenhahn, and the anonymous reviewers for their valuable comments and suggestions. This work was supported in part by ONR under Grants N00014-17-1-2045 and N00014-20-1-2128. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the ONR.

References

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

[2] Processor tracing, 2013. https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing.

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

[4] When drones fall from the sky, 2014. https://www.washingtonpost.com/sf/investigative/2014/06/20/when-drones-fall-from-the-sky.

[5] F.A.A. Opens Inquiry After Baby Hurt in Drone Crash, 2015. https://www.nytimes.com/2015/09/23/business/drone-crash-injures-baby-highlighting-faa-concerns.html.

[6] White House Drone Crash Described as a U.S. Worker's Drunken Lark, 2015. https://www.nytimes.com/2015/01/28/us/white-house-drone.html.

[7] Facebook drone investigation: Wind gust led to broken wing, 2016. https://www.cnet.com/news/facebook-drone-investigation-wind-gust-led-to-broken-wing.

[8] *Amazon Prime Air*, 2017. https://www.amazon.com/Amazon-Prime-Air/b?node=8037720011.

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

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

[11] *Zipline's Ambitious Medical Drone Delivery in Africa*, 2017. https://www.technologyreview.com/s/608034/blood-from-the-sky-ziplines-ambitious-medical-drone-delivery-in-africa.

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

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

[14] *Navio2*, 2018. https://emlid.com/navio.

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

[16] *PX4 Pro Open Source Autopilot*, 2018. http://px4.io.

[17] *Raspberry Pi 3 Model B*, 2018. https://www.raspberrypi.org/products/raspberry-pi-3-model-b.

[18] *TPM Main Specification*, 2018.

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

[20] *ArduPilot :: About*, 2020. https://ardupilot.org/about.

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

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

[23] Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C Briand, and Thomas Stifter. Testing autonomous cars for feature interaction failures using many-objective search. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018.

[24] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 1996.

[25] Steve Best. Journaling file systems. *Linux Magazine*, 4:24–31, 2002.

[26] 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)*, 2017.

[27] Trishul M. Chilimbi, Ben Liblit, Krishna Mehra, Aditya V. Nori, and Kapil Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, 2009.

[28] 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 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[29] Devon R Clark, Christopher Meffert, Ibrahim Baggili, and Frank Breitinger. Drop (drone open source parser) your drone: Forensic analysis of the dji phantom iii. *Digital Investigation*, 22:S3–S14, 2017.

[30] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[31] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. Hafix: Hardware-assisted flow integrity extension. In *Proceedings of the 52nd Annual Design Automation Conference (DAC)*, 2015.

[32] Drew Davidson, Hao Wu, Rob Jellinek, Vikas Singh, and Thomas Ristenpart. Controlling UAVs with Sensor Input Spoofing Attacks. In *Proceedings of the 10th USENIX Workshop on Offensive Technologies (WOOT)*, 2016.

[33] Matthew Eagon, Zhan Tu, Fan Fei, Dongyan Xu, and Xinyan Deng. Sensitivity-based dynamic control frequency scheduling of quadcopter mavs. In *Proc. SPIE 11009, Autonomous Systems: Sensors, Processing, and Security for Vehicles and Infrastructure*, 2019.

[34] 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)*, 2018.

[35] Aurélien Francillon, Quan Nguyen, Kasper B Rasmussen, and Gene Tsudik. A minimalist approach to remote attestation. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2014.

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

[37] 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.

[38] Saeid Habibi. The smooth variable structure filter. In *Proceedings of the IEEE*, volume 95, pages 1026–1059, 2007.

[39] Taylor Hardin, Ryan Scott, Patrick Proctor, Josiah Hester, Jacob Sorber, and David Kotz. Application memory isolation on ultra-low-power mcus. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, 2018.

[40] Zhijian He, Yao Chen, Enyan Huang, Qixin Wang, Yu Pei, and Haidong Yuan. A system identification based oracle for control-cps software fault localization. In *Proceedings of the IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2019.

[41] Liang Heng, Daniel B Work, and Grace Xingxin Gao. Gps signal authentication from cooperative peers. *IEEE Transactions on Intelligent Transportation Systems*, 16(4):1794–1805, 2014.

[42] Kate Highnam, Kevin Angstadt, Kevin Leach, Westley Weimer, Aaron Paulos, and Patrick Hurley. An Uncrewed Aerial Vehicle Attack Scenario and Trustworthy Repair Architecture. In *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, 2016.

[43] Owen S Hofmann, Sangman Kim, Alan M Dunn, Michael Z Lee, and Emmett Witchel. Inktag: Secure applications on an untrusted operating system. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[44] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.

[45] Upasita Jain, Marcus Rogers, and Eric T Matson. Drone forensic framework: Sensor and data identification and verification. In *Proceedings of Sensors Applications Symposium (SAS)*, 2017.

[46] Kai Jansen, Matthias Schäfer, Daniel Moser, Vincent Lenders, Christina Pöpper, and Jens Schmitt. Crowd-gps-sec: Leveraging crowdsourcing to detect and localize gps spoofing attacks. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (SP)*, 2018.

[47] Yang Ji, Sangho Lee, Mattia Fazzini, Joey Allen, Evan Downing, Taesoo Kim, Alessandro Orso, and Wenke Lee. Enabling refinable cross-host attack investigation with efficient data flow tagging and tracking. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[48] Simon J Julier and Jeffrey K Uhlmann. Unscented filtering and nonlinear estimation. *Proceedings of the IEEE*, 92(3):401–422, 2004.

[49] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, 2015.

[50] 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.

[51] Taegyu Kim, Chung Hwan Kim, Junghwan Rhee, Fan Fei, Zhan Tu, Gregory Walkup, Xiangyu Zhang, Xinyan Deng, and Dongyan Xu. Rvfuzzer: Finding input validation bugs in robotic vehicles through control-guided testing. In *Proceedings of 28th USENIX Security Symposium (USENIX Security)*, 2019.

[52] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, 2014.

[53] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. LogGC: Garbage Collecting Audit Log. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer Communications Security (CCS)*, 2013.

[54] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High Accuracy Attack Provenance via Binary-based Execution Partition. In *Proceedings of the 20th Annual Symposium on Network and Distributed System Security (NDSS)*, 2016.

[55] Yanlin Li, Jonathan M. McCune, and Adrian Perrig. Viper: Verifying the integrity of peripherals' firmware. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, 2011.

[56] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[57] Xin Liu, Yuhua Xu, Luliang Jia, Qihui Wu, and Alagan Anpalagan. Anti-jamming communications using spectrum waterfall: A deep reinforcement learning approach. *IEEE Communications Letters*, 22(5):998–1001, 2018.

[58] Zhilong Liu, Raja Sengupta, and Alex Kurzhanskiy. A power consumption model for multi-rotor small unmanned aircraft systems. In *Proceedings of the 2017 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2017.

[59] Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Marshall Kirk Mckusick. Ffsck: The fast file-system checker. *ACM Transactions on Storage (TOS)*, 10(1), 2014.

[60] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting. In *Proceedings of the 25th Annual Symposium on Network and Distributed System Security (NDSS)*, 2016.

[61] Dennis K Nilsson, Lei Sun, and Tatsuo Nakajima. A framework for self-verification of firmware updates over the air in vehicle ecus. In *Proceedings of the 2008 IEEE GLOBECOM Workshops*, 2008.

[62] K Ogata and Y Yang. Modern control engineering. 1970.

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

[64] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. Pres: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, 2009.

[65] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.

[66] Konstantinos Pelechrinis, Ioannis Broustis, Srikanth V Krishnamurthy, and Christos Gkantsidis. Ares: an anti-jamming reinforcement system for 802.11 networks. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2009.

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

[68] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Proceedings of International Workshop on Recent Advances in Intrusion Detection (RAID)*, 2008.

[69] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP)*, 2007.

[70] Yunmok Son, Hocheol Shin, Dongkwan Kim, Young-Seok Park, Juhwan Noh, Kibum Choi, Jungwoo Choi, and Yongdae Kim. Rocking Drones with Intentional Sound Noise on Gyroscopic Sensors. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*, 2015.

[71] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. Hdfi: hardware-assisted data-flow isolation. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (SP)*, 2016.

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

[73] Vinaitheerthan Sundaram, Patrick Eugster, and Xiangyu Zhang. Efficient diagnostic tracing for wireless sensor networks. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2010.

[74] Christopher Steven Timperley, Afsoon Afzal, Deborah S Katz, Jam Marcos Hernandez, and Claire Le Goues. Crashing simulated planes is cheap: Can simulation detect robotics bugs early? In *Proceedings of the 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, 2018.

[75] Michael Vierhauser, Jane Cleland-Huang, Sean Bayley, Thomas Krismayer, Rick Rabiser, and Pau Grünbacher. Monitoring cps at runtime-a case study in the uav domain. In *Proceedings of 2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2018.

[76] Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. Postmortem program analysis with hardware-enhanced post-crash artifacts. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, 2017.

[77] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: Error diagnosis by connecting clues from run-time logs. In *Proceedings of the 15th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

Appendix

A State Consistency Check Formula

During the CVDG-level corruption path inference (Section 6.1.2), MAYDAY must determine whether the vehicle state of the initial digressing controller (x_I) and that of the child controller (x_c) shows consistent vehicle states. We leverage the following equation to compare the two vehicle states:

$$err(C_I, C_c) = \frac{\int_t^{t+w_I} |x_I(s+w_c) - x_I(s) - \int_s^{s+w_c} \lambda(x_c(v))dv|ds}{w_I} \quad (1)$$

Intuitively, x_I and x_c are not directly comparable since their orders are different. To match the order, MAYDAY makes the order of x_c equal to that of x_I via integral using the time window (w_c). MAYDAY leverages IAE [37] to robustly check the error value between the two vehicle states within the time window (w_I), similar to the initial digression determination (Section 6.1.1). If this error value is larger than the threshold (Thr) of the child controller, we consider the two vehicle states as inconsistent. The above time windows (w_I, w_c) and threshold (Thr) are described in Appendix B.

Additionally, we leverage λ which is a conversion function to compare different child controllers. λ is normally an identity function. However, λ becomes the inverse form of an inter-cascading controller formula in Fig. 6 only if the controller states are located in multiple cascading controllers (Section 5.1).

B Parameters for Digression Determination

We used threshold (Thr) and time window (w) (refer to the IAE formula [37] and Equation 1) for both initial digression determination 6.1.1 and state consistency check in Appendix A. For the selection of reasonable Thr and w, we used the three-sigma rule [67] with fifty different experimental missions for 6DoF, similar to the previous work [51]. Compared to w in the previous work, we used much smaller windows to detect the more accurate time when the initial digression occurred. Specifically, we used 0.5 seconds for the x-, y-axis controllers, z-axis position, acceleration controllers, and yaw and yaw rate controllers. In addition, we used 0.25 seconds for the z-axis velocity controller, and roll and roll rate controllers, and pitch and pitch rate controllers.