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

Taegyu Kim, Purdue University; Chung Hwan Kim, University of Texas at Dallas; Altay Ozen, Fan Fei, Zhan Tu, Xiangyu Zhang, Xinyan Deng, Dave (Jing) Tian, and Dongyan Xu, Purdue University

https://www.usenix.org/conference/usenixsecurity20/presentation/kim

This paper is included in the Proceedings of the 29th USENIX Security Symposium.
August 12-14, 2020
978-1-939133-17-5

Open access to the Proceedings of the 29th USENIX Security Symposium is sponsored by USENIX.
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 model in the control program. The two categories involve hardware or software issues separately, while the third category entangles both the cyber and physical aspects of an RAV 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-semantic bugs, which arise from program-
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,

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

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

We structured the rest of this paper as follows: Section 2 illustrates background on RAV control model and program 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_c(t) is the vehicle state (i.e., position). r_x(t) is the reference which indicates the desired position.
We assume that the logs are fully recoverable from the vehicle’s “black box”. (4) We assume the control flow integrity of control programs ArduPilot [12], PX4 [16] and Paparazzi [15]. The RA V control program already generates high-level 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 RA V control model. (2) We assume that the RA V 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 RA V 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 RA V 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 RA V 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 RA V control model is also testified to by its wide adoption by RA V 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 RA V 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/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 RA V 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 RA V 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 RA V 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.
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).
5.1 Control Variable Dependency Graph

**Mapping CVDG to Control Program**

**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 bitcode (i.e., 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.

**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 bytecode 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.

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>Initialise M</code> [O] Our algorithm entry point</td>
</tr>
<tr>
<td>2</td>
<td>for <code>cv</code> ∈ <code>CV</code> do Backward slicing for each <code>CV</code></td>
</tr>
<tr>
<td>3</td>
<td><code>PV</code> ← <code>POINTS-TOANALYSIS</code>(cv)</td>
</tr>
<tr>
<td>4</td>
<td><code>S</code> ← <code>BACKWARDSLICINGVARSET</code>(PV) Backward slicing for aliases of <code>cv</code></td>
</tr>
<tr>
<td>5</td>
<td><code>N</code> ← <code>GETAFFECTINGNODES</code>(S) Get CVDG nodes connected to <code>cv</code></td>
</tr>
<tr>
<td>6</td>
<td>for <code>n</code> ∈ <code>N</code> do Get a CVDG edge connecting between <code>cv</code> and another CVDG node</td>
</tr>
<tr>
<td>7</td>
<td><code>M[e]</code> ← <code>GETINSTSFOREDGE</code>(e, S) Mapping instructions to each edge</td>
</tr>
<tr>
<td>8</td>
<td><code>return M</code></td>
</tr>
<tr>
<td>9</td>
<td><code>function BACKWARDSLICINGVARSET</code>(SV) [O] This function is called recursively</td>
</tr>
<tr>
<td>10</td>
<td><code>V</code> ← <code>SV</code></td>
</tr>
<tr>
<td>11</td>
<td><code>S</code> ← ∅ Backward slicing set for the given variable set</td>
</tr>
<tr>
<td>12</td>
<td>for <code>v</code> ∈ <code>SV</code> do</td>
</tr>
<tr>
<td>13</td>
<td><code>S</code> ← <code>BACKWARDSLICINGONEVAR</code>(v)</td>
</tr>
<tr>
<td>14</td>
<td><code>s</code> = <code>S</code> ∪ <code>S′</code> Add new slicing results for each <code>v</code></td>
</tr>
<tr>
<td>15</td>
<td><code>V′</code> ← <code>GETAFFECTINGVARS</code>(S′) – V Get newly found variables</td>
</tr>
<tr>
<td>16</td>
<td><code>V</code> ← <code>V′</code> ∪ <code>V′′</code> Recursive slicing</td>
</tr>
<tr>
<td>17</td>
<td>for <code>v′</code> ∈ <code>V′</code> do Perform recursive slicing on new variables</td>
</tr>
<tr>
<td>18</td>
<td><code>PV</code> ← <code>POINTS-TOANALYSIS</code>(v′)</td>
</tr>
<tr>
<td>19</td>
<td><code>S′</code> ← <code>BACKWARDSLICINGVARSET</code>(PV) Recursive slicing</td>
</tr>
<tr>
<td>20</td>
<td><code>S</code> ← <code>S</code> ∪ <code>S′</code> Add new slicing results for each <code>v′</code></td>
</tr>
<tr>
<td>21</td>
<td><code>return S</code></td>
</tr>
</tbody>
</table>

Algorithm 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 digressing-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 $C_I$’s effort to track the corrupted $r_c$, it generated the corrupted reference for its own child controller, and so on and 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 $x_I \rightarrow r_I$, or $r_I \rightarrow r_c$ as described in Figure 7c.

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

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

If the initial digression is between $r_I$ and mission input $M$

\[1\] 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 \( r_1 \) and the new \( r_1 \) value made \( C_f \) malfunction. In the CVDG, the corruption of \( r_1 \) happened on path \( M \rightarrow r_1 \), as described in Figure 7d. Similar to Types I-III, we can prove that the parent controller of \( C_f \) cannot initially corrupt \( r_1 \).

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>
<thead>
<tr>
<th>Type</th>
<th>Initial Digressing Variables</th>
<th>( x_1 ) and ( x_2 ) Consistent?</th>
<th>Initially Corrupted Variable</th>
<th>CVDG-Level Corruption Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Between ( r_2 ) and ( x_2 )</td>
<td>No ( x_2 )</td>
<td>( S \rightarrow x_2 \rightarrow r_2 )</td>
<td></td>
</tr>
<tr>
<td>II</td>
<td>Between ( r_2 ) and ( x_2 )</td>
<td>Yes ( x_2 )</td>
<td>( k_2 \rightarrow k_2 \rightarrow r_2 )</td>
<td></td>
</tr>
<tr>
<td>III</td>
<td>Between ( r_2 ) and ( x_2 )</td>
<td>Yes ( x_2 )</td>
<td>( t_2 \rightarrow t_2 \rightarrow r_2 )</td>
<td></td>
</tr>
<tr>
<td>IV</td>
<td>Between ( M ) and ( r_2 )</td>
<td>Yes ( t_2 )</td>
<td>( M \rightarrow r_2 )</td>
<td></td>
</tr>
</tbody>
</table>

### 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, MARYDAY 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

MARYDAY 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 bytecode-level data flow analysis.

#### 6.2.2 CVDG-Guided Program Trace Analysis

MARYDAY 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 bytecode-level program trace, indexed by control loop iteration number; (2) the initial digressing iteration number \( i_{digiress} \); (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.

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 \( P \rightarrow k_8 \rightarrow \bar{r}_4 \). The initial digressing time is Iteration 4930. \( P, k_8, \) and \( \bar{r}_4 \) 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 \( i_{digiress} \) 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 \( P_{\bar{r}_4} \rightarrow k_8 \rightarrow V_4, 4850 \rightarrow V_8, 4929 \rightarrow \bar{r}_4, 4930 \), which realizes CVDG-level path \( P \rightarrow k_8 \rightarrow \bar{r}_4 \). 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 \( i_{digiress} \)) 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 \( i_{digiress} \) (Line 3-5, 7-21). In Fig. 8, such an earlier data flow is \( P_{\bar{r}_4} \rightarrow k_8 \rightarrow V_4, 4850 \rightarrow V_8, 4851 \rightarrow k_8, 4852 \). We point out that, different from traditional program analysis, MARYDAY needs to capture the influence on the corrupted control variable \( (\bar{r}_4) \) in multiple control loop iterations towards (and including) \( i_{digiress} \). 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
We evaluate M’s effectiveness with respect to RAV accident investigation (Section 8.1) and bug localization (Section 8.2); and M’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.

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]

3https://github.com/ArduPilot/ardupilot/issues/12121
Table 2: List of accident cases caused by control-semantic bugs.

<table>
<thead>
<tr>
<th>Case ID</th>
<th>Category</th>
<th>Initial Digressing Controller</th>
<th>Initial Corruption</th>
<th>CVDG-Level Corruption Path</th>
<th>Program-Level Investigation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Controller Parameter Corruption</td>
<td>Command &amp; turn</td>
<td>No range check of kp parameter for x-y axis velocity controller</td>
<td>P → kp → e_xy</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Extreme vehicle instability or crash</td>
<td>Command &amp; turn</td>
<td>No range check of kp parameter for x-y axis velocity controller</td>
<td>P → kp → e_xy</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Extreme vehicle instability or crash</td>
<td>Command &amp; turn</td>
<td>No range check of kp parameter for roll angular controller</td>
<td>P → kp → e_xy</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>Extreme vehicle instability or crash</td>
<td>Command &amp; turn</td>
<td>No range check of kp parameter for pitch angular controller</td>
<td>P → kp → e_xy</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>Crash</td>
<td>Command &amp; speed change</td>
<td>Wrong variable name leading to out-of-range x-y axis velocity</td>
<td>e80328d</td>
<td>No</td>
</tr>
<tr>
<td>6</td>
<td>Flight Mission Corruption</td>
<td>Command</td>
<td>Wrong way-point computation based on non-existent coordinate</td>
<td>9739659</td>
<td>No</td>
</tr>
<tr>
<td>7</td>
<td>Crash</td>
<td>Command</td>
<td>Invalid type-casting of z-axis location causing an integer overflow</td>
<td>756256d</td>
<td>No</td>
</tr>
<tr>
<td>8</td>
<td>Crash</td>
<td>Command</td>
<td>Missing divided-by-zero check of kp parameter for z-axis position controller</td>
<td>c2a290b</td>
<td>No</td>
</tr>
<tr>
<td>9</td>
<td>Crash</td>
<td>Command</td>
<td>Missing divided-by-zero check of kp parameter for x, y-axis position controllers</td>
<td>c03a506</td>
<td>No</td>
</tr>
<tr>
<td>10</td>
<td>Crash</td>
<td>Weak or no wind</td>
<td>Missing divided-by-zero check in angular calculation</td>
<td>29a40d</td>
<td>No</td>
</tr>
</tbody>
</table>

Case 1-7, that duration can be arbitrarily long. More specifically, the initial corruption of a control variable on the CVDG-level path may happen first in just a few iterations (e.g., 8 in Case 1). But the controller’s initial digression could happen an arbitrary number of iterations later, depending on the timing of the vehicle’s operation that “sets off” the digression (e.g., a turn or a change of altitude). Such “low-and-slow” nature of accidents makes it harder to connect their symptoms to causes and highlights the usefulness of MAYDAY.

**Program-Level Investigation:** The 5th and 6th columns of Table 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.

For Cases 8-10, accurate detection of divide-by-zero bugs

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 X: bug not detected.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Missing controller parameter range check</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>Δ</td>
</tr>
<tr>
<td>2</td>
<td>Missing controller parameter range check</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>Δ</td>
</tr>
<tr>
<td>3</td>
<td>Missing controller parameter range check</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>Δ</td>
</tr>
<tr>
<td>4</td>
<td>Missing controller parameter range check</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>Δ</td>
</tr>
<tr>
<td>5</td>
<td>Comparison with a wrong variable</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>Δ</td>
</tr>
<tr>
<td>6</td>
<td>Wrong waypoint computation based on non-existent coordinate</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>7</td>
<td>Integer overflow on a mission variable</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>Δ</td>
</tr>
<tr>
<td>8</td>
<td>Divide-by-zero caused by invalid controller parameter</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>Δ</td>
</tr>
<tr>
<td>9</td>
<td>Divide-by-zero caused by invalid controller parameter</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>Δ</td>
</tr>
<tr>
<td>10</td>
<td>(Probabilistic) Divide-by-zero caused by sensor input</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

* 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 mutations 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.
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. 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., \( \dot{x}_x \) always tracked \( \dot{r}_x \)), 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 \( P \rightarrow \dot{k}_x \rightarrow \dot{r}_x \) (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 \( k_{xy} \)) 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, \texttt{set_and_save} saves the new parameter value. The value is later retrieved by \texttt{get_p}, when \texttt{rate_to_accel_xy} is called by the x, y-axis velocity controller. The code indicates that the controller

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.
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 velocity mission 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 15 cm/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 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 miti-gates the long-existing problem of state explosion [53, 54] faced by generic program attack provenance.
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.

<table>
<thead>
<tr>
<th></th>
<th>Average Latency / Iteration (µs)</th>
<th>Breakdown (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log generation</td>
<td>37.22</td>
<td>19.71</td>
</tr>
<tr>
<td>Log I/O</td>
<td>153.5</td>
<td>81.29</td>
</tr>
<tr>
<td>Logging total</td>
<td>190.72</td>
<td>100</td>
</tr>
</tbody>
</table>

Storage Overhead We measure MAYDAY’s log data genera-
tion 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] exclud-
ing 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 con-
sumes 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 af-
ter instrumentation, log integrity, and log recover-ability in
MAYDAY. To achieve code integrity, we can apply content-

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.

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., SecVi-
ator [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 de-
ployed with ArduPilot for memory access protection, even on
low-end micro-controllers with only an MPU (memory pro-
tection 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 theoreti-
cal model. As acknowledged in Section 2, there exist other
types of vulnerabilities in RAV systems, such as traditional
program vulnerabilities and vulnerabilities in physical com-
ponents (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-
execution trace analysis would not be necessary for detecting or investigating physical attacks.

Fortunately, defenses against physical attacks exist and can be deployed alongside with MAYDAY. Many sensor attacks can be detected by checking the RAV control log [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 MB) 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 event system 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


### B Parameters for Digression Determination

We used threshold ($T hr$) 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 $T hr$ and $w$, we used the three-sigma rule [67] with fifty different experimental mis-sions 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.

### 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:

\[
err(C_I, C_C) = \int_{t_s}^{t_e} x_I(s+w_C) - x_C(s) - \int_{t_s}^{t_e} \lambda(x_C(v)) dv ds
\] (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 ($T hr$) of the child controller, we consider the two vehicle states as inconsistent. The above time windows ($w_I$, $w_C$) and threshold ($T hr$) are described in Appendix B.

Additionally, we leverage $\lambda$, which is a conversion function to compare different child controllers. $\lambda$ is normally an identity function. However, $\lambda$ 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).