LTA: Control-Driven UAV Testing and Bug Localization with Flight Record Decomposition

Changyul Lee[†]
The Pennsylvania State University
cpl5774@psu.edu

Deokjin Kim[†¶]
The Affiliated Institute of ETRI
djkim@nsr.re.kr

Giyeol Kim
The Pennsylvania State University
gpk5233@psu.edu

Sangwook Lee
The Affiliated Institute of ETRI
leesw@nsr.re.kr

Taegyu Kim
The Pennsylvania State University
tgkim@psu.edu

ABSTRACT

As UAVs have been widely used in various domains, such as the military and industry, their safety and security have become crucial. One of their root causes is software bugs, which fall into two bug categories: traditional software bugs, such as memory safety bugs, and UAV-specific logical model-misimplementation (LMM) bugs leading to physical misbehavior, such as crashes. To discover and localize bugs, many proactive and reactive techniques have been proposed. However, LMM bug mitigation techniques are still immature, unlike well-established techniques for traditional software bugs, because existing approaches are unable to track the causal relationship between the LMM bug root cause in software and its resulting physical misbehavior. Specifically, existing proactive approaches require extensive, time-consuming dynamic testing to capture the physical impacts of LMM bug exploitation amidst a vast input space. Conversely, previous reactive approaches are inaccurate because existing work cannot accurately identify the causal relationship between misbehavior and bug-triggering inputs mixed with benign but suspicious inputs.

To address the aforementioned problems, we propose LTA, the replay-based proactive LMM bug localization technique for UAVs. This technique encompasses three key strategies: (i) an accident playback-based input generation to narrow down bug-triggering input candidates, (ii) an input and trace decomposition to exclude false-positive bug-triggering inputs, and (iii) a causal analysis to precisely backtrack from bug-triggering inputs to their root causes. We evaluate LTA on PX4 with three models for quadcopters, hexacopters, and VTOL UAVs. As a result, LTA found 72 real accident cases (caused by LMM bugs) obtained from public accident logs and then localized bugs with 100% accuracy.

CCS CONCEPTS

• **Computer systems organization → Robotics**; • **Security and privacy → Software security engineering**; • **Software and its engineering → Software testing and debugging**.

SenSys '24, November 4–7, 2024, Hangzhou, China

ACM ISBN 979-8-4007-0697-4/24/11…$15.00
https://doi.org/10.1145/3666025.3699350

KEYWORDS

Unmanned aerial vehicle, Testing, Bug localization, Control model

ACM Reference Format:
Changyul Lee, Deokjin Kim, Giyeol Kim, Sangwook Lee, and Taegyu Kim. 2024. LTA: Control-Driven UAV Testing and Bug Localization with Flight Record Decomposition. In *ACM Conference on Embedded Networked Sensor Systems (SenSys '24), November 4–7, 2024, Hangzhou, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3666025.3699350

1 INTRODUCTION

UAVs have been used in various domains, such as delivery [21] and military [5]. The heart of UAVs is its software. It executes the given tasks (e.g., reaching a waypoint) and stabilizes flight movements by properly controlling motors in response to sensor values.

Unfortunately, UAV accidents can cause critical safety and security issues, damaging critical infrastructures and human lives. There are three types of accident causes, as follows: (1) physical causes, such as mechanical failures and environmental factors (e.g., wind gusts), (2) traditional software bugs, such as memory corruption bugs, and (3) logical model-misimplementation (LMM) bugs arising from logically misimplemented UAV control model code (e.g., incorrect or missing range check). Exploiting an LMM bug leads to flight misbehavior [36, 40, 43], such as a crash and flight route deviation, which may lead to accidents. For notation simplicity, we will refer to the ending results of flight misbehavior as an accident. We will provide its detailed and systematic definition in Section 3.1. Instead of the first two types of bugs, this paper focuses on the third type of bug which has paid less attention to than other bug types.

Let us take one example accident. During the UAV flight, a UAV operator sent two pitch angular rate control parameter update commands due to a change in the UAV's weight (e.g., loading an extra item). Although this operator changed only those two parameters, the UAV z-axis position (whose *controller was not under attack*) decreased gradually and then crashed into the ground. After that, UAV investigators tried to identify the accident root causes by analyzing the flight log. The accident root causes are buggy code accessing the two changed pitch angular rate control parameters without checking the validity of their values updated by inputs. However, it is hard to identify the causal relationship between all the input commands (including the ones triggering those two bugs) and flawed descent control only with a one-time UAV accident log.

[†]Equal contribution. Order was decided by coin flip.
[¶]Work done during a scholarly visit at the Pennsylvania State University.

Furthermore, it is hard to distinguish which inputs (one of the two-parameter update inputs, both parameter update inputs, and other benign-looking inputs) cause the accident and localize the relevant buggy code. LTA found this motivating case (detailed in Section 3) by analyzing a flight log in the public flight record repository [7].

To eliminate bugs, bug discovery and localization are prerequisites. However, if we target LMM bugs, most existing bug discovery and localization techniques lack the following cross-domain analysis capabilities: (1) identifying inputs causing misbehavior, (2) backtracking from an accident to a misbehaving controller which causes this misbehavior, (3) transforming the misbehaving controller into corresponding program components, and (4) localizing LMM bugs and bug-triggering inputs.

There have been proactive LMM bug discovery and reactive LMM bug localization techniques. However, existing proactive discovery techniques [36, 39, 40, 43] cannot localize LMM bugs since they focus only on bug-triggering input identification. Moreover, the majority of their generated tests make those techniques spend most testing time executing cases irrelevant to bug activation. This problem is worse due to the large input space caused by a large number of input variables and their inter-dependencies (e.g., a safe parameter value can be unsafe if its dependent parameter is changed). Finally, each test takes non-negligible time because a test requires reflecting physical laws in accordance with real-time progression. On the other hand, existing reactive bug localization techniques [42] are not applicable until an accident caused by an LMM bug occurs. Moreover, they may misidentify bug root causes, especially in scenarios where a mix of benign and bug-triggering inputs is fed. This challenge arises from their reliance on limited information from a single log recorded during an accident log.

In this paper, we propose LTA, a replay-based proactive LMM bug localization framework by **L**earning **T**est cases from **A**ccident records (LTA). We address previously identified challenges as follows: First, LTA analyzes previous public flight records and then generates inputs that likely trigger LMM bugs. This approach significantly improves testing efficiency by filtering out testing inputs unrelated to LMM bugs. Next, LTA distinguishes bug-triggering inputs and then decomposes an accident log into multiple logs, each of which corresponds to its respective bug-triggering input(s). This approach isolates the effects of other bug-triggering and benign inputs, thereby enhancing the accuracy of the subsequent bug localization phase. Moreover, LTA identifies causal relationships between accident root causes and their resultant misbehavior. This is achieved by (1) mapping both control model components and their corresponding control program code into our control-model-to-control-program (or model-program) mapping graph and (2) backtracking from misbehavior to the root causes through analysis of decomposed input and trace pairs. Finally, LTA effectively localizes vulnerable code by finding an intersection and subset of code implementing a misbehaving controller and dynamically executed code. Thereby, LTA markedly narrows down the scope of the code in comparison with analyses only on programs.

We evaluated LTA on PX4 [19], one of the most popular open-source UAV programs. PX4 is widely adopted in commercial UAVs with various UAV models [14–16]. LTA covers the three major models, quadcopter, hexacopter, and VTOL. LTA has found a total of 72 accident cases caused by LMM bugs from the public flight record repository [7] and identified their root causes with 100% accuracy. On average, LTA distinguishes 1.31 bug-triggering inputs from 10.61 inputs and localized bugs within 37.1 basic blocks.

2 BACKGROUND

UAV Control System. A UAV control system comprises three core control components: sensors, motors, and a control program. They interplay with each other to stabilize physical movements. As defined in a control model, sensors (e.g., an inertial measurement unit (IMU)) measure physical movements, such as a z-axis velocity. Then, a control program takes measurement results and then controls motors to generate proper propulsive forces. This procedure is repeatedly executed to stabilize flight movements. We denote the above procedure as a control loop and each execution of this procedure as a control loop iteration.

To stabilize flight movements at every control loop iteration, a control program manages four control state types: measured states, reference states, control parameters, and missions (the details of the two latter states will be introduced later). Regarding the first two states, measured states are physical movements measured by sensors. Reference states are target physical movements at every control loop iteration. A control program determines those reference states to stabilize physical movements, considering measured states. Then, it converts reference states into the motor control outputs to make measured states closer to reference states at every control loop iteration. This way, over the multiple control loop executions, a UAV can stabilize physical movements.

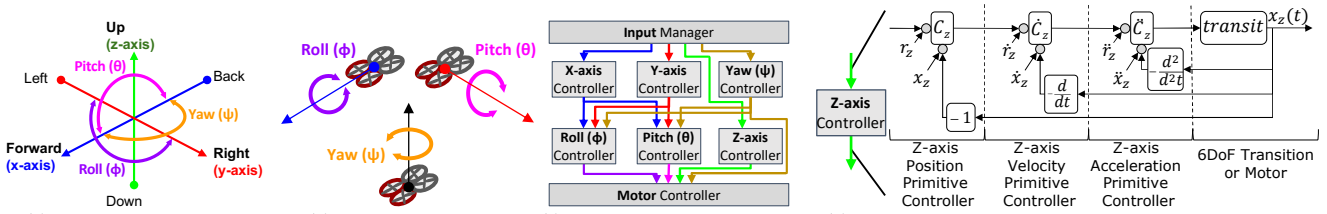
Moreover, a control program has two management components: a (remote) operator interface and flight log recorders. An operator interface, such as ground control station (GCS) software, allows operators to send commands to control UAVs. These commands support updating control parameters (e.g., control gains) and missions (e.g., target moving destination or velocity). Specifically for control parameters, they define how controllers react against changes in measured states to generate appropriate reference states. As such, they are typically used for troubleshooting purposes. Furthermore, an operator interface regularly receives control states from a control program. This allows operators to monitor UAV control states during flight. Meanwhile, modern UAVs are equipped with flight log recorders. They record control states in a UAV's local storage.

UAV Control Model. The UAV control model is designed to properly control movements along the six degrees of freedom (6DoF). Figure 1a shows 6DoF, including x, y, and z-axes, and three rotations in Figure 1b, roll, pitch, and yaw, along the three axes. In accordance with the control model, six *cascaded controllers* controls the corresponding 6DoF in Figure 1c. Each cascaded controller consists of three *primitive controllers* to manage position, velocity, and acceleration of each 6DoF, as shown in Figure 1d.

$$x(t) = P \cdot e(t) + I \cdot \int_0^t e(x)dx + D \cdot \frac{de(t)}{dt} + FF \cdot r(t) \quad (1)$$

$$e(t) = r(t) - x(t) \quad (2)$$

Let us explain how those controllers work internally. We will take the *z-axis cascaded controller*, consisting of three *primitive controllers* managing z-axis position, velocity, and acceleration, respectively, as an example. Each primitive controller, such as the proportional-integral-derivative (PID) controller, is designed based on a controller algorithm and maintains its measured state and reference state. This



(a) Six degrees of freedom. (b) Roll, pitch, and yaw. (c) Cascaded controllers and their dependencies. (d) Primitive controllers in the z-axis cascaded controller.

Figure 1: UAV's control model and its controller structure.

PID controller is widely used in real-world UAVs [18–20]. We show Equations 1 and 2 to illustrate the mathematical operations of the PID controller. In these equations, variables of control states, x(t), r(t), and e(t), respectively represents a *measured state*, *reference state*, and *error state*. For our denotation convenience, we call variables of control states as control variables. The e(t) is the difference between x(t) and r(t), and a PID controller stabilizes control states by minimizing e(t) to make x(t) closer to r(t) over multiple control loops. P, I, D, and FF are the control parameters, another type of control state variables, that define how the controller responds to measured states as defined in Equation 1.

We represent one cascaded controller using both nodes (i.e., gray circles) and controller blocks in Figure 1d. Nodes represent variables of measured states and reference states, while blocks represent instances of a controller algorithm including its control parameters. For our denotation convenience, we omitted (t) for each control state. On the other hand, each controller block (e.g., C_z, $\dot{C}_z$, and $\ddot{C}_z$) consists of a primitive controller formula (e.g., Equation 1) and parameters and takes nodes as inputs to compute values of control variables. Each cascaded controller has three controller blocks. Those controller blocks correspond to the z-axis position, velocity, and acceleration primitive controllers, respectively. These controller receive reference states (r_z, $\dot{r}_z$, and $\ddot{r}_z$) and measured states (x_z, $\dot{x}_z$, and $\ddot{x}_z$), computes error states (e_z, $\dot{e}_z$, and $\ddot{e}_z$), and then generate outputs (o_z, $\dot{o}_z$, and $\ddot{o}_z$).

The output of a primitive controller flows to its dependent controller (e.g., from a z-axis position primitive controller to a z-axis velocity primitive controller). We refer to such flows as *control variable dependencies* for primitive controllers. The final output flows to either another 6DoF cascaded controller or motors. Figure 1c shows how the final output flows into either another cascaded controller or motors, as shown by the edges in Figure 1c. They are *control variable dependencies* for cascaded controllers.

Flight Records. Public flight records [7] contain a huge number of flight logs uploaded by worldwide anonymous UAV users. Those logs record not only the aforementioned control states but also user inputs to control UAVs. LTA leverages them to discover LMM bugs.

3 SCOPE, MOTIVATION, ASSUMPTION

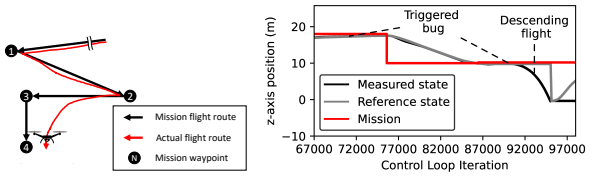
3.1 Our Safety Model and Scope

Safety Model. LTA focuses on localizing LMM bugs which corrupt control variables, leading to persistent misbehavior — not temporary misbehavior that can be mitigated thanks to modern and robust controller design. LMM bugs can be triggered mistakenly or maliciously via the UAV operator interface [12]. For example, an immature user can inadvertently send a mission or control parameter update command that sets a control variable as an invalid value. Next, attackers can exploit LMM bugs [39, 42, 45, 55] by identifying bug-triggering inputs and conditions through the analysis of (public) flight record [7]. Note that UAV model changes (e.g., a change in frame shape and motor power) can invalidate a previously valid parameter value. This can happen in various situations, such as a delivery item loading or (discontinued) mechanical component replacement due to its unavailability. In such scenarios, an LMM bug, coupled with a model change, can also be exploited. Moreover, we assume that UAVs record every primitive controller's control states and inputs updating missions and parameters in flight logs as discussed in Section 2. Due to legal restrictions, mature UAVs, such as PX4 [19], ArduPilot [18], and Paparazzi [20], record such logging data necessary for LTA. Finally, we assume that all the logs were recorded at the time of the accident and earlier than this time. Those data are essential for LTA's analysis.

Target Bug Scope. LTA aims at identifying LMM bugs and their bug-triggering inputs. LMM bugs can remotely manipulate any of a mission and a control parameter. This type of bug is beneficial to (malicious) operators or attackers in practice because they can be intentionally or mistakenly triggered via a remote operational interface with a single or few inputs. There are two types of accident conditions: *reference state divergence* and *measured state divergence*.

Reference state divergence is the case where a UAV cannot execute the given mission for the given timeout. This type of divergence involves the persistent difference between a mission and reference state without being decreased. This type of bug can appear when a controller has buggy logic processing a mission input. This may cause a malfunctioning controller to prevent from setting a proper reference state for a mission input given by a UAV operator. As a result, a UAV cannot execute missions, which leads to various implications according to the flight context, such as a flight toward an incorrect location and zero flight speed without moving anywhere.

Measured state divergence is the case where a measured state (e.g., z-axis measured velocity) cannot track its relevant reference state (e.g., z-axis reference velocity). This divergence involves the large and persistent difference between measured and reference states. Such failure in tracking a reference state appears because a malfunctioning controller keeps setting a reference state as an invalid value without considering its corresponding measured state. This divergence may cause controllers to fail to stabilize flight control, leading to oscillating flights. Its implication varies up to environmental factors, such as a crash into the ground and a collision with an obstacle that is not on the planned flight route.



(a) Z-axis flight trajectory. (b) Flight mission, measured and reference states over time.

Figure 2: Example misbehaving flight record.

Meanwhile, we do not consider accident cases caused by non-software issues, such as mechanical failure (e.g., motor failure), battery failure, sensor failure and attacks, and environmental factors (e.g., wind gusts). Instead, existing flight-record forensic work [11, 27, 37] can identify those accident root causes. Furthermore, traditional bugs (e.g., memory corruption bugs) are out of scope because many existing techniques [31, 38, 50, 56, 66] can address them. *Note that our target bug-triggering inputs must be recorded right before they are processed, although traditional bugs potentially (but practically do not) may bypass this input recording logic.*

UAV Model Support. LTA prototype supports UAV models, such as quadcopters, hexacopters, helicopters, VTOLs, etc, because their UAV models align with the 6DoF control model in Figure 1. However, LTA can be extended to support the other non-6DoF-based models. We discuss its extension details in Section 7.

3.2 Why LMM bug Discovery and Localization Are Challenging?

Triggering an LMM bug can cause physical misbehavior, leading to UAV accidents. To prevent such accidents, LMM bug discovery and localization are essential preliminary steps for bug elimination but require suitable analyses across the control and program domains. To enable those analyses, there are four challenges. First, we need to selectively test the cases that likely trigger accidents. Otherwise, there is a large testing input space, leading to wasting much time to test. Next, our technique should be able to detect misbehaving controller components. This detection requires a method to monitor all the primitive controller operations (introduced in Section 2) and controller-specific misbehavior detection rules. Third, we need to fill the semantic gap between physical misbehavior in the control domain and bugs in the program domain. Otherwise, we cannot identify the relationships between misbehavior and a triggered bug across the two different domains (i.e., control and program domains). Fourth, we must be able to backtrack the causal relationship from resulting misbehavior to a root cause (i.e., an LMM bug in UAV software and its bug-triggering input) through the filled domain gap. This process needs a new cross-domain backtracking analysis.

3.3 Research Problems

There have been several works to discover LMM bugs. In general, there are two bug discovery approaches: proactive and reactive bug discovery. Proactive bug discovery means we discover bugs in advance before bugs are exploited (e.g., fuzzing [39, 40, 43]). On the other hand, reactive bug discovery means we discover bugs after bugs are exploited to prevent further exploitation of identical bugs, such as (accident) root cause analyses [26, 36, 42]. However, those works propose limited solutions for the aforementioned research problems in Section 3.1. As a result, they still have limited bug discovery and localization efficiency and accuracy. There are three research problems to address to improve efficiency and accuracy.

Problem 1: Limited Efficiency of LMM bug Discovery. We have observed that both proactive and reactive approaches have limited efficiency. Proactive approaches demand non-trivial time for every testing run to observe and evaluate bug exploitation implications under the faithful physical laws with a physical simulator [44]. Therefore, this approach requires enhancing input generation, creating inputs likely to trigger bugs. Existing proactive bug discovery techniques employed the input mutation technique guided by UAV specifications [3, 4, 6] or control properties. However, the majority of their generated tests are not pertinent to bug activation but consume most of the testing time. This is because the input space is significant due to a large number of input variables and their inter-dependencies (e.g., a safe parameter value can be unsafe if its dependent parameter is changed). Moreover, each test requires non-negligible time to reflect physical laws in accordance with real-time progression. Conversely, reactive approaches cannot detect any bug until an accident caused by an LMM bug occurs.

Problem 2: Inaccurate Misbehavior Identification. Identifying which controller misbehaves and its misbehavior type must be accurate. Inaccuracies in this process can lead to incorrect LMM bug localization. In the above motivating example case where a pitch rate controller misbehaved, existing bug localization techniques [36, 42] misidentify that the z-axis position controller shows misbehavior, leading to incorrect bug localization. This inaccuracy arises from the reliance on heuristically determined control metrics during misbehaving controller detection although these metrics may be effective for detecting misbehavior's presence [39, 40, 43].

Problem 3: Inaccurate Causal Analysis Between Multiple Inputs and Their Effects. During the flight, a mix of benign and bug-triggering inputs can be fed. While some bug-triggering inputs can cause misbehavior individually, some inputs can do that only in combination with their dependent inputs. In the latter case, existing techniques struggle to discern which inputs or pairs of inputs cause misbehavior. Specifically, proactive bug discovery techniques [36, 39, 40, 43] focus on detecting misbehavior occurrences without identifying accident root causes. Furthermore, reactive bug discovery techniques [42] cannot do that. This is because an accident can produce only one flight log recording the cumulative effects of all the benign and bug-triggering inputs without separating the effects of individual inputs. As a result, they cannot distinguish which input(s) leads to the negative effect(s) (i.e., misbehavior) only with such a one-time accident flight log. Consequently, they introduce false positives (i.e., a benign input is misidentified as a root cause) and false negatives (i.e., a bug-triggering input is not identified as a root cause), leading to identifying incorrect inputs and localizing vulnerable code as root causes.

3.4 Motivation Accident Example

Let us take one accident case uploaded by an anonymous user on the public repository [7] and assume that a quadcopter UAV tried to deliver an item under an operator's control. During delivery, its

operator needed to increase the pitch rate control gains because UAV was slightly heavier than usual due to a delivery item. To reduce UAV crash risks, this operator checked the parameter specification [6] and then increased the two pitch rate parameters falling within "safe" ranges. Specifically, she set MC_PITCHRATE_P (its safe range is from 0.01 to 0.6) as 0.45 and MC_PITCHRATE_K (its safe range is from 0.01 to 5.0) as 4.0. After a few seconds, the quadcopter descended from 17.53m into the ground before she noticed because the pitch rate controller buggy code was exploited unintentionally. To identify bugs, we reproduced this case and observed its flight trajectory in the z-axis as shown in Figures 2a and 2b. Note that this accident can happen only with both parameter modifications — not with only one of the parameter modifications. To analyze this accident case, LTA is necessary to selectively choose to analyze such a case, backtrack from physical misbehavior to its root cause across the different domains, and accurately localize multiple bug-triggering inputs and relevant LMM bugs. In fact, LTA found both bug-triggering inputs and localized bugs within 25 basic blocks (i.e., 44 lines of source code (LoC)).

3.5 Usage and Assumption

Motivational Usage. LTA can localize LMM bugs with 37.1 basic blocks and 73.39 LoC on average while avoiding testing inputs that are unlikely to trigger bugs. LTA is beneficial to the two types of users (but not limited to these types): (1) UAV vendors and (2) UAV operational institutes (e.g., institutes for UAV delivery service and military). Both user types must ensure the safety and security of their own UAVs that will be either sold or managed internally.

Why LTA is Proactive? Considering the motivational usage, one effective way is to inspect the existing flight records in the public or internally maintained flight record repositories. The use of those repositories is promising to *proactively* discover LMM bugs because the quantity of public flight records keeps growing thanks to worldwide UAV users. Those records may include accident records that are new to our target users and give *new promising accident analysis data* to discover LMM bugs instead of keeping waiting for new accidents of their own UAVs for new LMM bug discovery. Furthermore, UAV vendors or management companies could share their flight record repositories that are new to individual vendors or institutes to further proactively discover bugs. Finally, we can additionally obtain promising accident records by checking flight records even with different physical models (e.g., wing size or weight is different). This is because inputs triggering those accidents can likely cause similar accidents even with a different UAV model. Note that this approach is different from reactive approaches because they need to wait for accident appearance to discover even a single LMM bug.

Other Usage Scenarios. We can use LTA in combination with their input mutation algorithms of existing UAV fuzzers [39, 40, 43] while overcoming Problems 2 and 3 in Section 3.3. This helps discover and localize zero-day LMM bugs. Furthermore, UAV vendors or management companies that use LTA share their internally maintained flight record repository and analyze shared flight records.

User Knowledge Assumption. We assume that LTA users have control domain knowledge. This is required to report and inspect which specific control code has an LMM bug and patch it.

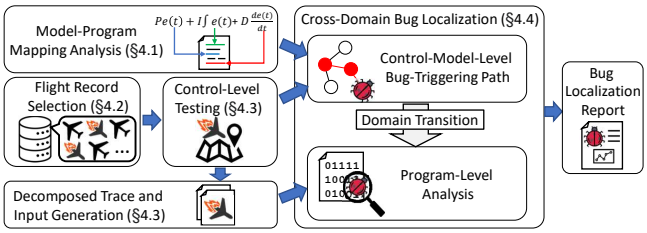


Figure 3: The overview of LTA.

4 DESIGN

We propose LTA to proactively discover and localize LMM bugs from UAV flight records, guided by the UAV control model and theory. Figure 3 shows the overview of LTA. LTA takes the following inputs: (1) an LLVM bitcode of a UAV control program source code, (2) flight records from (public) repositories, and (3) a model-program mapping graph. Note that we describe how to generate (3) a model-program mapping graph in Section 4.1. Given the above inputs, LTA takes the following steps.

Model-Program Mapping Analysis (Section 4.1): LTA generates the model-program mapping graph mapping both the UAV control model and its control program to this graph.

Flight Record Selection (Section 4.2): To address **Problem 1** in Section 3.3, LTA obtains flight records and shortlists records containing accidents caused by LMM bugs. Such shortlisting involves control-theory-based analysis of flight operations.

Control-Level Testing and Decomposed Input and Trace Generation (Section 4.3): LTA conducts tests to identify which inputs can cause accidents from the chosen flight records. To tackle **Problem 3** in Section 3.3, LTA repeatedly tests a simulated UAV program with a physical simulator [44]. Then, LTA performs our control-level analysis to distinguish which inputs trigger LMM bugs leading to misbehavior. Finally, LTA generates pairs of decomposed program-level traces, their corresponding bug-triggering input(s), and their control-level analysis results.

Cross-Domain Bug Localization (Section 4.4): Our cross-domain analysis localizes LMM bugs to address **Problems 2 and 3** in Section 3.3 by analyzing the given decomposed program-level traces with our mapping graph. Specifically, LTA first identifies control-model-level bug-triggering paths, using bug-triggering input(s) and the control-level analysis results. Next, LTA transforms these paths into the corresponding program-level components, using our mapping graph, and then generates the program-level bug-triggering paths. Finally, LTA localizes bugs within program-level paths.

4.1 Model-Program Mapping Analysis

We will introduce how to build the mapping graph based on the UAV control model and then explain how to map a UAV control program (in the LLVM bitcode form) into the model. Note that our mapping graph is generically applicable to various UAV models discussed in Section 3.1, without adapting it into a specific model.

Mapping Graph. LTA builds the mapping graph in Figure 4 by augmenting the generic UAV control model (Figure 1), inspired by existing work [42]. Our graph includes not only the existing control model components but also operator input paths that can trigger bugs. Note that Figure 4 shows only the z-axis cascaded controller

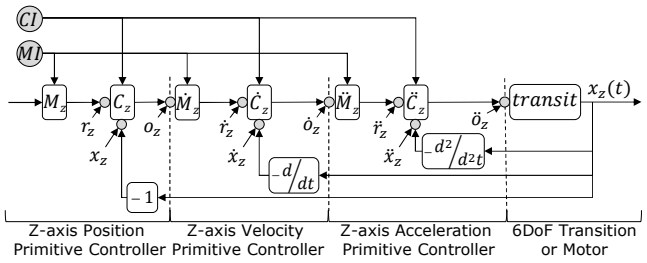


Figure 4: Mapping graph.

as a part of the control model. The complete graph includes the remaining five cascaded controllers and their dependencies, as shown in Figure 1c.

Let us explain the mapping graph with the z-axis cascaded controller example. Apart from existing control model components (control variables of a measured state (e.g., $\dot{x}_z$), a reference state (e.g., $\dot{r}_z$), a controller block (e.g., $\dot{C}_z$), and its control parameters), we added additional four component types. The first and second component types are a *control parameter update input (*CI*)* and a *mission update input (*MI*)*. Those inputs can trigger LMM bugs. CI updates a control parameter in a controller block. Then, controller blocks (C_z, $\dot{C}_z$, and $\ddot{C}_z$) respectively generate the output control variables (o_z, $\dot{o}_z$, and $\ddot{o}_z$). Those output control variables are the third new component type. On the other hand, MI updates a mission (e.g., destination or mission flight speed designation) in the mission block, the last new component type. In the z-axis cascaded controller, there are three mission blocks (M, $\dot{M}_z$, and $\ddot{M}_z$). Those blocks respectively take output control variables (o_z, $\dot{o}_z$, and $\ddot{o}_z$) from the preceding controller block and MI as inputs. Each mission block sets missions (i.e., another type of control variable), adjusts its inputs (e.g., o_z from the preceding controller block), and then sends the reference state (e.g., $\dot{r}_z$), the adjusted output control variable, to the following controller block (e.g., $\dot{C}_z$).

Program Component Mapping. LTA locates "program-level" control variables and control variable dependencies corresponding to those of the mapping graph. To locate control variables for CI and MI, their names appear in their specifications for popular major UAVs [3, 4, 6, 39]. By locating variables whose names are matched with names in the specifications, LTA can locate those control variables. To locate control variables for control states, LTA performs regular expressions using their common names as keywords. For example, a z-axis velocity reference state variable names include 'z', 'vel', and 'ref' or 'setpoint'. This way, LTA found those control variables with 100% accuracy for major open-source UAVs [18–20]. If LTA could not find any of them, we can manually inspect logging functions recording control variables. Note that mature UAVs are equipped with logging functions due to legal regulations, and their recorded control variables have names with easily recognizable common control variable keywords. Then, to locate their alias variables without missing them, LTA uses the points-to analysis [58].

Next, LTA identifies "program-level" control variable dependencies by performing inter-procedural backward slicing. Starting from the aliases of control variables, LTA statically locates control code by tracking all the possible program-level control variable dependencies defined in our mapping graph. Let us take CI $\rightarrow \dot{C}_z \rightarrow \dot{o}_z$ (a path in Figure 4) as one example. Then, LTA backtracks from $\dot{o}_z$ to CI. The resulting code on program-level control variable dependencies must include (1) one instruction reading CI (the starting control variable of the path), (2) another instruction writing $\dot{o}_z$ (the ending control variable of the path), and (3) the instructions found by backward slicing between (1) and (2).

During the program-level dependency tracking, LTA encounters global and heap variables. It can create extra dependencies because such a variable can be accessed by one writing instruction and then reading instruction in a different call context. In this case, LTA adds those two instructions' dependency in a different call context and then resumes tracking code on its dependency.

After finishing the above backtracking procedure, LTA reversely tracks the identified dependencies "in a forward manner". This additional step discards redundant paths that are not part of the paths in the mapping graph.

ROS-Aware Dependency Tracking. A UAV program is also a type of robot program. Many robot developers adopt ROS [1] to improve robot program development productivity by modularizing control program parts (called ROS nodes) and providing efficient inter-module communication APIs (called ROS communication APIs). However, because such APIs introduce additional means to exchange data, LTA needs to track this new type of control variable dependencies. Specifically, `publish` and `subscribe` are ROS functions that are responsible for sending and receiving data, respectively. Those functions' calls take a `topic` instance as arguments. LTA identifies which `subscribe` function call receives data from a `publish` function call by tracking whether they use the aliases of a `topic` instance. LTA identifies those `topic` instance aliases by using the points-to analysis [58]. This enables covering ROS-design-based UAV programs, unlike existing works [42].

4.2 Flight Record Selection

We will explain how LTA selects flight records containing the inputs that likely trigger LMM bugs. LTA first analyzes (public) flight records to gather the following information: (i) reference and measured states of all primitive controllers and (ii) mission and control parameter update inputs. Then, LTA detects whether a UAV showed misbehavior in each flight record. There are two misbehavior types: *measured state divergence* and *reference state divergence.*

Measured state divergence is the case where the large and persistent difference appears between measured and reference states. This divergence happens if a measured state (e.g., z-axis measured velocity) cannot track its relevant reference state (e.g., z-axis reference velocity). To determine this type of divergence occurs, LTA uses the control-theory-based metric, the integral absolute error (IAE) [33] ($\int_t^{t+w} \frac{|r(T)-x(T)|}{w} dT$). Given a time window w since t, LTA computes IAE. If IAE is larger than a threshold (thres) for measured and reference states of any primitive controller during w, LTA considers an accident happens. On the other hand, *reference state divergence* is the case where the UAV cannot execute the given mission for the given timeout. To detect that, LTA checks whether the difference persistently appears between a mission and reference state without being decreased.

Note that we will show our w and thres and timeout values and how to determine them in Section 5. Furthermore, LTA chooses flight records only involving persistent divergence by setting w

and timeout to sufficiently large values. This approach ensures temporary divergences, such as those caused by a sharp turn, are not misclassified as misbehavior because modern robust controllers can quickly recover from such temporary divergences.

Once LTA finds all the records involving misbehavior, it checks whether the accidents in records are likely caused by LMM bugs. For that, LTA inspects whether control parameter or mission update commands in the records exist. If so, LTA considers this accident may be caused by LMM bugs and performs further analyses.

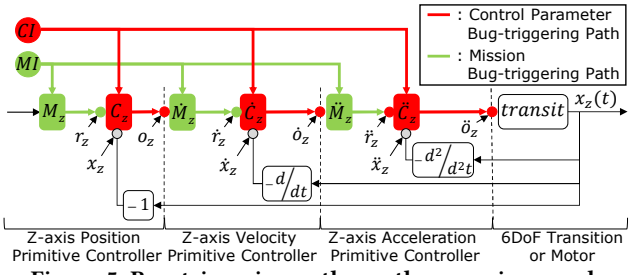


Figure 5: Bug-triggering paths on the mapping graph.

4.3 Control-Level Testing and Decomposed Input and Trace Generation

We first introduce the testing setup, how to identify bug-triggering inputs (more precisely, input sequences as introduced later), misbehavior type, and initial misbehavior time, and how to generate decomposed program execute traces.

Testing Setup. First, LTA obtains input sequences of flight mission and control parameter update commands from flight records. Next, LTA customizes a UAV operator interface to mutate mission and control parameter update input sequences and send them to a simulated UAV program. Simultaneously, UAV program exchanges sensor values (i.e., measured states) and propulsive forces generated by motors with a physical simulator [44]. Meanwhile, a UAV program sends measured and reference states and inputs to LTA to analyze them to detect misbehavior.

Bug-Triggering Input Sequence Mutation. With the above testing setup, LTA finds minimal a bug-triggering input sequence triggering LMM bugs, inspired by one existing work [26]. When LTA tests each flight record, LTA retrieves its UAV model (e.g., a quadcopter and a plane) and flight mission and control parameter update command inputs and their feeding time. In this part, we focus on describing input mutation. We will explain misbehavior detection later.

For each flight record analysis, LTA mutates mission and control parameter update commands as inputs based on the following algorithm. The initial testing run begins with all gathered inputs. If no accident occurs, LTA concludes that this accident is caused by a patched bug or other factors such as a mechanical fault and environmental factor (e.g., wind gust). Then, LTA tests a UAV to first find bug-triggering mission update input sequences. LTA keeps mission update inputs and their feeding times while keeping control parameters default and runs tests. If "reference state divergence" occurs during the test, the mission executed after a mission update input fed is a bug-triggering input. Then, LTA removes this input from the mission update input sequence. LTA repeatedly tests with the modified input sequence to find additional bug-triggering inputs until no bug is triggered. Mission input update sequences without triggering bugs are benign.

Next, LTA tests a UAV program to find bug-triggering control parameter update inputs with the above "benign" mission update input sequence. This involves mutating control parameter update input sequences and testing them. LTA sets one of those parameter update inputs while keeping other parameters default. If misbehavior occurs, LTA considers that input as a "single bug-triggering input". Meanwhile, some input can cause an accident only if its dependent bug-triggering input(s) are fed sequentially. We call such inputs a "bug-triggering input sequence". LTA identifies such input sequences via incrementally including/excluding inputs and repeatedly tests with different input sequence combinations. If a certain input sequence causes any misbehavior, LTA considers that a bug-triggering input sequence without testing its inputs.

Misbehavior Type and Time Identification. During each testing run, LTA identifies misbehavior types. For that, LTA analyzes control states with the IAE formula as described in Section 4.2. Meanwhile, LTA additionally identifies *initial misbehavior time* (i.e., start time to show persistent misbehavior) and an initial misbehavior type (i.e., either measured state divergence or reference state divergence). *Plus, LTA records the bug-triggering input feeding time.*

Trace Decomposition Per Inputs. During the test runs, LTA selectively records program-level traces per bug-triggering input(s). The reason why it stores individual traces per input(s) is to separately analyze such "decomposed" traces and inputs without being affected by other inputs. Furthermore, LTA uses Intel PT [9] to record program-level traces and their timestamps for the next step.

4.4 Cross-Domain Bug Localization

In this step, LTA localizes LMM bugs. LTA first takes decomposed bug-triggering input sequences, execution traces, initial misbehavior types and times, and input feeding time (Section 4.3) as inputs. Then, LTA takes the following three steps: (1) control-model-level bug-triggering path identification, (2) transformation from those paths into the corresponding program-level components, and (3) program-level bug localization by analyzing execution traces starting from the initial misbehavior time and bug-triggering input feeding times. Through those steps, LTA localizes LMM bugs within a small number of basic blocks. As discussed in the second last paragraph of Section 1, we can achieve this goal by identifying the intersection of the basic blocks only on control-model-level bug-triggering paths and execution traces.

Control-Model-Level Bug-Triggering Path Identification. LTA identifies control-model-level bug-triggering paths ($\mathcal{CP}$s) according to the bug-triggering input sequence and initial misbehavior pattern. As defined in our mapping graph, a $\mathcal{CP}$ consists of (1) a misbehaving controller, (2) control variables ($\mathcal{CV}$s), and (3) control variable dependencies.

LTA identifies each $\mathcal{CP}$ as follows. First, LTA identifies a misbehaving controller having control variable dependencies with a bug-triggering input rather than a controller showing initial misbehavior. This is because misbehavior detection relies on threshold values (thres in Section 4.2). This sometimes causes our analysis to misidentify another controller as a misbehaving controller. For

example, the varying wind speed makes a roll rate controller misbehave faster than a roll controller because the derivative term (e.g., a roll rate controller) is more sensitive to show misbehavior earlier.

Next, LTA identifies (2) and (3). Let us assume the identified misbehaving controller is a z-axis velocity primitive controller. There are two types of $\mathcal{CP}$s according to the two misbehavior patterns (Section 4.2). If measured state divergence happens, the *red* path in Figure 5 is the $\mathcal{CP}$. This path spans from CI through CI's dependent control variables in $\dot{C}_z$ (e.g., a z-axis control gain parameter) to $\dot{o}_z$. This output is corrupted after the velocity controller's computation due to the corrupted control parameter. Otherwise, the *green* path in Figure 5 is the $\mathcal{CP}$. This example path spans from MI through MI's dependent control variables in $\dot{M}_z$ (e.g., a variable for a z-axis velocity controller) to $\dot{r}_z$. This path corrupts the reference state of this controller when it processes MI.

The above steps identify each $\mathcal{CP}$ and are repeated until all the $\mathcal{CP}$s, unlike existing work [42] assuming a single root cause exists.

Domain Transition. This step is responsible for converting "control-model-level" components into "program-level" components. LTA has $\mathcal{CP}$s and their initial misbehavior and input feeding times (right before this step) and program-level traces (from Section 4.3). Given those inputs, LTA (1) transforms $\mathcal{CV}$s into the corresponding control variables in the program ($\mathcal{PV}$s) and (2) synchronizes the time between program-level traces and control-level times. Note that we will explain transforming control variable dependencies (the other part of $\mathcal{CP}$s) in **Program-Level Analysis**.

$\mathcal{CV}$s can be readily converted into $\mathcal{PV}$s, thanks to our mapping graph. On the other hand, for (2), LTA synchronizes the time between program-level traces and control-level times by using timestamps. Specifically, program-level traces have timestamps (i.e., branch instruction execution time), and control-level analysis results have input feeding and initial misbehavior times. LTA utilizes these timestamps to align the times recorded in program-level traces with control-level analysis results.

Program-Level Analysis. Following $\mathcal{CP}$s, LTA localizes LMM bugs by backtracking the "program-level" bug-triggering paths ($\mathcal{PP}$s) from initial misbehavior to the $\mathcal{PV}$ of bug-triggering input(s). A $\mathcal{PP}$ consists of $\mathcal{PV}$s and their executed paths following control variable dependencies in our mapping graph. Note that LTA identifies if paths are executed by analyzing execution traces.

For that, LTA performs our interprocedural backward slicing on UAV program execute traces to find all the reachable paths between $\mathcal{PV}$s of the entire $\mathcal{PP}$s. Algorithm 1 shows this procedure in detail. Specifically, LTA backtracks from the ending $\mathcal{PV}$ to the $\mathcal{PV}$ connected by one subpath (or edge in the mapping graph) and repeats it until the starting $\mathcal{PV}$ is reached (Lines 2-3 and 6-30).

Let us take the pitch rate controller example in Section 3. Considering a $\mathcal{CP}$, the entire $\mathcal{PP}$ is CI → MC_PITCHRATE_P → $\dot{o}_{pitch}$. Note that the bug-triggering input targets MC_PITCHRATE_P which is part of $\dot{C}_{pitch}$. Its subpaths of this $\mathcal{PP}$ should be CI → MC_PITCHRATE_P and MC_PITCHRATE_P → $\dot{o}_{pitch}$. In this example, LTA starts to backtrack the last subpath spanning from the ending $\mathcal{PV}$ (i.e., $\dot{o}_{pitch}$) to MC_PITCHRATE_P (Lines 9-10 and 15-30). In this case, the ending $\mathcal{PV}$ ($\dot{o}_{pitch}$) should be updated by the value read from MC_PITCHRATE_P. Hence, LTA backtracks the instruction updating $\dot{o}_{pitch}$ to the instruction reading MC_PITCHRATE_P, using the interprocedural backward slicing.

Algorithm 1 Program-level bug-triggering path identification.

Input: Execution trace (T), Control-model-level bug-triggering path (p_c), Control loop iteration at an initial misbehaving time (i_m), Control loop iteration at input feeding time (i_{in}).
Output: Identified program-level bug-triggering path set.

```
1:  function PathIdentification(p_c, i_in, i_m, T)                    ▷ Main function
2:    BP_p ← ∅                          ▷ Initialize backtracked program-level path set
3:    SP_p ← ∅                          ▷ Initialize backtracked program-level subpath set
4:    BP_p ← PathBT(p_c, BP_p, SP_p, i_in, i_m, T)                    ▷ BT: Backtrack
5:    return PathForwardVerify(p_c, BP_p)
6:  function PathBT(p_c, BP_p, SP_p, i_begin, i_end, T)               ▷ BT: Backtrack
7:    if Size(p_c) ≤ 0 then                          ▷ No more subpaths to be tracked
8:      return SP_p
9:    sp_c ← PopLastSubPath(p_c)                    ▷ sp_c: control-model-level subpath
10:   NewSP_p ← SubPathBT(sp_c, i_begin, i_end, T)                    ▷ Backtrack a sp_c
11:   SP_p ← LinkSubPaths(NewSP_p, SP_p)
12:   for sp_p ∈ SP_p do                          ▷ Track every untracked subpath
13:     BP_p ← BP_p ∪ PathBT(p_c, BP_p, SP_p, i_begin, sp_p.i_begin, T)
14:   return BP_p
15: function SubPathBT(sp_c, i_begin, i_end, T)                       ▷ BT: Backtrack
16:   SP_p ← ∅                          ▷ PSP_p: partial program-level subpath set
17:   PSP_p ← BSSubPathSet(sp_c, {sp_c.cpv_end}, i_end, T)            ▷ BS: backward slicing
18:   for i ∈ {i_end...i_begin} do                ▷ Backtrack subpaths at every iteration
19:     PrevPSP_p ← PSP_p
20:     while Size(PrevPSP_p) > 0 do
21:       psp_p ← Pop(PSP_p)
22:       if isCompleteSubPath(psp_p) == True then
23:         SP_p ← SP_p ∪ {psp_p}
24:         PSP_p ← PSP_p − {psp_p}
25:       else                                          ▷ Track sub-paths
26:         BSP_p ← BSSubPathSet(sp_c, RetReadCPVSet(psp_p), i, T)
27:         NewPSP_p ← LinkPathInterIter(BSP_p, psp_p)                ▷ Link
    subpaths if there are linkable subpaths at this iteration and the analyzed
    later iteration
28:         if Size(NewPSP_p) > 0 then
29:           PSP_p ← PSP_p − {psp_p} ∪ NewPSP_p
30:   return SP_p
31: function PathForwardVerify(p_c, BP_p)
32:   P_p ← ∅
33:   for p_p ∈ BP_p do
34:     if PathForwardConsistencyCheck(p_p, p_c) == True then
35:       P_p ← P_p ∪ {p_p}
36:   return P_p
```

Then, LTA repeats it until it backtracks the entire $\mathcal{PP}$ (Lines 12-13). Note that LTA considers (sub)paths (i.e., sliced instructions) only following the dependencies of the given $\mathcal{CP}$. Otherwise, the other (sub)paths are discarded. Furthermore, LTA may find multiple $\mathcal{PP}$s because they can exist even for one $\mathcal{CP}$. This is because one $\mathcal{PV}$ (e.g., MC_PITCHRATE_P) can affect multiple program variables which are part of subpaths our backward slicing detects.

Meanwhile, LTA may encounter the case where some subpaths between $\mathcal{PV}$s pass through global and heap variables. LTA considers such variables as intermediate $\mathcal{PV}$s. Similar to the example in Section 4.1, let us assume that there is one *writing* instruction on one intermediate $\mathcal{PV}$. Then, one *reading* instruction in a different call context reads that intermediate $\mathcal{PV}$ later. In this case, LTA tracks subpaths through such intermediate $\mathcal{PV}$s by following the dependency between those reading and writing instructions.

However, the $\mathcal{PP}$ mentioned earlier is incomplete due to a lack of temporal information. To add this missing information, LTA needs to track subpaths across the different control loop iterations because

these subpaths may exist in earlier control loop iterations than the one under analysis. To handle this, when the backtracking analysis encounters the beginning of a trace in a control loop iteration (i.e., there is no more trace to track in the nth iteration), LTA repeats the backtracking from the last program-level trace in the earlier iteration (e.g., n−1th iteration) (Lines 18-29). After that, LTA creates a complete $\mathcal{PP}$ by annotating the control loop iteration numbers in $\mathcal{PP}$, $\text{CI}_{70,999} \rightarrow \text{MC\_PITCHRATE\_P}_{71,000} \rightarrow \dot{\text{o}}_{\text{pitch},93,008}$. If any $\mathcal{PP}$ reaches $\text{CI}_{70,999}$ (i.e., code processing a bug-triggering input), LTA stops backtracking (Lines 7-8).

Once the backtracking analysis is done, LTA performs forward tracking analysis to check whether the identified $\mathcal{PP}$s are reachable to the ending $\mathcal{PV}$ (e.g., $\dot{\text{o}}_{\text{pitch},93,008}$) (Lines 5 and 31-36). $\mathcal{PP}$s must be trackable in both forward and backward manners. Otherwise, because they do not actually follow the control variable dependencies defined in our mapping graph, LTA discards $\mathcal{PP}$s or subpaths that are not trackable by any of forward and backward tracking (Lines 33-35). Then, LTA extracts a set of basic blocks executed along the found $\mathcal{PP}$s with the bug-triggering input. Those basic blocks must include bugs. LTA repeats the above procedures for every input of the bug-triggering input sequences and its relevant $\mathcal{CP}$.

Technical Benefits of Our Approach. Note that LTA effectively mitigates dependency (or path) explosion problems [32] because LTA (1) identifies only $\mathcal{PP}$s that strictly follow $\mathcal{CP}$s and (2) considers only $\mathcal{PP}$s that were executed. This approach prevents LTA from tracking extraneous dependencies (or paths), which could exacerbate the dependency explosion problem. Meanwhile, existing techniques' "program-level-only analyses" lack our "control model oracles". Without them, they cannot recognize the starting points (i.e., bug-triggering inputs) and ending points (i.e., control variables of a misbehaving controller) of $\mathcal{PP}$. As a result, such existing techniques track dependencies irrelevant to misbehavior.

On the other hand, LTA does not miss LMM bugs because our resulting $\mathcal{PP}$s contain LMM bugs. Specifically, the identified resulting $\mathcal{PP}$s are a subset of $\mathcal{CP}$s. Those $\mathcal{CP}$s cover all the possible $\mathcal{PP}$s containing LMM bugs.

5 IMPLEMENTATION

Our LTA prototype targets PX4 v1.12.3 [19] because the PX4 community manages the public flight record repository [7]. We use QGroundControl v4.2.3 [13] as operator interface software which enables controlling UAV software and monitoring its control states using MAVLink protocol [12], which is the de-facto standard UAV protocol. Finally, we use Gazebo v11.10.2 [44], the default and high-fidelity physical simulator of PX4, to replay accidents.

We implemented our program analysis (Sections 4.1 and 4.4) on top of LLVM. We compiled PX4 into an LLVM bitcode using gLLVM v1.3.1 [17] and LLVM v13.0 [46]. Then, we used SVF v2.5 to perform the points-to analysis [58] and modified it to support our mapping analysis. We used Intel PT [9] and libipt v2.0.5 [8] to record and decode program-level traces. In addition, we implemented the execution trace translator, based on libdwarf v0.4 [10], to figure out which instructions in UAV's LLVM bitcode are executed, using DWARF debugging symbols. To implement LTA, we wrote 16,770 lines of C++ code and 7,636 lines of Python code.

To detect misbehavior, we set thresholds, windows, and timeouts used in Sections 4.2 and 4.3. Specifically, we set thresholds as the top 1.5% largest difference values between measured states and reference states of each primitive controller. We set different threshold values according to the models as shown in Table 1. Furthermore, we heuristically set the windows (w) as five seconds and the timeout (to detect stuck movements in the sky or flight route deviation) as 60 seconds. In our experiments, those values do not cause false positives or negatives.

6 EVALUATION

We first show our experimental setup in Section 6.1. Next, we show the effectiveness and efficiency of LTA by showing the step-by-step UAV localization results in Sections 6.2 and 6.3. Finally, we show one case study to prove the practicality of LTA in Section 6.4.

6.1 Experimental Setup

We evaluate LTA with PX4 software-in-the-loop (SITL) with Gazebo [44]. Note that SITL contains control code identical to that of physical PX4 firmware, and Gazebo, one of the most widely used physical simulators in the control research and industrial institutes, is a high-fidelity physical simulator. We run them on Ubuntu 22.04 64bit with the Intel i5-12450H 2.3GHz processor and 48 GB memory. Our target UAV software runs its tasks and records control states (including initial misbehavior and input feeding times) at a fixed rate (e.g., the highest frequency for PX4 is 250Hz). Using such regular execution frequency as a time unit (i.e., 4ms), LTA synchronizes task execution time with the relevant control loop iteration.

We gathered the flight records from the public repository [7] uploaded on the dates between 11/01/2017 and 04/03/2023. The number of the entire UAV flight records during that period is 28,215. We show the detailed flight record statistics in Table 2. Out of them, we chose the 25,117 flight records containing all the flights of the three major models: a quadcopter (having 23,432 flight records), a hexacopter (having 1,216 flight records), and a VTOL (having 469 flight records). We chose those three models because those three models' records occupy 89.02% of the entire fight records due to their relatively high control performance [23, 57].

Those three models are PX4's default models. Note that the actual models can be different from those models (e.g., model customization). However, LMM bugs for one model can likely be exploitable for the other models. This helps generate inputs that likely trigger LMM bugs for our target models, as discussed in Section 3.3. In this context, LTA found 232 *accident* cases (comprising 150, 51, and 31, respectively, for quadcopter, hexacopter, and VTOL models) out of the found 25,117 records for normal and accident flights. Of the 232 cases, LTA found that 72 cases are caused by LMM bugs. During our analyses, only those cases can be triggered by bug-triggering inputs, and all of them are reproducible. Furthermore, with our best efforts, we thoroughly analyzed all the 25,117 flight records manually due to the unavailability of accident ground truth. As a result, 232 cases are indeed accidents, and 72 cases are caused by LMM bugs. All of the bugs are successfully identified with 100% accuracy without false positives and negatives. We verify the correctness of this result. Also, if the root causes are our target bugs, they must cause reproducible accidents with the given inputs.

Table 1: Threshold values of all the primitive controllers for misbehavior detection.

Type	X-axis position (m)	Y-axis position (m)	Z-axis position (m)	X-axis velocity (m/s)	Y-axis velocity (m/s)	Z-axis velocity (m/s)	Roll angle (degree)	Roll angular rate (degree/s)	Pitch angle (degree)	Pitch angular rate (degree/s)	Yaw angle (degree)	Yaw angular rate (degree/s)
Quadcopter	2.60	2.57	3.5	1.93	1.93	2.0	15.98	59.09	17.10	60.97	167.30	150.0
Hexacopter	7.62	8.07	72.78	2.49	2.55	1.56	45.51	108.4	30.52	114.11	299.1	189.1
VTOL	170.6	235.2	2.0	20.62	19.67	3.5	15.97	42.07	40.0	43.30	300.0	29.17

Table 2: Statistics of the number of flight records.

	# of Flight Records			
	Quadcopter	Hexacopter	VTOL	Total
# of flights	23,432	1,216	469	25,117
# of flight accident cases	150	51	31	232
# of target flight accident cases	40	19	13	72

6.2 Bug Localization Results

We analyze 72 cases caused by LMM bugs. The majority of the root causes are a lack of or incorrect input sanitization logic, such as a lack of range check and incorrect conditions of an "if" statement. Those bugs can appear not only due to the failure to comply with their specifications [3, 4, 6] but also the incorrect specifications. Note that the latter can happen because the correct specification in a certain UAV model is not valid for another UAV model. However, developers do not deploy model-specific sanitization logic because they apply generic sanitization logic to support various models [41].

This procedure starts with **Flight Record Selection** (Section 4.2) selecting those bug cases from the public flight records [19]. Table 3 summarizes the details of all the inputs and bug-triggering inputs, the average number of misbehaving controllers, control loop iterations between bug-triggering input feeding time and initial misbehavior, and the number of the total investigated cases. We show those numbers according to the UAV control models denoted by *Quadcopter*, *Hexacopter*, and *VTOL* and all the UAVs denoted by *All models* in Column 1. As shown in Column 19, the numbers of the investigated cases are 40 for quadcopters, 19 for hexacopters, and 13 for VTOLs out of 72 cases.

First, we found that many of the inputs in public flight records are benign, as shown in their statistics in Columns 2-4. The number of all the inputs ranges from 5 to 20, and their average number is 10.61. Columns 2-4 show their numbers for respective UAV models. Through our control-level testing (Section 4.3), LTA identifies which inputs cause misbehavior, as shown in Columns 5-7. Specifically, the number of bug-triggering inputs ranges from 1 to 4. Columns 5-7 show the average number of bug-triggering inputs for all three models is 1.31 out of an average of 10.61 inputs, including the numbers for our target UAV models. Meanwhile, some of those inputs can cause misbehavior only if other inputs are fed together. We call such inputs "input pairs." We show the per-accident total numbers of input pairs in Columns 8-10, and the number of decomposed program-level traces is equal to the number of input pairs. Specifically, the number of bug-triggering input pairs ranges from 1 to 4 for all the UAV models, and its average number is 1.21. These results show that many accidents involve multiple bug-triggering inputs. This indicates LTA needs to pinpoint 1.31 bug-triggering inputs on average out of 10.61 inputs on average. Unlike LTA, existing work [36, 42] may identify false-negative or false-positive bug-triggering inputs, leading to incorrect LMM bug localization. For example, Mayday considers the last benign input relevant to a misbehaving controller as the bug-triggering input. Otherwise, we rely on existing techniques [39, 43] requiring huge testing time to find such specific bug-triggering inputs and their combinations (i.e., certain input pairs).

Moreover, LTA identifies the number of controllers for each accident, as shown in Columns 11-13. Note that this number is smaller than the number of bug-triggering inputs because some bug-triggering inputs target the same controller. The number of misbehaving controllers ranges from 1 to 3, with an average of 1.18. Those numbers show that some bug-triggering inputs can cause misbehavior in seemingly unrelated controllers. For example, we showed the motivating accident case where the pitch rate controller misbehaves in Section 3.4, but its initial misbehavior appeared from the z-axis position controller. Correct analysis requires our input decomposition and cross-domain analyses (Sections 4.3 and 4.4).

Next, we show the statistics of the length of control loop iterations in Columns 14-16. The implication of a bug can happen right after an arbitrary time. This can happen after some time due to the inertia or appear after a certain condition (e.g., after a sharp turning). In other words, there can be a time gap between the bug-triggering input and the initial misbehavior. We show such a time gap using the control loop iterations (one iteration takes 4ms). In detail, the length of control loop iterations for all three models ranges from 1,626 to 91,435, with an average of 15,189.2. Note that we consider the last fed bug-triggering input to measure the control loop iteration if misbehavior occurs by an "input pair". This is because an input pair's implication starts after all the pair's inputs are fed. Their time gap between those inputs can potentially lead to false positives in bug discovery because benign inputs fed in the middle of such iterations can cause analyzers to misidentify bugs.

Finally, we show the number of accidents and their misbehavior types (measured state and reference state divergence) that our control-level testing (Section 4.3) identifies. LTA found 56 cases where measured state divergence happened, as shown in Column 17. On the other hand, LTA discovered 21 instances of reference state divergence, as listed in Column 18. Note that the number of total cases is 72, which is five less than the sum of the cases where measured state divergence and reference state divergence appear. This is because our analysis with input and trace decomposition (Section 4.3) revealed the presence of both reference state divergence and measured state divergence in the five cases.

Comparison with Existing Work. Table 4 shows the statistics of control-model-level analysis results in comparison with Mayday [42]. We chose Mayday for comparison with LTA because this is the most recent LMM bug localization work. We inspected only its bug-triggering input algorithm to check whether it can detect bug-triggering inputs correctly because Mayday does not work with PX4. As a result, Mayday could not find all the bug-triggering inputs in 41 accident cases out of 72 cases (57%), as shown in Column 4.

Table 3: Per-accident statistics of bug-triggering input, control loop iterations, and misbehaving controllers.

UAV model	# of total inputs			# of bug-triggering inputs			# of bug-triggering input pairs			# of misbehaving controller			Length of control loop iterations			# of measured state divergence	# of reference state divergence	# of total investigated cases
	Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.			
Quadcopter	5	20	11.10	1	4	1.35	1	3	1.20	1	3	1.18	1626	74730	16710.20	34	8	40
Hexacopter	7	19	10.26	1	2	1.21	1	2	1.16	1	2	1.21	1646	28681	7868.07	11	9	19
VTOL	5	18	9.62	1	4	1.31	1	4	1.31	1	2	1.15	2965	91435	21024.36	9	4	13
All Models	5	20	10.61	1	4	1.31	1	4	1.21	1	3	1.18	1626	91435	15189.20	56	21	72

Table 4: Per-accident control-model-level analysis comparison with Mayday [42]. IM: incorrect accident record analysis results due to Mayday's initial-misbehavior-based detection algorithm, MI: incorrect accident record analysis results due to multiple bug-triggering inputs.

UAV model	Incorrect Results of Mayday				# of total accident cases
	# of IM	# of MI	# of IM ∪ MI	# of IM ∩ MI	
Quadcopter	24	11	28	7	40
Hexacopter	8	4	10	2	19
VTOL	2	2	3	1	13
Total	34	17	41	10	72

Table 5: Per-accident bug localization results. Avg.: average, BB: basic block, and LoC: lines of source code.

UAV model	# of BB			# of LoC		
	Min	Max	Avg.	Min	Max	Avg.
Quadcopter	21	80	39.43	31	165	79.48
Hexacopter	21	80	36.47	31	168	70.26
VTOL	19	58	30.85	29	120	59.23
All Models	19	80	37.10	29	168	73.39

Column 2 shows that 34 cases are caused by Mayday's assumption: an initial misbehaving controller has an LMM bug. However, initial misbehavior can originate from a controller different from the actual misbehaving controller, as discussed in Section 4.4. Furthermore, Column 3 shows that incorrect bug-triggering input identification from 17 cases is caused by Mayday's limitation: it can detect only one bug-triggering input out of the multiple bug-triggering inputs, preventing Mayday from detecting all the bug-triggering inputs. Out of them, Column 4 shows ten incorrect analysis results caused by both Mayday's problems. This shows LTA is superior to bug-triggering input and misbehavior identification.

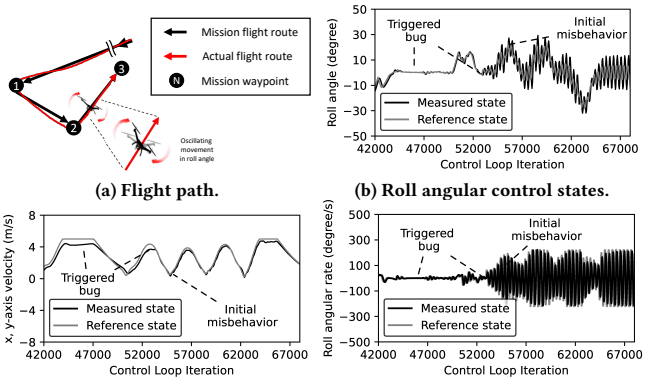
Bug Localization Results. Table 5 summarizes our LMM bug localization results. The number of basic blocks for all three models ranges from 19 to 80, with an average of 37.1, and the number of lines of code ranges from 29 to 168, with an average of 73.39. Those results show that LTA significantly reduces LoC that investigators need to analyze, considering the large code base whose LoC is 327,511 of our target software (PX4 v1.12.3). Note that we manually verified that LTA does not miss any bugs because LTA conservatively localizes LMM bugs, considering all the paths defined in control models as discussed in Section 4.4.

6.3 Step-by-Step Throughput

Now, we show the step-by-step throughput. The first step (**Flight Record Selection** in Section 4.2) is to filter out flight records that do not have inputs that likely trigger LMM bugs. The entire analysis time for all the cases, 25,117 flight records including normal flight records, is 11.25 hours. Each flight record analysis takes 1.73 seconds.

Table 6: Analysis time.

	Flight Record Selection (Section 4.2)	Control-Level Testing & Decomposed Trace Generation (Section 4.3)	Cross-Domain Bug Localization (Section 4.4)
Average (sec)	1.73	144.42	468.52
Total (hour)	11.25	15.11	9.37



(a) Flight path. (b) Roll angular control states. (c) x, y-axis velocity control state. (d) Roll rate control states.

Figure 6: Control states after Pair 1 triggers LMM bugs.

The second step (**Control-Level Testing & Decomposed Trace Generation** in Section 4.3) further reduces the cases where bugs cause accidents and then finds the bug-triggering input pairs of accidents caused by LMM bugs. This step analyzes 232 shortlisted flight records and their total analysis time is 15.11 hours. Each flight record analysis takes 144.42 seconds. The final step (**Cross-Domain Bug Localization** in Section 4.4) finds the bug-triggering input pairs and localize LMM bugs. This step analyzes 72 shortlisted flight records with 1.21 input pairs on average and takes 9.37 hours. Each flight record analysis takes 468.52 seconds.

6.4 Accident Case Study

In this section, we show how LTA localizes LMM bugs with one example accident case. This case differs from the motivating accident case in Section 3.4 to show LTA can address various misbehavior types and root causes. In this motivating case, LMM bugs exist within the pitch angular rate controller codes, but interestingly, initial misbehavior appears in the z-axis position controller which seems unrelated to bugs. This pattern makes existing approaches confused as specified in **Problem 2** in Section 3.3. However, LTA can accurately locate them. Meanwhile, this case study shows a misbehaving UAV caused by bug-triggering input pairs. To identify bugs related to all the input pairs, LTA should address **Problems 2 and 3**. Note that existing works [36, 39, 42, 43] can identify only some or none of the bug-triggering inputs and their relevant bugs.

Case Study: Oscillating Quadcopter Flight. We show how to localize LMM bugs from one accident case with a quadcopter model. After **Flight Record Selection** in Section 4.2, LTA performed control-level testing in Section 4.3 and found three input pairs: Pair 1 (MC_ROLL_P set to 17.9 and MPC_XY_VEL_P_ACC set to 4.78), Pair 2 (MPC_XY_VEL_P_ACC set to 54.79999), and Pair 3 (MPC_XY_P set to 0) with their corresponding traces.

In this case study, we focus on Pair 1, MC_ROLL_P (roll angular P control gain parameter) and MPC_XY_VEL_P_ACC (x, y-axis velocity P control gain parameter). These inputs, fed at 46,067 and 52,939 Iterations, respectively, caused severe measured state divergence in the roll rate angular control (Figure 6a). This misbehavior began at Iteration 55,004 in a roll rate angular control as shown in Figure 6d, followed by misbehavior in the roll angular and x, y-axis velocity controllers as shown in Figures 6b and 6c. This situation could cause incorrect analysis results because those inputs trigger bugs within the roll angular and x, y-axis velocity controllers — not the roll rate angular controller.

As a first analysis step, LTA identifies two control-model-level bug-triggering paths ($\mathcal{CP}$s) because there are two bug-triggering inputs based on our mapping graph in Figure 5. The path with an MC_ROLL_P input is $\mathrm{CI} \rightarrow \mathrm{C_{roll}} \rightarrow \mathrm{o_{roll}}$. On the other hand, the path with an MPC_XY_VEL_P_ACC is $\mathrm{CI} \rightarrow \dot{\mathrm{C}}_{\mathrm{xy}} \rightarrow \dot{\mathrm{o}}_{\mathrm{xy}}$.

On the identified $\mathcal{CP}$s, LTA derives program-level bug-triggering paths ($\mathcal{PP}$s) to localize LMM bugs, using the decomposed program-level trace. The initial misbehavior appeared at 55,004 Iteration as shown in Figure 6a but MC_ROLL_P and MPC_XY_VEL_P_ACC were changed at Iterations 46,067 and 52,939 respectively according to traces. For the trace with MC_ROLL_P, the path is $\mathrm{CI}_{46,067} \rightarrow \mathrm{C}_{\mathrm{roll},46,068} \rightarrow \mathrm{o}_{\mathrm{roll},55,004}$. For this trace, LTA identified 22 basic blocks (i.e., 48 LoC), which contain the bug related to MC_ROLL_P. On the other hand, $\mathrm{CI}_{52,939} \rightarrow \dot{\mathrm{C}}_{\mathrm{xy},52,940} \rightarrow \dot{\mathrm{o}}_{\mathrm{xy},55,004}$ is the path for the trace with MPC_XY_VEL_P_ACC. In this case, LTA identified 58 basic blocks (i.e., 116 LoC) containing the bug related to MPC_XY_VEL_P_ACC. As a result, for Pair 1, LTA identified 80 deduplicated basic blocks (i.e., 163 LoC).

7 DISCUSSION

Impacts of Bug Reproduction on Simulators. In Section 3.1, we discuss that accidents caused by external environmental factors (e.g., wind gusts) are out of scope. However, we admit that LMM bugs can be triggered when fed inputs and environmental factors meet "hard-to-satisfy" bug-triggering conditions coincidentally. Because LTA does not emulate such environmental factors, those bugs are false negatives for LTA.

To mitigate this limitation, we can use wind estimates from UAV software or wind data measured by airspeed sensors. Note that these wind estimates or measurements are typically recorded by mature UAV software, such as PX4 [19], ArduPilot [18], and Paparazzi [20]. If they are not available, we can alternatively incorporate realistic wind gust models [61, 64] with a Gazebo wind plugin [2].

Furthermore, LTA cannot discover bugs caused by real-time constraint violation [53, 54] and battery-related issues (e.g., battery depletion and failure), which can be a potential accident root cause but hard to reproduce in simulators. The reason is that simulated UAV programs encounter neither real-time constraint violations nor battery issues because they run on a high-performance computer, which has ample computation resources preventing real-time constraint violations and virtually unlimited power supply, unlike resource-constrained embedded systems with limited batteries.

Finally, faulty sensor failure and attacks are out of scope due to the difficulty in bug reproduction. Measured states (or sensor values) depend on various physical factors, such as wind gusts and sensor noise at flight time, with 100% accurate reproduction. However, such information is not recorded in logs, which limits bug reproduction for bug discovery and localization.

Barriers to Industrial Adoption. LTA can be adopted by industrial UAV management companies. Those industrial companies must have access to (1) the UAV software source code equipped with a flight log recorder (whose availability is mandated by law) and (2) a substantial number of flight records generated by its UAV software. Note that (1) is required to localize LMM bugs while (2) is necessary to obtain sufficient accident logs to find LMM bugs.

Impacts of Faulty Log Records. We acknowledge that log contents can be recorded imperfectly in practice. There are three types of issues: (1) incomplete log data, (2) delayed log data, and (3) missing log data. For (1), we assume that the recorded logs are complete. If the logs are incomplete at the accident time or earlier, LTA may fail to discover and localize LMM bugs.

Meanwhile, log data recording can be temporarily delayed, or some log data can be missing. Fortunately, LTA is resilient to both cases due to the two reasons. First, our misbehavior detection (Section 4.2) is robust because it can analyze other valid logs recorded after a few control loops. More importantly, such records are sufficient to identify "candidate records for accidents potentially caused by LMM bugs" — not necessarily the exact records for accident flights caused by LMM bugs, in **Flight Record Selection** (Section 4.2). This is because LTA ultimately reproduces flight operations and verifies if those records are truly relevant to LMM bug discovery.

Extended Support of Non-6DoF-Based Models. LTA can be extended to support non-6DoF-based models, such as unmanned ground vehicles (a model without a z-axis control), with additional engineering efforts. Such a vehicle model may have different cascading controller dependencies in Figure 1c, although it has primitive controller dependencies identical to the model in Figure 1d. Incorporating such a difference in model dependencies into our graph models (Section 4.1) can enable supporting such vehicle models.

8 RELATED WORK

LMM Bug Discovery in UAVs. There are proactive and reactive LMM bug discovery techniques. Proactive approaches aim to detect LMM bugs before flight. For example, some works [39, 40, 43, 59] propose fuzzing-based bug discovery techniques. However, they neither identify bug-triggering inputs nor localize LMM bugs. Furthermore, their fuzzing input mutation still needs to consider the large number of input options (e.g., over a thousand and large ranges of possible values), their interdependent combinations, and the considerable time to run flight testing with real-time progression. On the reactive approach side, LogAnalyzer [11] and Flight Review [7] support flight crash analysis. However, they can examine only the appearance of misbehavior, such as the measured angles within a valid range, mechanical faults, and battery issues,

rather than investigating accident root causes. He et al. [36] and Mayday [42] proposed LMM bug localization techniques. However, they cannot proactively identify bug-triggering inputs (e.g., Mayday requires an accident to discover even a single bug). Furthermore, they cannot pinpoint actual bug-triggering inputs if multiple inputs are mixed. Finally, RVPlayer [26] can address this issue. However, as admitted in the paper, it does not localize bugs. On top of that, it cannot detect and localize bugs from public flight records to find inputs that likely trigger bugs because it requires UAVs to record additional logs.

Bug Discovery in Cyber-Physical Systems. There are various bug discovery studies in different cyber-physical systems (e.g., industrial control systems (ICSes)) beyond the UAVs addressed in our research. These works primarily target either bugs within control software or the communication protocol implementations.

Specifically, three prior works [22, 24, 60] target bugs in control systems. Among them, two works [22, 60] target traditional software bugs. They are black-box fuzzers for ICSes without analyzing the interaction between cyber and physical components and misbehavior root causes. The other work [24] targets LMM bugs while analyzing the interaction between cyber and physical components, using a physical simulator as LTA does. However, it can detect only "measured state divergence" without localizing bugs.

On the other hand, several works [28, 30, 47, 63] target cyber-physical system fuzzing with a focus on ICS protocols. They detect protocol specification violation bugs or traditional software bugs. However, unlike LTA, none of these works necessitate and consider the physical behaviors of the target systems.

Finally, verification works have been introduced to discover safety issues in cyber-physical systems. Two prior works [35, 51] verify the safety of PLC or SCADA control systems, using the manually crafted safety violation detection rules. However, these approaches struggle to detect some LMM bugs because they rely on the safety rules defined only in their specifications (e.g., the water level must not exceed 50 centimeters) rather than general control invariants (e.g., rules to detect measured state divergence). Therefore, this approach is feasible only if target systems' specifications are available and has limited effectiveness in uncovering misbehaviors if they are unconsidered in the specification. Similarly, the other two works [29, 34] depend on manually crafted rules defined in specifications, but they target protocol-related issues, which differ from the problems addressed in this work.

Program Bug Testing and Localization. Many testing techniques [48, 49, 56, 65] have been published in recent years. Their main focus is how to optimize input mutation algorithms (e.g., coverage-guided testing) to trigger traditional bugs, such as memory corruption. However, they are not optimized to trigger LMM bugs whose implication occurs over long interactions with the physical world. On the other hand, many bug localization techniques [25, 50, 52, 62] have been proposed. For example, statistical bug localization can localize bugs by statistically comparing the different execution coverage patterns during respective runs [25, 62]. Other techniques [50, 52] analyze program crash logs or execution traces for bug localization. However, their common problems lack the "control" oracles to localize LMM bugs. Instead, LTA can detect misbehavior by analyzing control states.

9 CONCLUSION

We propose LTA, the replay-based proactive LMM bug discovery and localization framework by analyzing public flight accident records. Through our experiments, LTA accurately discovered and localized all 72 accident cases, some of which existing techniques cannot localize. Consequently, it helps enhance the safety and security of UAV software, preventing future UAV accidents and attacks.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd for their helpful feedback. This work results from a collaborative research project supported by the affiliated institute of ETRI.

REFERENCES

[1] 2020. ROS - Robot Operating System. https://www.ros.org.
[2] 2020. *Wind Plugin.* https://github.com/hongjun9/CPinconsistency/tree/master/plugins.
[3] 2021. *ArduPilot Parameter List.* http://ardupilot.org/copter/docs/parameters.html.
[4] 2021. *DJI Parameter Index.* https://dji.retroroms.info/howto/parameterindex.
[5] 2021. *FLIR's Black Hornet is the World's Smallest Military Drone.* https://flymotionus.com/2021/11/05/flirs-black-hornet-is-the-worlds-smallest-military-drone.
[6] 2021. *PX4 Parameter List.* https://dev.px4.io/en/advanced/parameter_reference.html.
[7] 2022. *Flight Review.* https://review.px4.io/browse.
[8] 2022. *Ilibipt - an Intel(R) Processor Trace decoder library.* https://github.com/intel/libipt.
[9] 2022. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3C: System Programming Guide, Part 3.* https://cdrdv2.intel.com/v1/dl/getContent/671506.
[10] 2022. *libdwarf, a library for reading DWARF2 and later DWARF.* https://github.com/davea42/libdwarf-code.
[11] 2022. *LogAnalyzer: Diagnosing problems using Logs for ArduPilot.* http://ardupilot.org/copter/docs/common-diagnosing-problems-using-logs.html.
[12] 2022. *MAVLink Micro Air Vehicle Communication Protocol.* https://mavlink.io.
[13] 2022. *QGC - QGroundControl - Drone Control.* http://qgroundcontrol.com.
[14] 2023. *DeltaQuad Pro VTOL UAV.* https://www.deltaquad.com.
[15] 2023. *Draco.* https://www.uvify.com/draco.
[16] 2023. *H520E/520 - Yuneec.* https://us.yuneec.com/h520-series.
[17] 2023. *Whole Program LLVM in Go.* https://github.com/SRI-CSL/gllvm.
[18] 2024. *ArduPilot.* http://ardupilot.org.
[19] 2024. *PX4 Pro Open Source Autopilot - Open Source for Drones.* http://px4.io.
[20] 2024. *Welcome to Paparazzi UAV.* https://wiki.paparazziuav.org.
[21] Gino Brunner, Bence Szebedy, Simon Tanner, and Roger Wattenhofer. 2019. The urban last mile problem: Autonomous drone delivery to your balcony. In *Proceedings of the 2019 International Conference on Unmanned Aircraft Systems (ICUAS).*
[22] Andrei Bytes, Prashant Hari Narayan Rajput, Constantine Doumanidis, Michail Maniatakos, Jianying Zhou, and Nils Ole Tippenhauer. 2023. Fieldfuzz: In situ blackbox fuzzing of proprietary industrial automation runtimes via the network. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID).*
[23] Aurélien Cabarbaye, Rogelio Lozano Leal, Patrick Fabiani, and Moisés Bonilla Estrada. 2016. VTOL aircraft concept, suitable for unmanned applications, with equivalent performance compared to conventional aeroplane. In *Proceedings of the 2016 International Conference on Unmanned Aircraft Systems (ICUAS).*
[24] Yuqi Chen, Bohan Xuan, Christopher M Poskitt, Jun Sun, and Fan Zhang. 2020. Active fuzzing for testing and securing cyber-physical systems. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA).*
[25] Trishul M. Chilimbi, Ben Liblit, Krishna Mehra, Aditya V. Nori, and Kapil Vaswani. 2009. HOLMES: Effective Statistical Debugging via Efficient Path Profiling. In *Proceedings of the 31st International Conference on Software Engineering (ICSE).*
[26] Hongjun Choi, Zhiyuan Cheng, and Xiangyu Zhang. 2022. RVPLAYER: Robotic Vehicle Forensics by Replay with What-if Reasoning. In *Proceedings of 29th Annual Network and Distributed System Security Symposium (NDSS).*
[27] Devon R Clark, Christopher Meffert, Ibrahim Baggili, and Frank Breitinger. 2017. DROP (DRone Open source Parser) your drone: Forensic analysis of the DJI Phantom III. *Digital Investigation* 22 (2017), S3–S14.
[28] Ganesh Devarajan. 2007. Unraveling SCADA protocols: Using sulley fuzzer. In *Defon 15 hacking conference.*

[29] Jannik Dreier, Maxime Puys, Marie-Laure Potet, Pascal Lafourcade, and Jean-Louis Roch. 2019. Formally and practically verifying flow properties in industrial systems. *Computers & Security* 86 (2019), 453–470.

[30] Dongliang Fang, Zhanwei Song, Le Guan, Puzhuo Liu, Anni Peng, Kai Cheng, Yaowen Zheng, Peng Liu, Hongsong Zhu, and Limin Sun. 2021. Ics3fuzzer: A framework for discovering protocol implementation bugs in ics supervisory software by fuzzing. In *Proceedings of the 37th Annual Computer Security Applications Conference (ACSAC)*.

[31] Bo Feng, Alejandro Mera, and Long Lu. 2020. P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*.

[32] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal de Lara. 2005. The Taser Intrusion Recovery System. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP)*.

[33] Dunstan Graham and Richard C Lathrop. 1953. 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 (1953), 273–288.

[34] Enrico Guerra, Mariano Ceccato, and Marco Rocchetto. 2022. Formal verification and risk assessment of an implementation of the OPC-UA Protocol. *Universit' a degli Studi di Verona* (2022).

[35] Nannan He, Victor Oke, and Gale Allen. 2016. Model-based verification of PLC programs using Simulink design. In *2016 IEEE International Conference on Electro Information Technology (EIT)*. IEEE, 0211–0216.

[36] Zhijian He, Yao Chen, Enyan Huang, Qixin Wang, Yu Pei, and Haidong Yuan. 2019. A System Identification based Oracle for Control-CPS Software Fault Localization. In *Proceedings of the IEEE/ACM 40th International Conference on Software Engineering (ICSE)*.

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

[38] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. 2018. Securing Real-Time Microcontroller Systems through Customized Memory View Switching. In *Proceedings of the 27th Annual Symposium on Network and Distributed System Security (NDSS)*.

[39] Hyungsub Kim, Muslum Ozgur Ozmen, Antonio Bianchi, Z Berkay Celik, and Dongyan Xu. 2021. PGFUZZ: Policy-Guided Fuzzing for Robotic Vehicles. In *Proceedings of the 30th Annual Symposium on Network and Distributed System Security (NDSS)*.

[40] Seulbae Kim and Taesoo Kim. 2022. RoboFuzz: fuzzing robotic systems over robot operating system (ROS) for finding correctness bugs. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*.

[41] Taegyu Kim, Aolin Ding, Sriharsha Etigowni, Pengfei Sun, Jizhou Chen, Luis Garcia, Saman Zonouz, Dongyan Xu, and Dave (Jing) Tian. 2022. Reverse Engineering and Retrofitting Robotic Aerial Vehicle Control Firmware Using Dispatch. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services (MobiSys)*.

[42] Taegyu Kim, Chung Hwan Kim, Altay Ozen, Fan Fei, Zhan Tu, Xiangyu Zhang, Xinyan Deng, Dave (Jing) Tian, and Dongyan Xu. 2020. From Control Model to Program: Investigating Robotic Aerial Vehicle Accidents with MAYDAY. In *Proceedings of 29th USENIX Security Symposium (USENIX Security)*.

[43] Taegyu Kim, Chung Hwan Kim, Junghwan Rhee, Fan Fei, Zhan Tu, Gregory Walkup, Xiangyu Zhang, Xinyan Deng, and Dongyan Xu. 2019. RVFuzzer: Finding Input Validation Bugs in Robotic Vehicles through Control-Guided Testing. In *Proceedings of 28th USENIX Security Symposium (USENIX Security)*.

[44] Nathan P Koenig and Andrew Howard. 2004. Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

[45] Y. Kwon, J. Yu, B. Cho, Y. Eun, and K. Park. 2018. Empirical Analysis of MAVLink Protocol Vulnerability for Attacking Unmanned Aerial Vehicles. *IEEE Access* 6 (2018), 43203–43212. https://doi.org/10.1109/ACCESS.2018.2863237

[46] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*.

[47] Pei-Yi Lin, Chia-Wei Tien, Ting-Chun Huang, and Chin-Wei Tien. 2021. ICPFuzzer: proprietary communication protocol fuzzing by using machine learning and feedback strategies. *Cybersecurity* 4 (2021), 1–15.

[48] Yi Liu, Yuekang Li, Gelei Deng, Yang Liu, Ruiyuan Wan, Runchao Wu, Dandan Ji, Shiheng Xu, and Minli Bao. 2022. Morest: Model-based RESTful API Testing with Execution Feedback. In *Proceedings of 44th IEEE/ACM International Conference on Software Engineering (ICSE)*.

[49] Chenyang Lyu, Hong Liang, Shouling Ji, Xuhong Zhang, Binbin Zhao, Meng Han, Yun Li, Zhe Wang, Wenhai Wang, and Raheem Beyah. 2022. SLIME: Program-sensitive Energy Allocation for Fuzzing. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.

[50] Samaneh Navabpour, Borzoo Bonakdarpour, and Sebastian Fischmeister. 2011. Software Debugging and Testing Using the Abstract Diagnosis Theory. In *Proceedings of the 2011 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*.

[51] Fadi Obeid and Philippe Dhaussy. 2019. Formal verification of security pattern composition: application to SCADA. *Computing and Informatics* 38, 5 (2019), 1149–1180.

[52] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. 2009. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*.

[53] Junayed Pasha, Zeinab Elmi, Sumit Purkayastha, Amir M. Fathollahi-Fard, Ying-En Ge, Yui-Yip Lau, and Maxim A. Dulebenets. 2022. The Drone Scheduling Problem: A Systematic State-of-the-Art Review. *IEEE Transactions on Intelligent Transportation Systems* 23, 9 (2022), 14224–14247.

[54] Andrea Patelli and Luca Mottola. 2016. Model-based real-time testing of drone autopilots. In *Proceedings of the 2nd Workshop on Micro Aerial Vehicle Networks, Systems, and Applications for Civilian Use (DroNet)*. 11–16.

[55] Nils Rodday. 2016. Hacking a Professional Drone. *Blackhat ASIA* (2016).

[56] Majid Salehi, Luca Degani, Marco Roveri, Danny Hughes, and Bruno Crispo. 2023. Discovery and Identification of Memory Corruption Vulnerabilities on Bare-Metal Embedded Devices. *IEEE Transactions on Dependable and Secure Computing (TDSC)* 20, 2 (2023), 1124–1138.

[57] Dongjie Shi, Xunhua Dai, Xiaowei Zhang, and Quan Quan. 2017. A practical performance evaluation method for electric multicopters. *IEEE/ASME Transactions on Mechatronics* 22, 3 (2017), 1337–1348.

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

[59] Christopher Steven Timperley, Afsoon Afzal, Deborah S Katz, Jam Marcos Hernandez, and Claire Le Goues. 2018. 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)*.

[60] Dimitrios Tychalas, Hadjer Benkraouda, and Michail Maniatakos. 2021. ICSFuzz: Manipulating I/Os and Repurposing Binary Code to Enable Instrumented Fuzzing in ICS Control Applications. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*.

[61] G. E. Uhlenbeck and L. S. Ornstein. 1930. On the Theory of the Brownian Motion. *Phys. Rev.* 36 (Sep 1930), 823–841. Issue 5. https://doi.org/10.1103/PhysRev.36.823

[62] Zhaogui Xu, Shiqing Ma, Xiangyu Zhang, Shuofei Zhu, and Baowen Xu. 2018. Debugging with intelligence via probabilistic inference. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*.

[63] Hyunguk Yoo and Taeshik Shon. 2016. Grammar-based adaptive fuzzing: Evaluation on SCADA modbus protocol. In *Proceedings of the 2016 IEEE International conference on smart grid communications (SmartGridComm)*. 557–563.

[64] Rafael Zárate-Miñano, Marian Anghel, and Federico Milano. 2013. Continuous wind speed models based on stochastic differential equations. *Applied Energy* 104 (2013), 42–49.

[65] Qian Zhang, Jiyuan Wang, and Miryung Kim. 2021. Heterofuzz: Fuzz testing to detect platform dependent divergence for heterogeneous applications. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[66] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. 2021. Automatic Firmware Emulation through Invalidity-guided Knowledge Inference. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*.