Reverse Engineering and Retrofitting Robotic Aerial Vehicle Control Firmware using DisPatch

Taegyu Kim
The Pennsylvania State University
tgkim@psu.edu

Aolin Ding
Security R&D,
Accenture Labs, Accenture
a.ding@accenture.com

Sriharsha Etigowni
Purdue University
setigown@purdue.edu

Pengfei Sun
F5 Networks
p.sun@f5.com

Jizhou Chen
Purdue University
chen2731@purdue.edu

Luis Garcia
University of Southern California,
Information Sciences Institute
lgarcia@isi.edu

Saman Zonouz
Georgia Institute of Technology
samance@gmail.com

Dongyan Xu
Purdue University
dxu@purdue.edu

Dave (Jing) Tian
Purdue University
daveti@purdue.edu

ABSTRACT

Unmanned Aerial Vehicles as a service (UAVaaS) has increased the field deployment of Robotic Aerial Vehicles (RAVs) for different services such as transportation and terrain exploration. These RAVs are controlled by firmware, which is often closed-source, developed by vendors, and flashed into the ROM. While these binary blobs enable off-the-shelf management of RAVs, end users (individuals or organizations) have no idea if the control firmware is designed and implemented correctly, and can only rely on firmware updates from vendors when any vulnerability is discovered. This paper proposes DisPatch, the first reverse engineering and patching framework for understanding and improving controller design and implementation within RAV firmware. DisPatch first decompiles binary instructions and recovers controller functions and core controller variables by combining control theory with program analysis using symbolic execution and data flow analysis. End users can then write a patch in a domain-specific language (DSL), which will be translated and injected into the binary firmware by DisPatch automatically. We have applied DisPatch to two instances of commodity firmware from 3DR IRIS+ and MantisQ RAVs and demonstrated 100% and 80.7% accuracy respectively in the controller decompilation. We have also shown the ability to prevent severe controller performance degradation by patching two real-world bugs within the firmware and without breaking other functionality. Finally, we show that DisPatch introduces less than 0.53% of space overhead and 1.48% of runtime overhead without violating the soft real-time deadlines. DisPatch provides the first step towards an RAV binary firmware reverse engineering and patching system to customize controller design and implementation.

MobiSys '22, June 25–July 1, 2022, Portland, OR, USA

ACM ISBN 978-1-4503-9185-6/22/06…$15.00
https://doi.org/10.1145/3498361.3538938

CCS CONCEPTS

• **Computer systems organization → Firmware**; • **Security and privacy → Software reverse engineering**; **Mobile platform security**.

KEYWORDS

Robotic Aerial Vehicle, Binary Analysis, Firmware Analysis, Security

ACM Reference Format:
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 *The 20th Annual International Conference on Mobile Systems, Applications and Services (MobiSys '22), June 25–July 1, 2022, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3498361.3538938

1 INTRODUCTION

Unmanned Aerial Vehicles as a Service (UAVaaS) [103] has increased the field deployment of Robotic Aerial Vehicles (RAVs) for different services, including disaster relief [83], espionage [8], and delivery services [46, 58]. In addition, RAV manufacturers such as Yuneec [39], Cheerson [1], and CUAV Raefly [23] further enable UAVaaS by providing off-the-shelf RAVs that are easy to use and configure for different purposes. RAV developers usually implement certain control algorithms, such as Proportional-Integral-Derivative (PID) controllers [81], in firmware to control the flight. While these binary blobs enable off-the-shelf management of RAVs, RAV operators, including individuals or organizations (e.g., an air delivery company), have no knowledge of the implementation details within the firmware, let alone checking the correctness of the design and implementation. Consequently, they can only rely on firmware updates from RAV vendors when suboptimal or buggy controller design and implementation within firmware are discovered.

Suboptimal controller design and implementation can degrade RAV control performance. For example, changes to an RAV's physical attributes, such as payload, may render the original controller design suboptimal. While end users can modify an RAV's physical attributes

with valid reasons (e.g., replacing a discontinued physical component or mounting a camera), they cannot easily mitigate the impact of the now-suboptimal controller, without the control firmware's source code or updates from the vendor. As a result, the RAV, without proper control firmware retrofitting, may fail in stabilizing its physical movements or tracking its designated flight trajectory.

Moreover, RAV control firmware may contain *control-semantic bugs* [71, 72], e.g., bugs in PID controller implementation caused by over-generalization of vehicle dynamics (detailed cases in Sections 3 and 6.2). For instance, the velocity control gain parameter of a z-axis controller is set large enough to accommodate a diverse range of RAV *physical features and properties*, such as frame size and shape (e.g., quadcopter vs. fixed-wing plane). As a result, these "loose" ranges of control variables leave room for mistaken operations that convert into attacks against certain RAV models without corrupting memory or relying on other typical software bugs, leading to highly unstable flights, mission failures, or even crashes [2, 3, 6, 75, 84].

While semantic disassemblers and binary patching frameworks exist, none of them can be applied to controllers within RAV firmware. State-of-the-art semantic disassemblers can only identify the locations of PID controller functions [92] or unchanged library functions (e.g., PID functions provided as library functions) [66] within the firmware, but cannot recognize customized variants of the same controller functions (e.g., custom P, PI, and PID controllers), let alone determining the semantics of these controllers (e.g., z-axis position or roll angle control), which is crucial for RAVs. Similarly, existing binary patching frameworks are either designed for typical binaries without the knowledge of an RAV control model and mandating user operations at the instruction level directly [70, 97], or requiring dynamic execution frameworks [79, 80] assuming a full-fledged operating system, e.g., Linux. A *stripped* RAV firmware further impedes applying existing binary patching frameworks to RAV firmware.

In this paper, we present DisPatch, the first RAV firmware binary reverse engineering and patching framework for understanding and improving controller design and implementation. Compared with existing semantic disassemblers, DisPatch is able to "decompile" a number of PID controllers within firmware, recognizing both controller variables (e.g., control gain and target z-axis velocity) and controller semantics. Compared with existing binary patch frameworks, DisPatch allows end users to write a patch (a.k.a., *control-semantic patch*) in a high-level Domain-Specific Language (DSL), which gets translated and then injected into binary firmware automatically. Specifically, we use symbolic execution to extract symbolic abstract syntax trees (ASTs) of identified PID design and implementation within firmware, and compare them with AST templates of standardized PID controller algorithms to recognize controller variables. We further leverage data flow analysis to capture the representative RAV controller structure and inter-controller dependencies to infer controller semantics.

We evaluated DisPatch on two instances of commodity RAV firmware from 3DR IRIS+ [7] and Mantis Q [30] RAVs, which are customized upon ArduPilot [15] and PX4 [35]. We also applied DisPatch to open-source ArduPilot firmware with three RAV physical specifications (i.e., a helicopter, a plane, and a submarine). Our evaluation shows that DisPatch identifies and decompiles controller functions with 100% accuracy for the ArduPilot-based firmware and 80.7% accuracy for the MantisQ firmware. Meanwhile, DisPatch only incurs 0.53% space and 1.48% runtime overheads without violating the soft real-time deadlines set in ArduPilot or introducing notable overhead in MantisQ. Our contributions are as follows:

- We propose a semantic decompiler to locate the PID controller functions and variables and recover their semantics via symbolic execution and data flow analysis guided by the RAV control model.
- We design and implement DisPatch, enabling end users to write high-level controller patches using a domain-specific language (DSL), which gets translated into binary-level instrumentation automatically to improve controller design and implementation within RAV firmware.
- We evaluate DisPatch against 3DR IRIS+ copter, MantisQ copter, and three instances of open-source ArduPilot-based firwmare. Our evaluation shows that DisPatch achieves 100% accuracy in controller identification and decompilation for the 3DR IRIS+ copter firmware and three instances of open-source ArduPilot-based firmware, and 80.7% accuracy for the MantisQ firmware, with only incurring at most 0.53% space and 1.48% runtime overheads.

2 BACKGROUND

RAV Control Model and Physical Features and Properties. RAV control models consist of the three following components: sensor modules, controller modules, and engine modules. During a flight, multiple sensor modules (e.g., GPS and gyroscopes) measure the physical movements (e.g., measured z-axis velocity of an RAV) along the six degrees of freedom (6DoFs), x, y, z-axes, roll, pitch, and yaw, as shown in Figure 1a. Controller modules maintain internal states (i.e., controller states) by generating proper control signals according to the measured physical movements along the 6DoF. Engine modules (e.g., motors) properly generate propulsion forces according to the control signals sent from controller modules.

To control the physical movements, controller modules run the control program implementing the RAV control model. Typically, RAVs use the common control model orchestrating multiple controllers (e.g., a z-axis controller). Each controller relies on a control algorithm to properly update its states. Those controller states include measured control variables (e.g., measured z-axis velocity) and target control variables (e.g., target z-axis velocity) Among the multiple controller algorithms, the proportional-integral-derivative (PID) controller prevails [48, 81, 104].

While the same RAV control model is commonly shared among different RAVs, its internal controllers must be properly customized and tuned for each RAV. This is because the physical movements with the same propulsion force can appear differently according to the physical features and properties of an RAV. Specifically, multiple factors determine the physical features and properties, such as weights, propeller lengths, number of motors, and frame shapes. In particular, if the frame shape and number of motors follow features and properties of the well-known models (e.g., a copter, helicopter, plane, or submarine), we refer to them as *physical specifications*. To tune controllers, RAV operators can configure control parameters via the (remote) ground control station (GCS) interface to adapt the target RAV's physical features and properties.

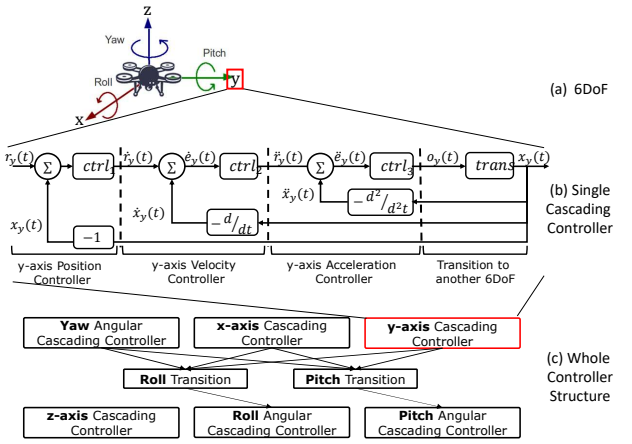


Figure 1: An RAV's six degrees of freedom (6DoF) and its controller structure and controller to control 6DoF.

PID Controllers and Controller States. To guarantee stable physical operations, a set of PID controllers must be customized for each RAV physical feature and property. Figure 1a shows that an RAV controls physical movements along the 6DoF, involving six *cascading* controllers. Each of them is responsible for controlling each of 6DoF, such as the y-axis cascading controller, as described in Figure 1b. Zooming in the y-axis cascading controller, there are three *primitive* controllers (denoted as $ctrl_1$, $ctrl_2$, and $ctrl_3$) respectively computing the y-axis position, velocity, and acceleration operations. These primitive controllers are usually designed and implemented as PID controllers in RAVs [15, 22, 35], mathematically expressed as follows:

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

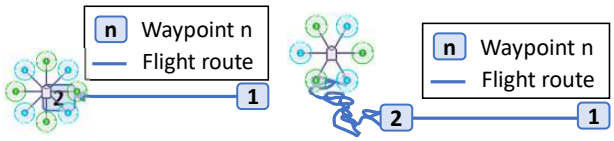
$$e(t) = r(t) - x(t) \quad (2)$$

In Equations 1 and 2, $x(t)$ denotes the *vehicle state*; $r(t)$ denotes the *reference* indicating the target state; $e(t)$ is the *control error*, which is the difference between the vehicle state and the reference. In this mathematical expression, there are four coefficients (P, I, D, and FF) multiplying either $e(t)$ or $r(t)$. These coefficients are also known as *control parameters*, pre-configured to properly control an RAV.

Each controller generates the output as the input of its adjacent component. For example, if the output of a primitive controller flows to its adjacent primitive controller, $x(t)$ and $r(t)$ (generating $e(t)$) act as an input of another dependent primitive controller. We call such dependencies as *primitive controller dependencies*. The output can also flow to a motor (e.g., $o_y(t)$) as a motor throttle or an input reference for another cascading controller (e.g., from the y-axis acceleration controller to the pitch angular controller). Such connections between cascading controllers create dependencies in the RAV control model, which we call *cascading controller dependencies*. We describe the common RAV controller structure consisting of six cascading controllers with their dependencies in Figure 1c [71].

3 MOTIVATION

RAV controller design and implementation can be suboptimal or even buggy because RAV vendors may not perform per-vehicle or per-mission customization of RAV control programs and parameters. For example, previous research [63, 71, 72] has reported suboptimal RAV control performance when a vehicle's generic control firmware is not fine-tuned for its specific physical features or missions. Such weaknesses can be exploited by adversaries to launch malicious attacks.



(a) Stable flight with an octacopter. **(b) Unstable flight with a hexacopter.**

Figure 2: An example implication of x, y-axis velocity PID parameter manipulation on the octacopter and hexacopter running on the identical copter firmware.

As a real-world example, ArduPilot supports both hexacopter and octacopter models. Despite their distinct differences in physical specifications, both copters share the same control program code base and control parameter specification [16, 34]. Simply put, the RAV firmware is exactly the same for both models in ArduPilot by default without tuning. Let us assume that we set the P (`PSC_VELXY_P`), I (`PSC_VELXY_I`), and D (`PSC_VELXY_D`) parameters of the x, y-axis velocity controllers, and the P (`PSC_POSXY_P`) parameter of the x, y-axis position controller respectively as 6.0, 0.02, 0.01, and 2.0 while keeping other default values. Note that those values are valid according to the specification, and they can be (remotely) set by either operators or attackers [71, 72]. We then launch both the hexacopter and the octacopter to monitor the construction progress of a building. The RAV should move from Waypoint 1 to 2 (to the west direction) and hover at Waypoint 2 to monitor the construction progress, as shown in Figure 2. While the octacopter is able to finish the task accordingly, the hexacopter starts to show severe and persistent controller degradation after a few seconds, leading to a crash in the end.

Furthermore, an RAV user needs to tune a controller's design and implementation within RAV firmware when a user has legitimate reasons to modify the physical features and properties, such as weight change (e.g., loading an extra item for delivery) or replacing motors or wings if they are discontinued or unavailable anymore. Such events lead to invalidating the previously tuned controller design and implementation.

For instance, a quadcopter user customized her drone by mounting a camera on the shell. Before this change, the quadcopter could follow each waypoint to finish a task as shown in Figure 4a. After this change, this quadcopter still flied well until reaching Waypoint 3. However, it could not reach Waypoint 4 after the sharp turning point (Waypoint 3), as shown in Figure 4b, and crashed near Waypoint 4 unexpectedly later. This happens because the quadcopter could not keep changing the flight direction fast enough by changing the pitch angle swiftly. The root cause of the slow pitch angle change is the low P gain (`ATC_ANG_PIT_P`) of the pitch angular controller compared to the heavier weight than usual.

Those motivating examples above highlight the existence of suboptimal controller design and implementation within RAV firmware and even indicate the feasibility of attacks against RAVs (e.g., exploiting control-semantic bugs). A possible fix might be as simple as adding an extra line in the source file to rule out those parameter

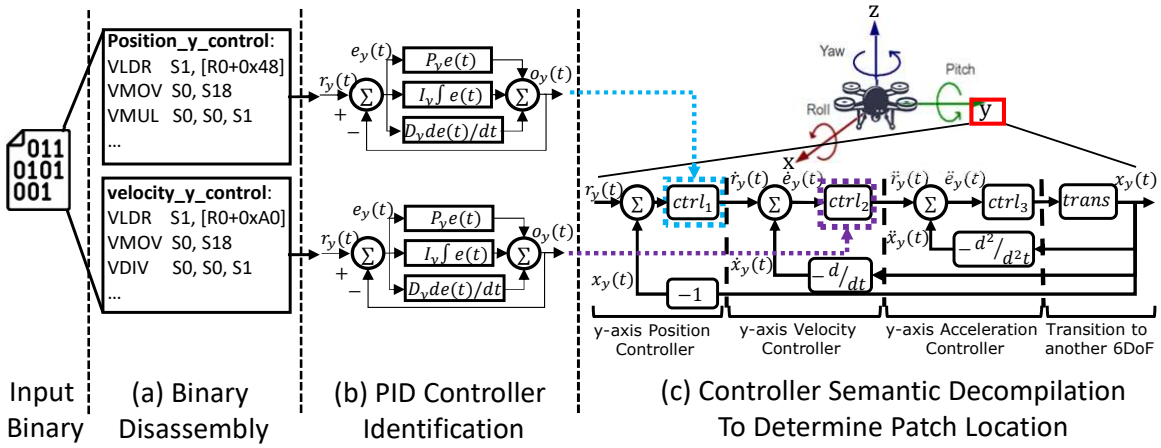
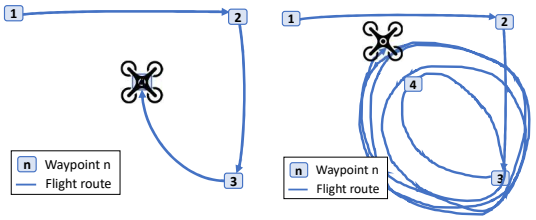


Figure 3: The necessity of identifying semantics of controller functions to solve Challenge 1 and 2.



(a) No trajectory deviation before increasing weight. **(b) Trajectory deviation after increasing weight.**

Figure 4: An example implication of the weight change on the pitch angular controller of a quadcopter.

values for the hexacopter. However, when source files are not available, users can only rely on firmware updates from RAV vendors and hope the controller design and implementation is improved in the updates. Unfortunately, as shown in our experiments in Section 6.2, sloppy vendors might decide to leave the suboptimal and potentially vulnerable controller design and implementation as is.

Alternatively, users can patch firmware using binary patching frameworks [70, 79, 80, 97]. Fortunately, those binary patching frameworks can provide fundamental patching functionality applicable to RAV firmware. However, this requires identifying controller code locations to patch in advance. Unfortunately, none of the existing semantic disassemblers [66, 92] can reverse engineer these controller design and implementation within RAV firmware due to the two following unique challenges.

Challenge 1: Multiple Variants of Controller Functions. We need to not only disassemble the binary code (as shown in Figure 3a) but also identify PID controller functions from the binary code (as illustrated in Figure 3b) before we can patch suboptimal controller design and implementation within RAV firmware. In reality, some controllers either omit some parameter variables if they are unnecessary (e.g., a PID controller designed and implemented only with one P parameter) or contain additional control parameters. In the former case, these controllers are still legitimate PID controllers with some parameter values (e.g., an I parameter) fixed at zero. In the latter case, additional mechanisms (e.g., an FF parameter) can be added. Accordingly, control-semantic patching should mandate the recognition of all these variants of controller functions within RAV firmware.

Challenge 2: Controllers with Different Semantics. PID controllers are generic controller models that can be used to control any physical operations (e.g., any of 6DoF). In other words, even if we find a specific controller variable (e.g., P control parameter), we cannot figure out whether that is the z-axis velocity or the roll angular P parameter, without knowing the exact semantics of each controller, let alone applying patches to the target parameters correctly. For example, we should not apply a patch for the z-axis velocity P parameter to the z-axis acceleration P parameter or even the y-axis velocity P parameter. As shown in Figure 3c, control-semantic patching demands the recovery of all the semantics of controller functions within RAV firmware.

4 DESIGN

To understand and improve suboptimally designed and implemented controllers within RAV firmware, we present DisPatch, an automatic reverse engineering and patching framework for end users. Overall, DisPatch identifies not only PID controller functions but also their variants (e.g., P, PID FF controllers). It further decompiles these controller functions to recover controller variables and semantics (e.g., z-axis velocity P parameter or roll angular P parameter). End users can then write a patch in a high-level DSL, which gets mapped into the corresponding controller function and variables by DisPatch, and translated into binary-level instrumentation to improve controller design and implementation. Figure 5 shows the general workflow of DisPatch.

To patch RAV firmware, DisPatch takes the four following inputs: (1) a (stripped) RAV firmware with the memory layout of the platform board, (2) a list of configurable control parameters, (3) common trigonometric function implementations, such as *sin* in a compiler's standard library (e.g., `libm`), and (4) a control-semantic patch written in the high-level DSL (e.g., enforcing values of controller variables within safe ranges). We note that those inputs do not require end users to have much background knowledge. We

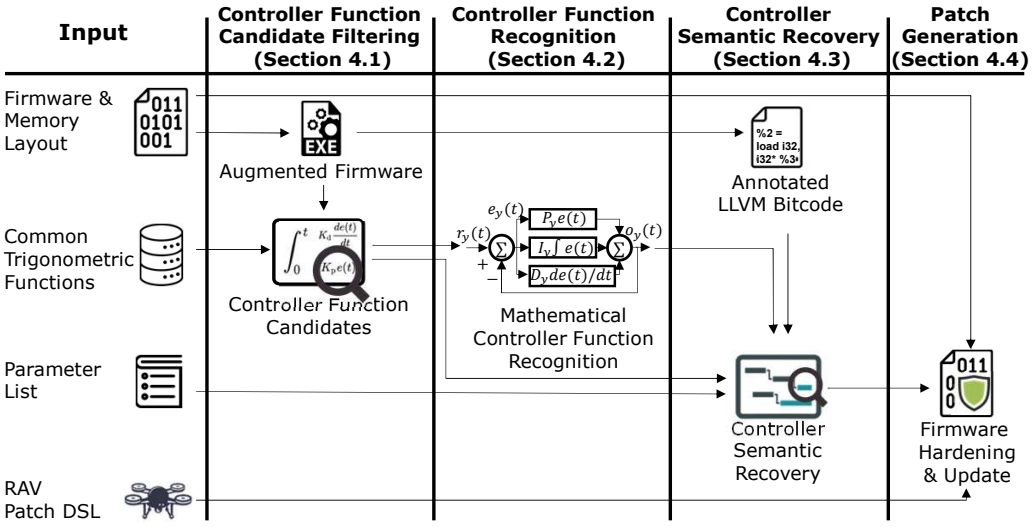


Figure 5: The architecture of DisPatch.

can obtain the memory layout by identifying the microcontroller unit (MCU) chip (e.g., STM32F427) on the target board [73]. A list of configurable control parameters is usually provided by vendors (e.g., ArduPilot [16], PX4 [34], and DJI [24]) or can be obtained by launching the command defined in a standard RAV communication protocol [32]. Finally, prior works have shown that the safe value ranges for controller variables in (4) can be automatically obtained [63, 69, 71, 72]. DisPatch then performs the following procedures to apply the patch to the input binary firmware.

(1) **Controller Function Candidate Filtering (Section 4.1)**: DisPatch starts with augmenting the binary firmware by annotating the memory layout, disassembling the binary firmware, and collecting a candidate list of controller functions using static analysis.

(2) **Controller Function Recognition (Section 4.2)**: To recognize customized PID controllers and their controller variables (a.k.a. **Challenge 1**), DisPatch captures the mathematical operations of each candidate controller function using symbolic execution [89] and converts the outcome into a symbolic abstract syntax tree (AST). DisPatch then compares and matches these binary-derived ASTs with the reference AST templates generated from the common PID design and implementation to recognize these controller functions and controller variables.

(3) **Controller Semantic Recovery (Section 4.3)**: To further recover the semantics of each controller (a.k.a. **Challenge 2**), DisPatch performs data flow analysis to identify the program-level dependencies of primitive and cascading controllers guided by the RAV control model. DisPatch then annotates the semantics of controller variables identified in the previous step (Section 4.2). We note that DisPatch identifies instructions within controller functions accessing those annotated controller variables as potential patch locations.

(4) **Patch Generation (Section 4.4)**: Using the results of the previous steps, users can prepare for a control-semantic patch written in a high-level DSL. DisPatch automatically finds the corresponding patch locations, translates and injects the patch at the binary level, and emits the hardened RAV firmware. Then, DisPatch uploads it onto the board (as described in Section 6).

4.1 Controller Function Candidate Filtering

Firmware Augmentation. We identify the PID controller candidates within firmware using static analysis, starting with augmenting the binary firmware with the memory layout information (i.e., code and data memory addresses) of the target platform. This memory layout information is publicly available from the target hardware board documents, such as system view description (SVD) files [21] written in the parsable Extensible Markup Language (XML) format or board-specific firmware development tools. The augmented firmware provides the base for the next step.

Shortlisting Controller Function Candidates. Observing that these controller functions require floating-point operations, we generate a list of function candidates that use floating-point instructions. This coarse-grained approach significantly reduces the number of candidate functions from thousands to hundreds because most non-controller functions, such as `malloc` and `strcpy`, do not use floating-point instructions. In order to further reduce the number of candidates, we build common controller functions, e.g., PID controllers, by ourselves and generate a reference binary dataset for the target platform. Using the reference binary dataset, we analyze each controller function in the assembly format and extract the features, e.g., the number of floating-point addition, subtraction, multiplication, and division instructions, as heuristics. We then use these heuristics to define filtering rules, e.g., specifying the range of occurrence count for each type of floating-point instructions in a controller function, further pruning the candidate list into only tens of candidates. Those candidate functions include floating-point instructions essential for control computation — not just loading and storing (e.g., logging controller states) or simple multiplication (e.g., conversion from `cm` to `m`) used for non-controller functions.

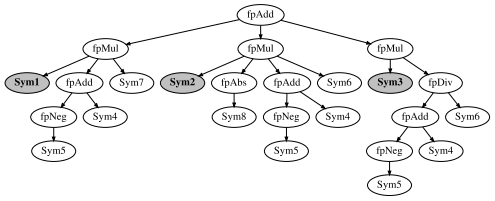
Algorithm 1 Controller Function & Variable Identification

Input: Mathematical controller function candidates (FC), AST templates based on high-level control algorithms (AST_{tmpl})
Output: Controller function list (FL) and controller variable list (CV)

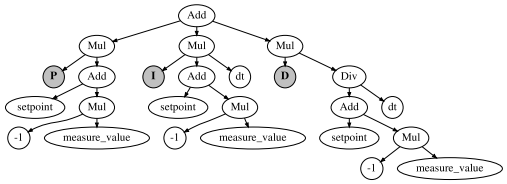
1: **function** BuildAST($NodeExpr$)
2: **for** $arg \in NodeExpr.args$ **do** ▷ Check every operator
3: $childNode \leftarrow$ Node($arg, parent = NodeExpr$) ▷ Find operand
4: **if** $childNode \notin ValidMathOperand$ **then** ▷ Check types
5: BuildAST($childNode$)) ▷ Recursively build up AST
6: **return** TreeTraversal($NodeExpr$) ▷ Output AST expression
7: **function** MatchCtrlVars(AST, AST_{temp})
8: $matchedSubtree \leftarrow$ CompareSubTreeStruc(AST, AST_{temp})
9: $matchedNode \leftarrow$ CompareSubTreeContent($matchedSubTree$)
10: $instrs, addrs \leftarrow$ getBinaryInfo($matchedNode$)
11: $identifiedNode \leftarrow$ IdentifySemanticVar($instrs, addrs$)
12: **return** $identifiedNode$
13: **function** IdentifyCtrlFuncAndVar(FC, AST_{tmpl}) ▷ Main function
14: $Initialize\ FL, CV$;
15: **for** $F \in FC$ **do** ▷ AST-based mapping for every candidates
16: $Initialize$ symbolic execution engine and AST generator;
17: $FoundPath \leftarrow$ RunSymExec($F_{startAddr}, F_{endAddr}, F_{avoidAddr}$)
18: $SymExpr \leftarrow$ GetSymExpression($FoundPath$)
19: $rootNode \leftarrow$ Node($SymExpr$) ▷ Create root node
20: $AST \leftarrow$ BuildAST($rootNode$) ▷ Convert symbolic expression
21: **for** $AST_{temp} \in AST_{tmpl}$ **do** ▷ Compare with templates
22: $CV_{matched} \leftarrow$ MatchCtrlVars(AST, AST_{temp})
23: **if** $CV_{matched}$ $is\ valid$ **then** ▷ Find matched node(s)
24: $FL \leftarrow FL \cup F$ ▷ Add into identified controller function list
25: $CV \leftarrow CV \cup CV_{matched}$ ▷ Add into identified CV list
26: **return** $\{FL, CV\}$

4.2 Controller Function Recognition

To find true controller functions within the candidate list, we customize the symbolic execution for each candidate by searching its target execution paths, which contain a flow of arithmetic operations in basic blocks. We extract the start address, the end address, and avoid addresses (e.g., error exception branches) for each execution path to precisely guide the symbolic execution, thus significantly reducing the execution state-space and avoiding the path explosion problem, especially for complex controller programs. Algorithm 1 shows the steps of applying symbolic execution and performing AST matching to determine the exact controller function type and variables. The input AST templates (AST_{tmpl}) are generated from the previous reference implementations for the target board. We first initialize and configure the symbolic execution engine (Line 16) with a proper binary architecture, the start address, the end address, and avoid addresses (e.g., non-covered branches). Then we run symbolic execution on each candidate (Line 17) and generate symbolic expression(s) (Line 18) as the output. We convert each symbolic expression into a symbolic abstract syntax tree (AST) (Line 19-20, 1-6), and compare the candidate AST with every AST template using model equivalence checking (Line 21-22). This checking examines the subtree contents and checks the equivalency of the symbolic arithmetic expressions from the two ASTs (Line 7). A semantically-equivalent candidate will be identified as a specific type of controller function, with concrete controller variables (e.g., PID gains, control error) recovered (Line 23-25). For example, Figure 6 depicts the AST for an identified PID function (Figure 6a) and the AST template for a high-level PID controller algorithm (Figure 6b). We determine the control variable (e.g., the P, I, and D gains) by comparing the subtree structures (e.g., the number of child nodes) and individual node contents (Line 8-9) in terms of arithmetic opcodes. The subtree traversal enables a fine-grained, modular exploration of an AST template that may be composed of subroutines, e.g., a PID controller algorithm



(a) Symbolic expression-derived AST example.



(b) Control algorithm-based AST template example.

Figure 6: Controller semantic matching and variable identification for a PID controller.

may be decomposed into three subroutine calls: P, I, and D subroutines. Hence, we identify the controller variables by collecting the symbolized memory addresses and backtracking their exact register loading/storing instructions in symbolic executions (Line 10-11).

4.3 Controller Semantic Recovery

After recognizing controller functions and variables in the previous step, we need to recover their semantics, e.g., which PID controller is the z-axis position controller. We first lift the binary firmware to LLVM bitcode, focus on controller code, detect the *program-level* controller dependencies using data flow analysis, and annotate controller semantics by comparing program-level dependencies with the RAV control model.

Annotated LLVM Bitcode Generation. DisPatch lifts the binary firmware augmented with the memory layout information (Section 4.1) into LLVM bitcode. Simultaneously, we annotate the LLVM-level instructions and functions with the addresses (in the binary firmware) of instructions, and tag them accordingly if they are common trigonometric functions (e.g., *sin*) using the binary function identifier [29], or controller functions and instructions.

Controller Dependency Model. To identify the semantics of different controllers, we need to know the controller dependencies of the given firmware. We leverage the list of configurable control parameters [16, 24, 34, 72] commonly available to support various physical features and properties as major RAVs do (e.g, PX4, ArduPilot, and DJI) and "control variable dependency graph (CVDG)" [71] to generate the controller dependency model. For example, suppose an x-axis cascading controller has a position P, velocity P, and acceleration P, I, and D control parameters mentioned in the list. This means that the x-axis cascading controller of the firmware has a P controller for the x-axis position, another P controller for the velocity, and a PID controller for the acceleration. CVDG defines the dependencies between the primitive and cascading controllers based on a generic RAV controller structure (i.e., part of the RAV control model described in Section 2).

Combining the aforementioned domain knowledge, we can derive the controller dependency model for the given firmware in Figure 7.

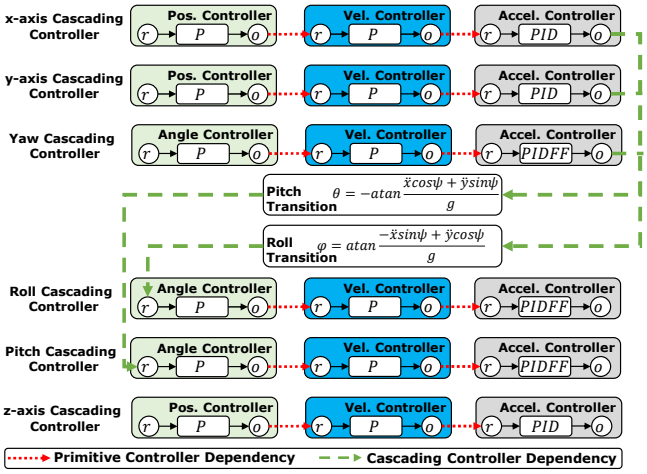


Figure 7: Visualized controller dependencies based on the controller model specification and CVDG [71].

Algorithm 2 Semantics of Controller Variables Identification.

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

```
1:  function GetPrimitiveControllerDeps(C_math, TC, F, CM)
2:      Initialize CD;                          ▷ CD: Primitive controller dependencies
3:      for c ∈ C_math do                       ▷ Check dependencies of primitive controllers
4:          d ← GetDepFromCtrls(c, C_math, F, CM)
5:          CD[c] ← GetClosestDepCtrl(c, d, TC, F, CM)
6:      return CD
7:  function GetCascadingControllers(CD, CM, TC)
8:      Initialize C_cc, chk;                   ▷ C_cc: A set of cascanding controllers
9:      for pc ∈ CD.keys do                     ▷ Check every primitive controller (pc)
10:         if pc ∉ chk then                    ▷ Skip it if pc has a dependency with another pc
11:             cc ← RecursiveTrackDeps(pc, CD)          ▷ Track data flows
12:             C_cc ← C_cc ∪ GetEachCascadingController(cc, CM, TC)
13:             chk ← chk ∪ GetPrimitiveCtrls(C_cc)      ▷ Marked checked pc
14:     return C_cc
15: function AnnotateControllerVariableSemantics(C_cc, T, CM)
16:     Initialize CVR;
17:     for cc ∈ C_cc do                        ▷ Check each cascading controller (cc)
18:         cr ← GetCascadingSemantic(cc, T, CM)
19:         for pc ∈ cc do                      ▷ Get semantics of primitive controllers in cc
20:             r_ctrl ← GetPrimitiveControllerSemantic(pc, cr, CM)]
21:             for cv ∈ pc do                  ▷ Iterate each controller variable (cv) in pc
22:                 CVR[cv] ← GetSemanticOfVar(cv, pc, CM)
23:     return CVR
24: function IdentifySemantics(C_math, TF, F, CM)          ▷ Main function
25:     T ← GetTransFuncs(TF, F)                ▷ T: Transition functions
26:     CD ← GetPrimitiveControllerDeps(C_math, T, F, CM)
27:     C_cc ← GetCascadingCtrls(CD, CM)
28:     CVR ← AnnotateControllerVariableSemantics(C_cc, T, CM)
29:     return CVR
```

Continuing on the x-axis example in Figure 7, its velocity primitive P controller has a dependency with its acceleration primitive PID controller (i.e., Vel. Controller → Accel. Controller in the x-axis cascading controller). Figure 7 also reflects controller variables, e.g., r for reference, P, I, D for control parameters in Accel. Controller. We note that we did not include the *error* and *vehicle state* (although they exist) to focus on the controller dependency illustration. Finally, Figure 7 shows the dependencies between cascading controllers via transition functions (e.g., Pitch Transition).

Controller Dependency Analysis. To recover the semantics of controller functions within the firmware, we perform inter-procedural data flow analysis on the LLVM bitcode of firmware as shown in Line 1-14, 25-26 in Algorithm 2 while comparing those data flows with the controller dependency model described in Figure 7.

To identify the primitive controller dependencies (Line 1-6, 26), DisPatch first collects both input (i.e., error variable obtained by subtracting reference and vehicle state variables) and output controller variables of each identified controller (stored in C_{math}), represented in any of "controller" boxes in Figure 7. The result of primitive controller dependencies is stored in CD (Line 26). DisPatch then chooses one primitive controller (Line 3) and performs data flow analysis from the chosen primitive controller. Specifically, DisPatch backtracks the variable update operations (e.g., multiplication) from an input variable of one controller to an output of the other controllers (Line 3-5).

While Figure 7 hints on how data flows among different primitive controllers, we need to track alias updates, e.g., backtracking data flows from the aliases of the variables found in load or store instructions (e.g., to access a heap or global variable). To identify aliases, DisPatch performs inter-procedural context-insensitive points-to analysis [90] in advance. However, this procedure may miss some aliases in practice due to missing information from compilation. To overcome this limitation, DisPatch collects instructions accessing either a global or heap variable from *this* pointer with a certain offset value, which is used as an alias identification key to heuristically identify alias accesses if the points-to analysis misses them.

During this step, DisPatch prioritizes the data flows between the controller input and output, with a small number of instructions (Line 5). The intuition is that each controller's output is tightly coupled with the input of the dependent controller; hence, light computation is typically involved. For instance, an output of a primitive controller is directly used as an input of its dependent controller (e.g., from a z-axis position controller to a z-axis velocity primitive controller).

We can differentiate a cascading controller dependency (Line 7-14, 27) from a primitive one because of the essential but unique computation to transit the degrees of freedom (Line 25). Specifically, as we described in Figure 7 and Section 2, the roll (ϕ) and pitch (θ) angles must be converted from the x, y-axis, and yaw (ψ) angular acceleration controllers with different formulas. Whereas the z-axis cascading controller does not have such a transition. Thus, we leverage this unique computation pattern (Line 12, 25) to distinguish all the different cascading controllers (Line 9-13) by checking every identified primitive controller dependency result (Line 9, 26) in the previous step (Line 1-6). For example, the inputs of the roll and pitch angles are determined by the outputs of its x, y-axis, and yaw controllers as shown in Equation 3 and 4.

$$\phi = atan((-\ddot{x}sin(\psi) + \ddot{y}cos(\psi))/g) \quad (3)$$

$$\theta = -atan((\ddot{x}cos(\psi) + \ddot{y}sin(\psi))/g) \quad (4)$$

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

Controller Function Semantic Decompilation To determine the semantics of controller functions and variables (Line 15-21, 28),

```
1 range , −, z.pos.ref=[−50.0~50.0]
2 range , roll.vel.i∉[0~0.1], roll.vel.p=[1.0~5.0]
3 ...
4 equal , z.acc.ff==0, z.acc.ff=1.0
```

Listing 1: An example of an RAV control-semantic patch DSL following our EBNF-based grammar.

DisPatch first identifies the possible semantics of the primitive controllers through checking each identified cascading controller (*cc* of C_{cc} in Line 17). DisPatch then checks the program-level *primitive controller dependency*, e.g., P controller → P controller → PID FF controller (Line 18), and checks whether there are identical dependency combinations specified in the controller dependency model (Line 19-20). For example, there are roll, pitch, and yaw cascading controllers having the dependency in Figure 7. Finally, DisPatch annotates each controller variable of the primitive controllers (Line 21-22, 28-29), such as the z-axis P parameter and reference variable.

4.4 Patch Generation

With the semantics of all controller functions and a control-semantic patch written in a high-level DSL, DisPatch is able to apply the patch to the target binary firmware automatically.

DSL Generation. To ease the development of control-semantic patches, we design a high-level domain-specific language (DSL) for RAV firmware patching. To write a patch, users first determine patch operations by running the previous fuzzing and investigation tools [63, 69, 71, 72]. Those tools help generate proper patch operations for controller variables (e.g., safe ranges for z-axis velocity control gain) without requiring much RAV background (described in Section 2). Using the results of those tools, users can write a patch in the DSL for the chosen patch operations and controller variables to protect or regulate those variables.

Specifically, our DSL follows the format consisting of <policy>, <condition>, and <operation> in order. <policy> represents the aforementioned eight patch policies (e.g., range). <condition> describes the condition where the policy is applied. <operation> defines the detailed operation for <policy>. Both <condition> and <operation> can designate the control variable to check and update. Each control variable is specified by (1) one of the 6DoFs (e.g., z-axis or roll), (2) primitive controller type (e.g., position (pos), velocity (vel)), and (3) mathematical name (e.g., P, I, D parameters or reference). We describe one example in Line 2 in Listing 1. DisPatch will add a range check on the P parameter gain of a roll rate controller (i.e., <condition> denoted as `roll.vel.i∉[0 0.1]`). If the `roll.vel.i` is not within the specified range, DisPatch limits the value as specified in "`roll.vel.p=[1.0 5.0]`" (i.e., <operation>).

Binary Patching. Finally, DisPatch translates the control-semantic patch (written in our DSL) into the corresponding binary-level patch using the list of the controller variables and controller code accessing those variables (more details in Section 5). DisPatch patches the binary firmware using the detour approach [64] due to its robustness (Section 7). One challenge is that common controller code (e.g., P control gain computation function) can have different semantics according to the callsites. For example, code at a callsite can perform any of controller's computation (e.g., a z-axis velocity or roll angular control computation). Hence, we do not know the semantics of these variables passed by until we check the callsite of each controller function. Therefore, if there are multiple callsites, DisPatch should perform *callsite-aware* instrumentation to multiplex the multiple semantics of the same controller code. Otherwise, if there is only one callsite, DisPatch directly deploys the patch to the target controller code.

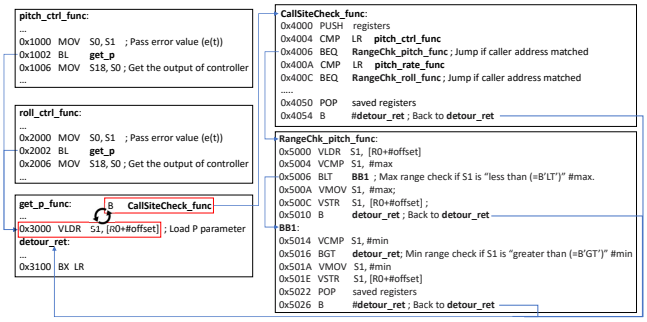


Figure 8: An example of an instrumented callsite-aware code payload to check the value range of P parameters of roll and pitch angular controllers.

For *callsite-aware* instrumentation, DisPatch checks the callsite address via either a return register or the return address pointer stored on the stack, and places a trampoline to distinguish different semantics of the target control variable when it is loaded. For instance, in Figure 8, we assume that both `roll_ctrl_func` and `pitch_ctrl_func` call `get_p_func` to compute the PID controller output. P is loaded at the same code location (`VLDR` at `0x3000`) with different semantics depending on the caller. To apply the patch to the P parameter, we replace the `VLDR` with the jump instruction to `CallSiteCheck_func` using the detour-based approach, which checks for the callsite address. In our example, `LR` stores the callsite address, and it is checked at `0x4004` and `0x400A` to determine the corresponding patch to apply. As an example of taking the `RangeChk_pitch_func` branch, once the patch (from `0x5000` to `0x5026`) is executed, the execution moves back to the next address of `0x3000` (i.e., the replaced instruction address) in `get_p_func` to continue the normal execution.

To guarantee our patch code does not affect the code other than the target controller code, we store and restore registers representing an execution state before and after our patch operation. Specifically, DisPatch inserts the following code into each patch code payload: a `PUSH` instruction (storing an execution state) at the beginning of our patch code payload (e.g., `0x4000`) and a `POP` instruction (restoring an execution state) right before jumping to the original control flow (e.g., `0x4050`).

5 IMPLEMENTATION

We modified RetDec [12] to tag binary instruction addresses in the firmware within their corresponding LLVM instructions in DWARF [25] format, as well as common trigonometric functions and identified controller functions and instructions into the corresponding LLVM objects. We note that common trigonometric functions (e.g., *sin* and *cos*) are typically non-inlined functions as discussed in Section 7. Hence, we use the signature-based function matching technique [29] to identify those functions from the binary firmware without missing any of them.

We observed that RetDec failed to identify all the argument and return variables, leading to missing inter-procedural data flows of controller dependencies. To complement such missing information, DisPatch tracks read and write operations on variables (e.g., registers or stack variable in the firmware) as typical binary analyses do [28, 89]. If either an argument of a caller or a return variable of a callee is read before it is written, we consider such variables as passed from either a caller or callee. To reduce false positives, we choose those argument/return registers and stack variables based on the calling convention in the ARM architecture (e.g., argument and return values are held in `R0`-`R3`).

We implement the controller function candidate filtering (Section 4.1) as a plugin for IDA Pro 7.5 [28]. We also implement a firmware augmentation script to add memory layout information on a binary blob using elftools [36]. We use the symbolic execution engine in angr [89] to generate the symbolic expression for controller functions' outcomes (Section 4.2) and develop the semantic matching module to perform AST conversion, invalid node pruning, arithmetic operator examination and optimization. We implement the controller semantic recovery (Section 4.3) on top of the SVF 1.8 static analysis tool [90] for points-to analysis with LLVM 9.0 [77]. Finally, we implement our binary patching (Section 4.4) based on detour-based binary rewriting technique as a plugin for IDA Pro 7.5. The number of lines of code (LoC) of our whole system is 10,582 (5,356 lines of C++ and 5,226 lines of Python).

6 EVALUATION

We evaluate DisPatch against real-world RAV firmware to answer the following key questions:

- **Q1**: How accurate is the controller function and variable identification and decompilation? (Section 6.1)
- **Q2**: How practical is DisPatch for reverse engineering and patching suboptmial or buggy controller design and implementation? (Section 6.2)
- **Q3**: How much overhead is imposed by the patch of DisPatch on RAV operations? (Section 6.3)

Experimental Setup. We evaluated the firmware from two commercial quadcopters, 3DR IRIS+ [7] and MantisQ [30]. The two instances of RAV firmware are based upon ArduPilot [15] and PX4 [35], which are the most popular control software [4, 42, 61], adapted and customized by many RAV vendors [10, 11] including Yuneec, Cheerson, and CUAV [1, 23, 39]. Furthermore, ArduPilot supports various RAV with other three *physical specifications* (e.g., plane, helicopter, and submarine), which are all covered in our evaluation. We note that many RAV software instances provide different firmware for each physical specification if the RAV physical specifications have non-negligible differences requiring controller code modification. PX4 is developed based on a different code base from ArduPilot.

We obtained RAV firmware in the case where the source code or binary firmware was not available. For the ArduPilot-based firmware, we connected a GDB debugger (e.g., black magic probe [19]) to a GDB port or soldered a debugger to microcontroller's debugging pins, e.g., SWDJ-DP as introduced in MCU chip's specifications, such as SVD [21]. This way, we obtained the firmware from the 3DR IRIS+ quadcopter [7]. For firmware from MantisQ [30], we encountered a read-out lock mechanism preventing direct firmware extraction. We exploited a software vulnerability in NuttX [14], used in PX4, to read firmware from the flash memory directly using the shell interface. We have reported this vulnerability to the vendor (Yuneec) and the open source communities (NuttX for RTOS and PX4 for RAVs). So far, NuttX and PX4 communities have patched the reported vulnerability[1].

We evaluate DisPatch with Pixhawk [9] for ArduPilot-based firmware and MantisQ's board (using an STM32F765 microcontroller unit (MCU)) as our target hardware boards. Pixhawk is equipped with a 192KB SRAM and a 2MB flash memory; MantisQ's board is equipped with a 512KB SRAM and a 2MB flash memory. Both boards are equipped with multiple sensors (e.g., a GPS and a gyroscope).

Table 1: Accuracy of PID controller function identification of the ArduPilot-based firmware with four different physical specifications (3DR IRIS+ copter, ArduPilot helicopter, ArduPilot plane, and ArduPilot submarine) and PX4-based MantisQ copter firmware in each step. We identify the ground truth from the list of configurable control parameters (Section 4) which tells the information on the deployed controllers in the firmware. P: P controller functions, I: I controllers functions, D: D controller functions, FF: FF controller functions, PID: PID controller functions, PID FF: PID FF controllers, FP: False positive rate, FN: False negative rate.

Target	Processing Step	# of P	# of I	# of D	# of FF	# of PID	# of PID FF	Total	FP (%)	FN (%)
3DR IRIS+ Copter	Candidate Filter	26	10	26	26	3	1	40	75.0%	0%
	Controller Recognition	5	2	2	1	0	0	10	0%	0%
	Ground Truth	5	2	2	1	0	0	10	-	-
ArduPilot Helicopter	Candidate Filter	27	11	27	27	3	1	42	76.2%	0%
	Controller Recognition	5	2	2	1	0	0	10	0%	0%
	Ground Truth	5	2	2	1	0	0	10	-	-
ArduPilot Plane	Candidate Filter	27	11	27	27	4	2	44	77.3%	0%
	Controller Recognition	5	2	2	1	0	0	10	0%	0%
	Ground Truth	5	2	2	1	0	0	10	-	-
ArduPilot Submarine	Candidate Filter	26	7	26	26	3	1	37	73.0%	0%
	Controller Recognition	5	2	2	1	0	0	10	0%	0%
	Ground Truth	5	2	2	1	0	0	10	-	-
MantisQ Copter	Candidate Filter	14	0	14	14	5	10	29	86.2%	3.4%*
	Controller Recognition	1	0	0	0	1	1	3	0%	25.0%*
	Ground Truth	2†	0	0	0	1	1	4	-	-

* Because we do not know the ground truth of one P controller location, we conservatively consider we miss that function even in the static analysis step.

† We expect there is a single controller for x, y, z-axes. Hence, we put one more controller function as the ground truth.

6.1 Controller Decompilation Accuracy

There are two steps in DisPatch to identify controller functions and variables: (1) candidate filtering using static analysis (Section 4.1), and (2) controller recognition using AST-based model checking and matching (Section 4.2). We show the results for each step in Table 1. In our experiment, we found not only P controller functions, PID controllers, and PID FF controllers, but also sub-controller functions of PID and PID FF controller functions as described in Table 1.

As shown in Table 1, candidate filtering cannot distinguish among P, D, FF controllers due to their similar mathematical instruction patterns. The total number of (sub-)controller function candidates are 40 functions for 3DR IRIS+ copter, 42 functions for ArduPilot helicopter, 44 functions for ArduPilot plane, 37 functions for ArduPilot submarine, and 29 functions for MantisQ, while the ground truth is

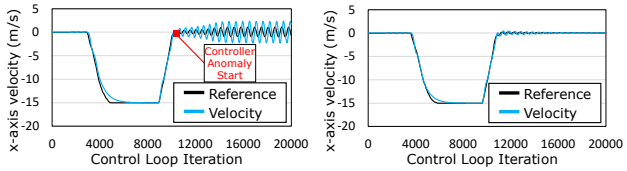
[1] https://github.com/apache/incubator-nuttx-apps/commit/6f4b133 for NuttX and https://github.com/PX4/PX4-Autopilot/pull/17264 for PX4.

Table 2: The semantics of mathematical PID controller functions and their controller variables for 3DR IRIS+ copter firmware, three ArduPilot-based firmwares of different physical specifications (helicopter, plane, and submarine), and MantisQ copter firmware. Ctrl.: Controller function, ✓: semantic of an identified controller used, ✗: semantic of a missing controller used, CV: controller variable, Accuracy: accuracy of controller variable identification.

Target Firmware	Controller Semantic	P Ctrl.	PID Ctrl.	PID FF Ctrl.	# of CVs	# of Identified CVs	Accuracy
3DR IRIS+ Copter	x, y-axis position	✓			7	7	100%
	x, y-axis velocity		✓		9	9	100%
	z-axis position	✓			4	4	100%
	z-axis velocity	✓			4	4	100%
	z-axis acceleration		✓		6	6	100%
	Roll angle	✓			4	4	100%
	Roll angular rate			✓	7	7	100%
	Pitch angle	✓			4	4	100%
	Pitch angular rate			✓	7	7	100%
	Yaw angle	✓			4	4	100%
	Yaw angular rate			✓	7	7	100%
	Total # of CVs	-	-	-	63	63	100%
ArduPilot For The Other Three Physical Specifications	x, y-axis position	✓			7	7	100%
	x, y-axis velocity		✓		9	9	100%
	z-axis position	✓			4	4	100%
	z-axis velocity	✓			4	4	100%
	z-axis acceleration		✓		6	6	100%
	Roll angle	✓			4	4	100%
	Roll angular rate			✓	7	7	100%
	Pitch angle	✓			4	4	100%
	Pitch angular rate			✓	7	7	100%
	Yaw angle	✓			4	4	100%
	Yaw angular rate			✓	7	7	100%
	Total # of CVs	-	-	-	63	63	100%
MantisQ Copter	x, y-axis position	✗			7	0	0%
	z-axis position	✗			4	0	0%
	x, y-axis velocity		✓		9	9	100%
	z-axis velocity		✓		4	4	100%
	Roll angle	✓			4	4	100%
	Roll angular rate			✓	7	7	100%
	Pitch angle	✓			4	4	100%
	Pitch angular rate			✓	7	7	100%
	Yaw angle	✓			4	4	100%
	Yaw angular rate			✓	7	7	100%
	Total # of CVs	-	-	-	57	46	80.7%

just either 10 or 4. Compared to candidate filtering, controller recognition can almost distinguish all of the different (sub-)controller candidate functions. We manage to identify all the 10 controllers within 3DR IRIS+ copter and ArduPilot-based helicopter, plane, and submarine firmwares and 3 out of 4 controllers within MantisQ firmware. The false negatives in MantisQ firmware is caused by a missing P controller, as DisPatch could not find the P controller for x, y, z-axes positions. In fact, we manually found one potential P controller in MantisQ firmware. However, we could not verify whether this is indeed the controller due to the drastic difference between the firmware implementation and the reference control formula.

Building upon the result of PID (sub-)controller identification, DisPatch recovers controller semantics (Section 4.3). Table 2 summarizes the results for (1) which variant of PID controllers are used for each controller semantic (e.g., z-axis velocity), and (2) the number of controller variables (including P, I, D, and FF control parameters, the reference, error, and vehicle state) of different controllers (Section 2). Specifically, 3DR IRIS+ copter firmware and ArduPilot open-source firmwares have one individual PID controller to control z-axis acceleration with P, I, and D control parameters while having reference, error, and vehicle states. On the other hand, MantisQ firmware has one individual PID controller to control z-axis velocity



(a) Attack implication on the hexacopter. **(b) Attack implication on the octacopter.**

Figure 9: Case I: The implications of manipulating P, I, and D of the x, y-axis velocity controller.

with P, I, and D control parameters including reference, error, and vehicle states. Both controllers have six controller variables. The other PID controller is customized to control two dimensional (x, y-axis) accelerations containing respectively one more reference, error, and vehicle states to control both x and y-axis simultaneously. Hence, it has nine controller variables as shown in Table 2.

As a result, DisPatch finds a total of 6 P controllers, 2 PID controllers, and 3 PID FF controllers in 3DR IRIS+ copter and ArduPilot-based firmware (for helicopter, plane, and submarine) with 100% accuracy. DisPatch finds a total of 5 P controllers, 2 PID controllers without any missing controllers and 3 PID FF controllers in MantisQ firmware with 80.7% accuracy due to missing x, y, z-axes P controllers. Furthermore, In 3DR IRIS+ copter and ArduPilot-based firmware, DisPatch identifies all of those controllers and 63 controller variables. Whereas DisPatch finds 46 controller variables out of 57 controller variables in MantisQ firmware.

6.2 Case Studies

We show two case studies as applications for controller design and implementation improvement through reverse engineering and patching real-world control-semantic bugs within RAV firmware. In consideration of safety regulations, we ran long-distance testing flights using a software-in-the-loop (SITL) simulator [37, 74] to inspect the impacts of control-semantic bugs. We note that SITL simulators have high accuracy and fidelity to emulate the physical world under the physics laws; hence SITL simulators are commonly used in RAV testing by academia [15, 35, 71, 72] and industry [13, 17, 33]. On the other hand, we applied our patch to the 3DR IRIS+ and MantisQ copter firmware and verified whether our patch worked correctly using real RAVs running the patched firmware. For instance, we performed indoor tests to check operations and inspected control variable data in memory.

6.2.1 Constraining Control Parameter Range. Following up on the motivating example introduced in Section 3, we show the controller states of both the hexacopter and octacopter in Figure 9. In this example, the RAV is supposed to execute the moving command (i.e., moving in the west direction at 15 m/s) and hover at the destination. As shown in Figure 9b, the octacopter flies stably, while the hexacopter shows severe and persistent controller anomaly with the same control parameter setting, as shown in Figure 9a. Specifically, from 10,655 Iterations (i.e., destination arrival and hovering in the sky), both reference ($\dot{r}_x(t)$) and vehicle state ($\dot{x}_x(t)$) of the x, y-axis velocity controller cannot track with each other.

```
1 range , -, xy.pos.p=[1.0~1.5]
2 range , -, xy.vel.p=[1.5~4.5]
3 range , -, xy.vel.i=[0.2~1.0]
4 range , -, xy.vel.d=[0.2~1.0]
```

Listing 2: An RAV control-semantic patch DSL for the Case Study I. Four parameters are patched.

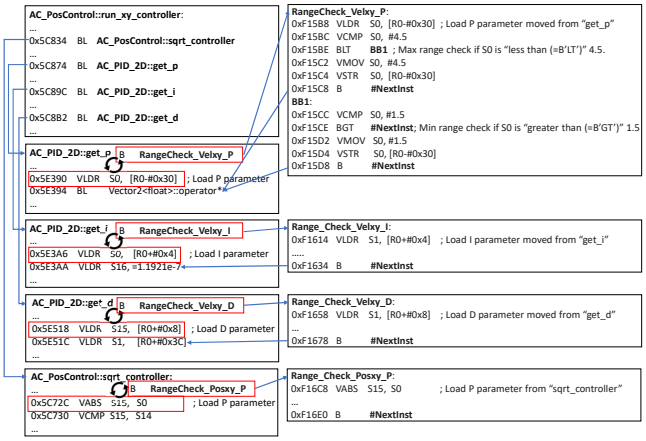


Figure 10: An example of the protection logics for PSC_VELXY_P, PSC_VELXY_I, PSC_VELXY_D, and PSC_POSXY_P parameters.

The root cause is that the ArduPilot-based RAV firmware for hexacopter and octacopter are identical, sharing the same loose check for the following parameters: the P (PSC_VELXY_P), I (PSC_VELXY_I), and D (PSC_VELXY_D) of the x, y-axis velocity controller, and the P (PSC_POSXY_P) of the x, y-axis position controller. Therefore, we decide to apply a specialized patch for the above four parameters to the hexacopter firmware to fix this vulnerability.

To patch such a control-semantic bug, we could use testing systems [63, 69, 71, 72] to determine the proper parameter range tightly coupled with the RAV physical feature and property. Users can reduce the default value range of PSC_VELXY_P or even make it fixed with DisPatch. In our example, we write a patch (i.e., Listing 2) to reduce the default value range of PSC_VELXY_P from [0.1,6.0] to [1.5,4.5] described as "range, -, xy.vel.p=[1.54.5]". We apply similar range reductions to the other three parameters.

Assuming DisPatch has been applied to the target firmware earlier and has all the information of controller functions and variables, it deploys the patch at the binary level as shown in Figure 10, which enforces range checks on the four control parameters in the different (sub-)controller functions by making the firmware call the four protection functions (RangeCheck_Velxy_P, RangeCheck_Velxy_I, RangeCheck_Velxy_D, and RangeCheck_Posxy_P). For example, DisPatch reduces the value range of PSC_VELXY_P within [1.5,4.5] by making the firmware jump to the RangeCheck_Velxy_P detour function. We ran the patched firmware and confirmed the stability of the hexacopter's operation at runtime.

6.2.2 Enforcing Inter-Dependencies. As shown in the previous work [72], the control parameters have inter-dependencies among them. If one parameter is changed, the original valid range of other parameters might not be valid anymore. For instance, control parameters MC_ROLLRATE_P and MC_ROLLRATE_I in PX4-based firmware interact with each other with default values 0.15 and 0.05 respectively. As shown in Figure 11, let us assume that MC_ROLLRATE_P is changed into the low value (i.e., less than 0.02) within the safe value range, while the value of MC_ROLLRATE_I is its default value (i.e., 0.05). Because of the inter-dependency between MC_ROLLRATE_I and MC_ROLLRATE_P, the RAV started to show severe controller anomalies, leading to the crash.

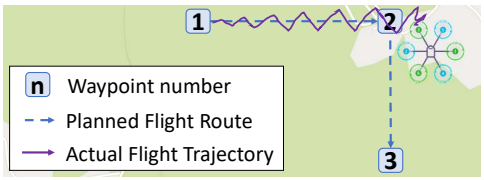


Figure 11: Case II: An RAV crash caused by the low MC_ROLLRATE_P with the default value of MC_ROLLRATE_I.

```
1 range , roll.vel.p<=[0.02], roll.vel.i=[0.0~0.01]
2 range , roll.vel.p>[0.02] , roll.vel.i=0.05
```

Listing 3: An RAV control-semantic patch DSL for the Case Study II to put the range check into MC_ROLLRATE_I according to the value of MC_ROLLRATE_P.

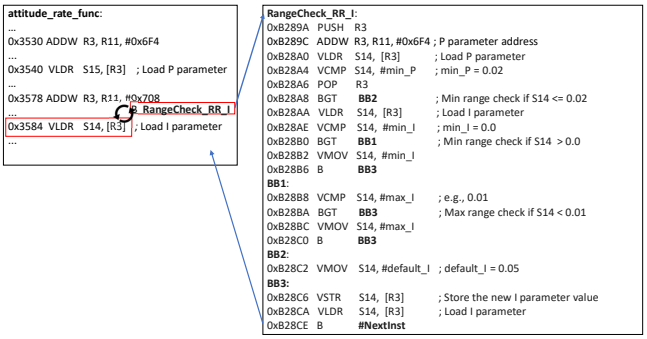


Figure 12: An example of the protection logic for MC_ROLLRATE_I according to the value of MC_ROLLRATE_P.

6.3 Performance Overhead

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

Table 3: Space overhead introduced by DisPatch. We instrumented all of the identified controllers in all of our target 3DR IRIS+ and three ArduPilot-based firmwares of different physical specifications (helicopter, plane, and submarine), and MantisQ firmwares.

RAV Control Model	Original Firmware Size	Instrumented Firmware Size	Space Overhead
3DR IRIS+ Copter	949KB	954KB	0.55%
ArduPilot Helicopter	936KB	941KB	0.56%
ArduPilot Plane	936KB	941KB	0.56%
ArduPilot Submarine	822KB	827KB	0.64%
MantisQ Copter	1,080KB	1,084KB	0.37%
Average	945KB	950KB	0.53%

Space Overhead. We measure DisPatch's space overhead on 3DR IRIS+ copter firmware and ArduPilot-based firmware with the three different RAV physical specifications (i.e., helicopter, plane, and submarine) and MantisQ firmware. As shown in Table 3, the average space overhead is 0.53%, negligible and acceptable for real-world deployments, and our patched RAV firmware can easily fit within the size-constrained flash memory of an RAV.

Runtime Overhead. We measured the runtime overhead of patched 3DR IRIS+ copter firmware and three instances of ArduPilot firmware for all the three physical specifications. However, we saw the runtime overhead in the 3DR IRIS+ copter firmware similar to the runtime overheads in three instances of ArduPilot-based firmware of the three physical specifications. Hence, we present the detailed and representative result for the 3DR IRIS+ copter firmware. We measured the runtime overhead of 48 real-time tasks running within the 3DR IRIS+ copter firmware. The execution frequency of each real-time task ranges from 0.1Hz to 400Hz. Figure 13 summarizes (1) the execution time without patches, (2) the execution time with patches, and (3) the soft real-time deadlines. Compared with the task execution time without patches, we observed 1.46% runtime overhead on average ranging from -8.1% to 9.7% in the copter firmware. The negative overhead should be caused by fluctuations of non-deterministic inputs (e.g., GPS and GCS communications).

The deviations in runtime overhead occurred more frequently for the tasks with small execution time (e.g., 7.1 µs for `update_optical_flow`, and 7.1 µs for `arm_motors_check` on unpatched firmware) since they are more sensitive to the patching overhead. Among all the tasks, only `one_hz_loop` missed the soft real-time deadline. We found that this task violated the soft real-time deadline even without the patches. We also found that controller-related tasks such as `update_altitude` and `run_nav_updates` showed relatively higher runtime overhead of 6.8% and 8.7% due to the presence of more controller variables instrumented by DisPatch. Overall, we observed 1.48% runtime overhead on average ranging from 1.41% to 1.55% for all ArduPilot-based firmware.

We note that we did not measure the runtime overhead of MantisQ firmware because it is infeasible to find locations for the performance measurement function insertion in the stripped MantisQ firmware. However, we did not observe notable runtime overhead when we conducted a series of typical operations, e.g., flying back and forth between two waypoints. We anticipate the runtime overhead to be negligible without affecting the performance in consideration of the low runtime overhead found in ArduPilot-based firmware.

7 DISCUSSION

Trigonometric Inline Function Detection. DisPatch requires identifying common trigonometric functions (e.g., *sin*) to locate controller components. However, if these functions are inline functions in the firmware, their identification can be challenging. Fortunately, we found that all the trigonometric functions are not inlined functions and they are provided as parts of a compiler's standard libraries (e.g., `libm`). Based on our observation, the reason is that a firmware compiler makes firmware more compact because embedded systems have limited memory space. Specifically, common trigonometric functions are called at multiple locations in the RAV firmware. Furthermore, the sizes of such functions are not small because those functions involve a non-trivial number of mathematical operations. Therefore, a compiler usually does not make such trigonometric functions inlined to reduce memory usage.

Binary Patching Risks. Binary patching can cause the failure of the patched program execution due to the incorrect disassembly results. However, modern disassembly shows the very high accuracy (e.g., IDA Pro which we used shows the 99.4% accuracy [65]). Thanks to such a high accuracy, we believe the possibility of incorrect disassembly at our target patch locations is extremely low. Furthermore, our chosen patching approach, the detour-based approach [64], is safe as long as the target location is correctly disassembled. This is because the detour-based approach and our target instrumentation location does not require changing unrelated parts, such as other pieces of existing code, control flows, the memory layout, pointer variables or the processor's state (e.g., interrupt flags), except for two things: appending the patch payload and adding two jump operations to jump to the patch code payload and return from this payload. This is the reason why many existing binary instrumentation systems leverage the detour-based approach [41, 78, 98].

Firmware Obfuscation and Encryption. We assume that the firmware can be disassembled before our technique is applied. This means we could not handle firmware whose actual binary instructions are only available during the runtime due to packing, obfuscation, or encryption. Based on our observation, such protection logics are not popular in practice for RAV firmware due to limited computing resources [67, 70, 88] of RAVs.

Firmware Patching Prevention. RAV vendors can protect their firmware against our patching operation. One way is to apply a firmware integrity check to prevent firmware patching. Additionally, vendors can keep the memory layout and parameter specification confidential to prevent RAV firmware patching. Fortunately, memory layout (e.g., DJI Phantom 4's board [38]) and parameter specification [16, 24, 34] (e.g., DJI RAVs' parameter specifications [24]) information for many off-the-shelf RAVs are often publicly available or obtainable through the standard RAV protocol (e.g., MAVLink [31]).

Limitations in Architecture and Language Support. Our prototype of DisPatch currently supports firmware written in C/C++ running on ARM Cortex-M processor considering that C/C++ and ARM architecture are dominating in embedded systems [70] including RAVs [15, 22, 35]. DisPatch can be further extended to support other architectures, e.g., x86, and other programming languages that are compiled into native instructions such as Rust and Go.

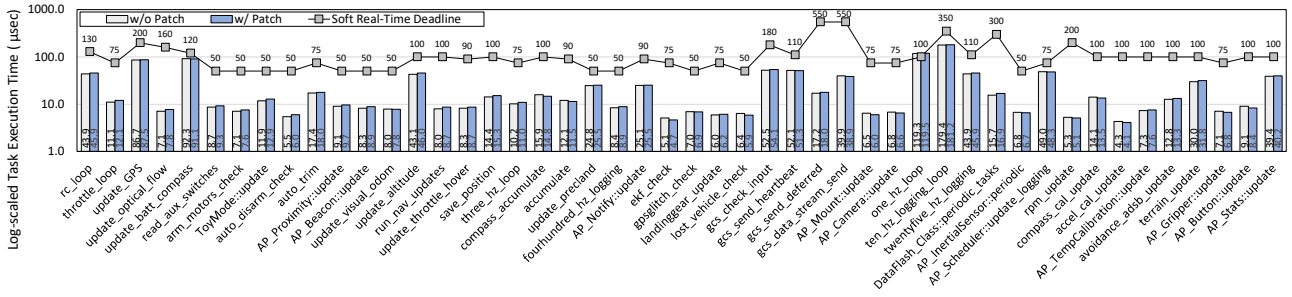


Figure 13: Runtime overhead of the 3DR IRIS+ copter firmware. All of its identified controller variables are instrumented: Average execution time of 48 soft real-time tasks in log scale with and without DisPatch's patch. Both unpatched and patched firmware meet the soft real-time deadlines except for the one_hz_loop task.

8 RELATED WORK

Disassembly, Decompilation, and Function Identification. Traditional function identification frameworks [20, 28, 43, 44, 94] cannot identify *semantics* of functions (e.g., a z-axis velocity controller function). To identify semantics, some approaches rely on static analysis, binary code signatures [29], and control flow graphs [40, 54, 57]). Others leverage dynamic analysis to record and analyze execution traces [53, 60, 68]. Additionally, there have been several works leveraging machine learning techniques [52, 91, 100]. However, none of them are robust enough to find controller functions, especially when such functions have multiple variants.

To help analysts understand how binary code works, there have been decompilation techniques [12, 18, 26, 27, 45, 59, 76, 86, 87, 101, 102], recovering the high-level language syntactics of binary code, such as variable type and control flows. Also, there are decompilation techniques targeting specific applications, such as machine learning models [99]. However, they cannot recover any control semantics that are the target of DisPatch to recover without understanding control models.

Two frameworks can identify the controller function and variable using static binary signatures [66] or dynamic symbolic execution [92]. However, none of them can find the customized PID controller functions (e.g., P or PID FF controllers), let alone their semantics (e.g., yaw angular controller) as shown in Section 3.

Controller Design and Implementation Improvement. Researchers have proposed a large number of solutions for improving controller design and implementation of RAVs. Some works apply filters, such as the extended Kalman filters [5, 62], for controller performance improvement. However, they cannot fix incorrect controller design and implementation or control-semantic bugs that go beyond their filtering capabilities as shown in the previous works [71, 72, 82]. Several efforts leverage an RAV control model [56, 82, 93] or machine learning models [51, 55, 85] to detect the controller anomaly. However, they focus on controller anomaly *detection* rather than the elimination of anomalies led by incorrect or vulnerable controller design and implementation. Recent work [55, 82, 93] can defend against controller anomaly (e.g., severe sensor noise or control-semantic bug exploitation). However, they can maintain controller states for a limited time or mitigate controller anomaly but not fully defend against controller anomaly.

Binary Rewriting. There have been a large body of works to patch the binary programs in general [47, 49, 50, 64, 70, 78–80, 95–97]. Although it is possible to use those techniques patching binary firmware, they fail to answer the fundamental questions in an automated fashion, such as (i) where the patch should be applied, (ii) what patch should be applied, and (iii) how easy it is for an operator to define and apply patches. Therefore, they require a fair amount of human efforts to patch RAV firmware. In contrast, DisPatch provides a user-friendly DSL interface to help users readily rewrite RAV firmware.

9 CONCLUSION

DisPatch is the first end-to-end RAV firmware reverse engineering and patching framework, allowing end users to write human-readable patches, reverse engineer and patch controller design and implementation within firmware with reasonable flexibility, high accuracy, and negligible overhead.

ACKNOWLEDGMENT

We thank our shepherd, Lin Zhong, and the anonymous reviewers for their valuable comments and suggestions. This work was supported in part by ONR under grant N00014-21-1-2328, DARPA under contract N6600120C4031, National Science Foundation (NSF)'s Secure and Trustworthy Cyberspace (SaTC) and Cyber-Physical Systems (CPS) programs, and the Department of Energy under Award Number DE-OE0000780, Cyber Resilient Energy Delivery Consortium (CREDC). Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] Cheerson, 2015. http://cx.tmkcode.com.
[2] Drone crash at white house reveals security risks, 2015. https://www.usatoday.com/story/news/2015/01/26/drone-crash-secret-service-faa/22352857.
[3] *Hijacking drones with a MAVLink exploit*, 2015. http://diydrones.com/profiles/blogs/hijacking-quadcopters-with-a-mavlink-exploit.
[4] *ArduPilot :: About*, 2016. https://ardupilot.org/about.
[5] *Inertial Navigation Estimation Library*, 2016. https://github.com/priseborough/InertialNav.
[6] Here's what happens when a drone falls on your head, 2017. http://fortune.com/2017/04/29/drone-faa-head-crash-study.
[7] *Iris – The Ready to Fly UAV Quadcopter*, 2018. http://www.arducopter.co.uk/iris-quadcopter-uav.html.

[8] *US military equipped with tiny spy drones*, 2019. https://www.zdnet.com/article/us-military-equipped-with-tiny-spy-drones.

[9] Pixhawk, 2020. https://docs.px4.io/v1.9.0/en/flight_controller/pixhawk.html.

[10] Ready-to-use & Easy-to-buy Vehicles âĂŤ Copter Documentation 2020, 2020. https://px4.io/ecosystem/commercial-systems.

[11] Commercial Systems – PX4, 2021. https://ardupilot.org/copter/docs/common-rtf.html.

[12] *RetDec: a retargetable machine-code decompiler based on LLVM*, 2021. https://github.com/avast/retdec.

[13] *3DR*, 2022. https://3dr.com.

[14] *Apache NuttX Apache NuttX is a mature, real-time embedded operating system*, 2022. https://nuttx.apache.org.

[15] *ArduPilot*, 2022. http://ardupilot.org.

[16] *ArduPilot Parameter List*, 2022. http://ardupilot.org/copter/docs/parameters.html.

[17] *Auterion – The drone software platform built for enterprise*, 2022. https://auterion.com.

[18] Binary ninja, 2022. https://binary.ninja.

[19] Black magic probe, 2022. https://github.com/blacksphere/blackmagic/wiki.

[20] *Capstone*, 2022. http://www.capstone-engine.org.

[21] *CMSIS System View Description*, 2022. http://www.keil.com/pack/doc/CMSIS/SVD/html/index.html.

[22] *Crazyflie 2.1*, 2022. https://www.bitcraze.io/products/crazyflie-2-1.

[23] Cuav raefly, 2022. https://store.cuav.net/shop/raefly.

[24] *DJI Parameter Index*, 2022. https://dji.retroroms.info/howto/parameterindex.

[25] The dwarf debugging standard, 2022. http://dwarfstd.org.

[26] Ghidra, 2022. https://ghidra-sre.org.

[27] Hex rays, 2022. https://hex-rays.com/decompiler.

[28] *Hex-Rays, IDA Pro disassembler*, 2022. http://www.hex-rays.com/products/ida.

[29] *IDA F.L.I.R.T. Technology: In-Depth — Hex Rays*, 2022. https://www.hex-rays.com/products/ida/tech/flirt/in_depth.

[30] *Mantis Q*, 2022. https://us.yuneec.com/mantis-q-robust-travel-drone-with-voice-control.

[31] *MAVLink Micro Air Vehicle Communication Protocol*, 2022. https://mavlink.io.

[32] *MAVLink Parameter Protocol*, 2022. https://mavlink.io/en/services/parameter.html.

[33] *Parrot Bebop2*, 2022. https://support.parrot.com/global/support/products/parrot-bebop-2.

[34] *PX4 Parameter List*, 2022. https://dev.px4.io/en/advanced/parameter_reference.html.

[35] *PX4 Pro Open Source Autopilot - Open Source for Drones*, 2022. http://px4.io.

[36] *pyelftools*, 2022. https://github.com/eliben/pyelftools.

[37] *SITL Simulator (ArduPilot Developer Team)*, 2022. http://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html.

[38] *WM330 Flight Controller board*, 2022. https://github.com/o-gs/dji-firmware-tools/wiki/WM330-Flight-Controller-board.

[39] *Yuneec*, 2022. https://us.yuneec.com.

[40] *Zynamics — Bindiff*, 2022. https://www.zynamics.com/software.html.

[41] Tigist Abera, N Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-flat: control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

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

[43] Erick Bauman, Zhiqiang Lin, and Kevin W Hamlen. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *Proceedings of the 27th Annual Symposium on Network and Distributed System Security (NDSS)*, 2018.

[44] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. Bap: A binary analysis platform. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2011.

[45] David Brumley, JongHyup Lee, Edward J Schwartz, and Maverick Woo. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Proceedings of 22nd USENIX Security Symposium (USENIX Security)*, 2014.

[46] Gino Brunner, Bence Szebedy, Simon Tanner, and Roger Wattenhofer. The urban last mile problem: Autonomous drone delivery to your balcony. In *2019 International Conference on Unmanned Aircraft Systems (ICUAS)*. IEEE, 2019.

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

[48] Esteban Cano, Ryan Horton, Chase Liljegren, and Duke M Bulanon. Comparison of small unmanned aerial vehicles performance using image processing. 2017.

[49] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. Bistro: Binary component extraction and embedding for software security applications. In *Proceedings of the 18th European Symposium on Research in Computer Security (ESORICS)*, 2013.

[50] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (IEEE S&P)*, 2020.

[51] Aolin Ding, Praveen Murthy, Luis Garcia, Pengfei Sun, Matthew Chan, and Saman Zonouz. Mini-me, you complete me! data-driven drone security via dnn-based approximate computing. In *24th International Symposium on Research in Attacks, Intrusions and Defenses*, 2021.

[52] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (IEEE S&P)*, 2019.

[53] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *Proceedings of 23rd USENIX Security Symposium (USENIX Security)*, 2014.

[54] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovre: Efficient cross-architecture identification of bugs in binary code. In *Proceedings of the 25th Annual Symposium on Network and Distributed System Security (NDSS)*, 2016.

[55] Fan Fei, Zhan Tu, D Xu, and Xinyan Deng. Learn-to-recover: Retrofitting uavs with reinforcement learning-assisted flight control under cyberphysical attacks. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2020.

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

[57] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[58] Azade Fotouhi, Ming Ding, and Mahbub Hassan. Understanding autonomous drone maneuverability for internet of things applications. In *2017 IEEE 18th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*. IEEE, 2017.

[59] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. Coda: An end-to-end neural program decompiler. *Advances in Neural Information Processing Systems*, 32:3708–3719, 2019.

[60] Debin Gao, Michael K Reiter, and Dawn Song. Binhunt: Automatically finding semantic differences in binary programs. In *International Conference on Information and Communications Security (ICICS)*. Springer, 2008.

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

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

[63] Ruidong Han, Chao Yang, Siqi Ma, JiangFeng Ma, Cong Sun, Juanru Li, and Elisa Bertino. Control parameters considered harmful: Detecting range specification bugs in drone configuration modules via learning-guided search. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*, 2022.

[64] Galen Hunt and Doug Brubacher. Detours: Binary interception of win32 functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, 1999.

[65] Muhui Jiang, Yajin Zhou, Xiapu Luo, Ruoyu Wang, Yang Liu, and Kui Ren. An empirical study on arm disassembly tools. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2020.

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

[67] M Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. Speculative disassembly of binary code. In *Proceedings of the 2016 IEEE International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2016.

[68] Dohyeong Kim, William N Sumner, Xiangyu Zhang, Dongyan Xu, and Hira Agrawal. Reuse-oriented reverse engineering of functional components from x86 binaries. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014.

[69] Hyungsub Kim, Muslum Ozgur Ozmen, Antonio Bianchi, Z Berkay Celik, and Dongyan Xu. Pgfuzz: Policy-guided fuzzing for robotic vehicles. In *Proceedings of the 30th Annual Symposium on Network and Distributed System Security (NDSS)*, 2021.

[70] Taegyu Kim, Chung Hwan Kim, Hongjun Choi, Yonghwi Kwon, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. Revarm: A platform-agnostic arm binary rewriter for security applications. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC)*, 2017.

[71] Taegyu Kim, Chung Hwan Kim, Altay Ozen, Fan Fei, Zhan Tu, Xiangyu Zhang, Xinyan Deng, Dave (Jing) Tian, , and Dongyan Xu. From control model to program: Investigating robotic aerial vehicle accidents with mayday. In *Proceedings of 29th USENIX Security Symposium (USENIX Security)*, 2020.

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

[73] Taegyu Kim, Vireshwar Kumar, Junghwan Rhee, Jizhou Chen, Kyungtae Kim, Chung Hwan Kim, Dongyan Xu, and Dave (Jing) Tian. Pasan: Detecting peripheral access concurrency bugs within bare-metal embedded applications. In *Proceedings of 30th USENIX Security Symposium (USENIX Security)*, 2021.

[74] Nathan P Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2004.

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

[76] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Dire: A neural approach to decompiled identifier naming. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[77] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2004.

[78] Michael A Laurenzano, Mustafa M Tikir, Laura Carrington, and Allan Snavely. Pebil: Efficient static binary instrumentation for linux. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2010.

[79] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, 2005.

[80] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[81] Goh Ming Qian, Dwi Pebrianti, Luhur Bayuaji, Nor Rul Hasma Abdullah, Mahfuzah Mustafa, Mohammad Syafrullah, and Indra Riyanto. Waypoint navigation of quad-rotor mav using fuzzy-pid control. In *Proceedings of Symposium on Intelligent Manufacturing & Mechatronics (IMM)*, 2018.

[82] Raul Quinonez, Jairo Giraldo, Luis Salazar, and Erick Bauman. Savior: Securing autonomous vehicles with robust physical invariants. In *Proceedings of 29th USENIX Security Symposium (USENIX Security)*, 2020.

[83] Agoston Restas et al. Drone applications for supporting disaster management. *World Journal of Engineering and Technology*, 3(03):316, 2015.

[84] Nils Rodday. Hacking a professional drone. *Blackhat ASIA*, 2016.

[85] Ihab Samy, Ian Postlethwaite, and Dawei Gu. Neural network based sensor validation scheme demonstrated on an unmanned air vehicle (uav) model. In *Proceedings of the 47th IEEE Conference on Decision and Control (CDC)*, 2008.

[86] Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta, and Alexey Loginov. Evolving exact decompilation. In *Proceedings of the Workshop on Binary Analysis Research (BAR)*, 2018.

[87] Edward J Schwartz, Cory F Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S Havrilla, and Charles Hines. Using logic programming to recover c++ classes and methods from compiled executables. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[88] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *Proceedings of the 22nd Annual Symposium on Network and Distributed System Security (NDSS)*, 2015.

[89] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (IEEE S&P)*. IEEE, 2016.

[90] Yulei Sui and Jingling Xue. Svf: Interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction (CC)*, 2016.

[91] Pengfei Sun, Luis Garcia, Gabriel Salles-Loustau, and Saman Zonouz. Hybrid firmware analysis for known mobile and iot security vulnerabilities. In *Proceedings of 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2020.

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

[93] Zhan Tu, Fan Fei, Matthew Eagon, Dongyan Xu, and Xinyan Deng. Flight recovery of mavs with compromised imu. In *Proceedings of the 2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019.

[94] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable disassembling. In *Proceedings of 24th USENIX Security Symposium (USENIX Security)*, 2015.

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

[96] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Securing untrusted code via compiler-agnostic binary rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, 2012.

[97] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P Kemerlis. Egalito: Layout-agnostic binary recompilation. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[98] Jianliang Wu, Ruoyu Wu, Daniele Antonioli, Mathias Payer, Nils Ole Tippenhauer, Dongyan Xu, Dave Jing Tian, and Antonio Bianchi. Lightblue: Automatic profile-aware debloating of bluetooth stacks. In *Proceedings of 30th USENIX Security Symposium (USENIX Security)*, 2021.

[99] Ruoyu Wu, Taegyu Kim, Dave (Jing) Tian, Antonio Bianchi, and Dongyan Xu. Dnd: A cross-architecture deep neural network decompiler. In *Proceedings of the 31th USENIX Security Symposium (USENIX Security)*, 2022.

[100] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[101] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (IEEE S&P)*. IEEE, 2016.

[102] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *Proceedings of the 24th Annual Symposium on Network and Distributed System Security (NDSS)*, 2015.

[103] Justin Yapp, Remzi Seker, and Radu Babiceanu. Uav as a service: Enabling on-demand access and on-the-fly re-tasking of multi-tenant uavs using cloud services. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*. IEEE, 2016.

[104] Xinwei Zhang, Yuansen Du, Fang Chen, Linlin Qin, and Qing Ling. Indoor position control of a quadrotor uav with monocular vision feedback. In *Proceedings of the 37th Chinese Control Conference (CCC)*, 2018.