Detecting Attacks Against Robotic Vehicles: A Control Invariant Approach

Hongjun Choi, Wen-Chuan Lee, Yousra Aafer, Fan Fei, Zhan Tu, Xiangyu Zhang, Dongyan Xu, Xinyan Deng
{choi293,lee1938,yaafer,feif,tu17,xyzhang,dxu,xdeng}@purdue.edu
Purdue University

ABSTRACT

Robotic vehicles (RVs), such as drones and ground rovers, are a type of cyber-physical systems that operate in the physical world under the control of computing components in the cyber world. Despite RVs' robustness against natural disturbances, cyber or physical attacks against RVs may lead to physical malfunction and subsequently disruption or failure of the vehicles' missions. To avoid or mitigate such consequences, it is essential to develop attack detection techniques for RVs. In this paper, we present a novel attack detection framework to identify external, physical attacks against RVs on the fly by deriving and monitoring Control Invariants (CI). More specifically, we propose a method to extract such invariants by jointly modeling a vehicle's physical properties, its control algorithm and the laws of physics. These invariants are represented in a state-space form, which can then be implemented and inserted into the vehicle's control program binary for runtime invariant check. We apply our CI framework to eleven RVs, including quadrotor, hexarotor, and ground rover, and show that the invariant check can detect three common types of physical attacks – including sensor attack, actuation signal attack, and parameter attack – with very low runtime overhead.

CCS CONCEPTS

• **Security and privacy** → *Embedded systems security*; • **Computer systems organization** → *Embedded and cyber-physical systems*; *Evolutionary robotics*;

KEYWORDS

CPS Security; Robotic Vehicle; Control Invariant; Attack and Detection

ACM Reference Format:
Hongjun Choi, Wen-Chuan Lee, Yousra Aafer, Fan Fei, Zhan Tu, Xiangyu Zhang, Dongyan Xu, Xinyan Deng. 2018. Detecting Attacks Against Robotic Vehicles: A Control Invariant Approach. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3243734.3243752

1 INTRODUCTION

Robotic Vehicles (RVs) are a type of cyber-physical systems (CPS) that consist of both cyber and physical components working jointly to support the vehicle's operations in the physical world. RVs are becoming an integral part of our daily life. Self-driving vehicles [15, 17, 79] are expected to be commonly seen on streets and work sites. Unmanned Aerial Vehicles [4, 62] such as drones are widely used in defense scenarios [51] and start to appear in many commercial and personal applications. Amazon [3] has already demonstrated the feasibility of employing drones for order delivery. The first passenger drone, Ehang 184 [74], was introduced in 2016.

With increasing usage of RVs in a wide range of application domains, the security of RV has become an essential requirement and imperative challenge. Many recent efforts in RV security have focused on protecting the cyber components (e.g., control software and firmware) of an RV from cyber attacks [14, 43, 60], by software security approaches such as control flow integrity (CFI) [14, 60], memory isolation [43], and software/firmware hardening [16, 18, 20, 67]. These solutions are effective in defending against attacks launched via a cyber vector such as program vulnerability exploitation and with cyber payloads, such as injected or trojaned code and ROP.

To make attacks against RVs harder to detect, adversaries have started to target the *physical* components of a victim vehicle. First, the vehicle's sensors can be maliciously misguided through external, non-cyber vectors. For instance, GPS spoofing [35, 75, 78] can disturb GPS sensor readings. Optical sensor spoofing [21] allows an attacker to acquire an implicit control channel, by deceiving the optical flow sensor of a drone with a physically altered ground plane. Gyroscopic sensor spoofing through acoustic noises [72] can lead to drone crashes. In [69], it is shown that an automobile's anti-lock braking system (ABS) [69] can be attacked by injecting magnetic fields to tamper with the wheel speed sensor readings. In [76], it is shown that attackers can manipulate the measurements from MEMS accelerometers via analog acoustic signal injection. Second, attackers may disrupt vehicle communications, such as the wireless channel between a drone and the ground station [34]. Third, attackers may compromise important parameter values (e.g., those deciding control gains) stored in memory through *physical* interference. In [70], it is demonstrated that values in EEPROM and Flash memory can be corrupted by heating up a memory cell inside a memory array without damaging the device. These physical attacks – contrary to cyber attacks – pose new challenges as they cannot be effectively handled by traditional computer security techniques.

Meanwhile, invariant checking is a well-established approach to detecting runtime anomalies caused by program bugs or exploits.

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243752

Traditionally, invariants are properties of the program execution state that should always hold. Such invariants are manually specified by developers or automatically extracted via program analysis. For instance, DAIKON [29] infers invariants from execution using pre-defined templates and then monitors the invariants to detect runtime exceptions. Runtime verification (e.g., [12, 65]) represents legitimate program state transitions in an automaton that can be validated at runtime. Control Flow Integrity (CFI) derives control flow invariants [1] (e.g., function `gee()` can only be invoked by function `foo()`. Invocations from any other caller are considered exceptions). There is a large body of work [7, 22, 37, 42, 71, 83] demonstrating that invariant checking can prevent a wide spectrum of *software oriented* (i.e., cyber) attacks.

Inspired by program invariant checking, we propose a novel control invariant (CI) checking framework for detecting external, physical attacks against RVs. The novelty lies in the fact that we do not aim to check the traditional program-based invariants, but rather control invariants that model both control and physical properties/states of the vehicle. The control invariants are determined jointly by physical attributes of the RV (e.g., weights and shapes), its underlying control algorithm and the laws of physics (e.g., inertia and effects of gravity). The control invariants reflect (and set constraints to) an RV's normal behaviors according to its control inputs (e.g., make a 30^o turn) and current physical states (e.g., velocity and position); any deviation from them will be deemed anomalous.

Our control invariant (CI) framework works as follows. First, it leverages a control system engineering methodology called *system identification* (SI), the "physical" counterpart of program reverse engineering, to extract the control invariants from a subject RV. The SI method takes a control invariant template (equations with unknown coefficients) and a large set of vehicle profiling measurement data (such as system inputs, outputs, and states), as input. It then instantiates the template's coefficients so that the resulted equations provide the best fit for the measurement data. These equations will be used at runtime to predict the behaviors of the vehicle based on inputs and states and hence *serve as the control invariants of the vehicle.* A key observation – well-established in control system engineering – is that the same control invariant template can be used to instantiate control invariants for a family of vehicles with a similar physical organization (e.g., all quadrotors [9]). In other words, their control invariants can be based on the same equation template, only differing in coefficient values. This significantly reduces a subject RV's modeling space in SI, making our framework generic and practical.

Next, the CI framework involves instrumenting the vehicle's control program binary to insert a piece of *control invariant checking code* into the main control loop. At runtime, the code will periodically observe the current system state and independently compute the expected state using the control invariant equations. If the discrepancy between the computed and observed states accumulates and exceeds a threshold within a monitoring window, an alarm will be raised. The window is defined to filter out transient errors caused by physical disturbances (e.g., winds).

Contribution. The salient features of our CI framework include the following: (1) By modeling the physical/control properties and normal dynamics of a subject vehicle, the control invariants directly expose any violation caused by physical, external attacks (which may not cause any program-level anomaly); (2) Based on the generic method of SI, our framework is applicable to a wide range of RVs and does not require per-vehicle controller program reverse engineering to derive control invariants; (3) With monitoring window and threshold, our framework achieves high detection accuracy by filtering out false positive invariant violations. To realize these features we have addressed a number of design and engineering challenges, such as vehicle mission planning for profile data generation, monitoring window size determination, and binary control program instrumentation; (4) Our framework enables software-based detection of physical attacks without hardware modification or addition.

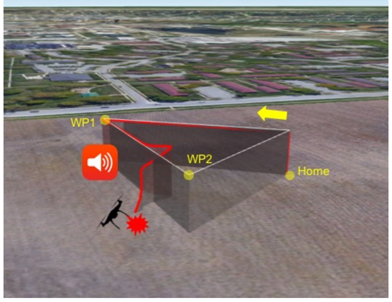


Figure 1: Acoustic noise attack and the affected flight trajectory while performing a simple flight mission

We have developed a prototype of the CI framework and applied it to 11 robotic vehicles including quadrotors, hexarotors and ground rovers. Our evaluation results demonstrate effectiveness of the framework: The derived control invariants are able to detect three types of common attacks including sensor spoofing, control signal spoofing, and parameter corruption; the inserted control invariant checking code incurs low runtime overhead (<2.3%); and the attack detection logic achieves zero false positives during normal operation of subject vehicles.

2 MOTIVATION

To further motivate our framework, we describe, as a working example, an external sensor spoofing attack [21, 35, 46, 58, 69, 72, 75–78] against an IRIS+ quadrotor. A sensor spoofing attack misleads sensor inputs by perturbing the physical environment being sensed. Given the malicious sensor inputs, the vehicle's controller will generate erroneous outputs which will disrupt or damage the vehicle.

A typical RV utilizes a number of sensors to measure the current physical states of the vehicle. In the quadrotor, its Inertial Measurement Unit (IMU) has gyroscopes, accelerometer sensors, and magnetometers, which measure the angular and linear state information. Among these sensors, our sample attack aims to spoof the gyroscope readings, from which erroneous angular state will be inferred, leading to a crash[72, 77]. In particular, the attacker intentionally injects acoustic noises at the resonant frequency of the gyroscope, causing the gyroscope to generate abnormal readings. We note that the attack is an external, physical one without access to the internals of the victim vehicle, and it cannot be detected by existing software security techniques.

```
1   main_loop() {
2
3       angles = read_AHRS();
4
5       targets = navigation_logic();
6
7       // invariant monitoring
8       inv_monitor(targets, angles);
9
10      inputs = attitude_controller(
            targets, angles);
11
12      motor.update(inputs);
13  }
```

(a) main loop

```
1   attitude_controller(targets,angles) {
2
3       error = targets - angles;
4
5       // example pid controller
6       P = kp * error;
7       I = ki * error_sum;
8       D = kd * (angles-angles_last);
9
10      inputs = P + I + D;
11
12      return inputs;
13
14  }
```

(b) attitude controller

```
1   inv_monitor(targets, angles) {
2
3       y = Cx + D*targets;
4       x = Ax + B*targets;
5
6       i_err = y - angles;
7       i_err_sum += i_err;
8       if(i_err_sum > threshold) {
9           raise_alram();
10      }
11
12      if(window == expired)
13          i_err_sum = 0;
14  }
```

(c) invariant monitor

Figure 2: Simplified example of a control loop and invariant monitor

Figure 1 illustrates the attack and its consequences. In the flight mission, the quadrotor is supposed to take off from the home position to an altitude of 20 meters and then move to waypoints 1 and 2 and then go back to the base. The white line indicates the expected trajectory. Between waypoints 1 and 2, the attack is launched. The red line shows the actual flight trajectory. Observe that after the attack is launched, the drone deviates from the planned route and eventually crashes.

To understand how the attack induces the abnormal behavior, Figures 2(a) and (b) show the related code snippets in the quadrotor's control program. (They have been substantially simplified for readability.) Figure 2(a) shows the main control loop. The loop is invoked by a real-time scheduler at a certain frequency. In each iteration, the loop starts by reading sensor inputs. At line 3, the angular information is obtained through the Attitude and Heading Reference System (AHRS). At line 5, the target states are computed by the autonomous navigation logic based on the flight plan. At line 10, the control loop invokes the attitude controller to generate control signals (e.g., rotational speeds of the four rotors) based on the difference between the current and target states.

Figure 2(b) shows the attitude controller based on the classic *Proportional-Integral-Derivative* (PID) control algorithm. Line 3 in `attitude_controller` calculates the error. In lines 6 to 8, the PID algorithm determines the control signals based on the error and a weighted sum of the propositional (P), integral (I), and derivative (D) terms.

During the spoofing attack, the sensor generates wrong angular position readings such that variable `angles` at line 3 in (a) has a faulty state. Subsequently, in the attitude controller, the `error` value at line 3 in (b) is corrupted, leading to wrong actuation signal in `inputs` at line 10 in (a). In the next control loop iteration, the real angular position (due to the wrong actuation signal) will not be reported by the spoofed sensor, further disrupting the vehicle's attitude and eventually leading to its crash.

The CI Approach. Under the CI framework, we can detect such an external attack by checking whether the (perceived) physical state of the vehicle is consistent with its expected state determined by its control model. The control model in turn is defined by the RV's system properties and control algorithm, mathematically represented by our control invariants. For example, the weight, frame shape, and parameter values in a PID controller determine how a drone would respond to external environmental conditions and control signals. Intuitively, the control invariants will predict the next move of the vehicle based on its current state and inputs. An external attack, by definition, will influence the vehicle to deviate from its normal, expected actions/motions, without accurate knowledge about the RV's *internals*, especially the controller's current input, state, and output values. Hence the deviation can be manifested by violation of the control invariants, as if the vehicle is no longer following the control and physics laws.

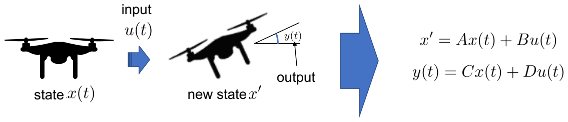


Figure 3: State-space representation of a quadrotor's control

More specifically, our CI framework will work as follows. Given a subject RV, the *system identification* (SI) method will first be applied to "reverse engineer" the dynamics and control model of the vehicle. More specifically, the control model will be represented by two equations [56]: the *output equation* that determines the control output (e.g., new angle of a drone) based on the system's current state (e.g., attitude and position) and its input (e.g., target position); and the *state equation* that determines the next system state from the current state and input. Figure 3 shows the two equations for a quadrotor drone, with $x(t)$, $u(t)$, $y(t)$, and x' denoting its control state at time t, input at t, output at t, and next expected state, respectively. Different systems have different A, B, C, and D matrices. The SI method will concretize the values of the matrices for a specific vehicle, based on its measurement and profile data.

In our working example, we conduct SI on the quadrotor to derive the matrices, which allows us to estimate the output y and future state x at lines 3-4 in Figure 2 (c) with y denoting the predicted value of *angles*. Then, line 6 calculates the error between the *observed* and the *expected* angle values. To avoid false positives due to transient errors, we would not raise an alarm every time an error is observed. Instead, we accumulate the errors in a monitoring window (line 7 in (c)) and compare the aggregated error with a threshold (line 8). We develop an analysis tool to determine the monitoring parameters: window size and threshold (to be described in Section 4.2). In an external attack, the attacker cannot precisely obtain and control the (internal) RV controller's *current* states; and the malicious sensor readings inflicted by the attacker cannot accurately reflect the physical properties or planned moves of the RV. As a result, the spoofed sensor readings would undermine the validity of the feedback loop, leading to substantial errors.

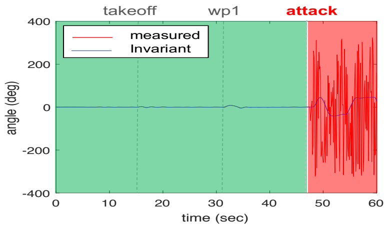


Figure 4: Roll changes during flight

Figure 4 shows the changes of roll angle, one of the attitude angles of a quadrotor under attack. The red curve indicates the sensor readings and the blue curve shows the corresponding values predicted by the control invariants. During normal operation (the green area), the two have negligible difference. During the attack, the roll values fluctuated in the red area and substantially deviated from the predicted values. Note that the red values are what the vehicle (under attack) perceived. The vehicle's controller thus tried to correct the (bogus) errors. Such correction conversely made the drone oscillate and eventually lose balance. In comparison, the predicted values (the blue curve in the red region in Figure 4) did not fluctuate as much because they follow the control model and physics.

Technical Challenges. Leveraging on SI's generality, the CI framework should be applicable to a wide range of RVs. To develop the framework, we need to address the following challenges, especially under the assumption of no control program source code: (1) We need to derive the control invariants (i.e., concretizing the matrices in the state and output equations). The SI method requires a set of training flights to determine the matrices. In addition, treating an RV as a black box during SI may lead to a very large search space (for the model) such that it might not converge with good precision within a reasonable amount of time. (2) We need to identify – from the control program binary – the main control loop and program variables that denote the current system states. The main control loop needs to be located for the insertion of control invariant checking code in the loop; the state variables need to be recognized as they are needed in the evaluation of the two control model equations. (3) We need to set an appropriate size of the monitoring window and the detection threshold (at lines 8 and 12 in Figure 2 (c)). If the threshold is too large, we may not be able to detect an attack in time. If the threshold is too small, transient errors (e.g., overshoot when a drone turns) and environmental disturbances – both correctable by the controller – may be reported as attacks.

3 FRAMEWORK OVERVIEW

Figure 5 gives an overview of the CI framework, which consists of three main components: control invariants extraction, control program reverse engineering, and monitor (i.e., control invariant check code) generation.

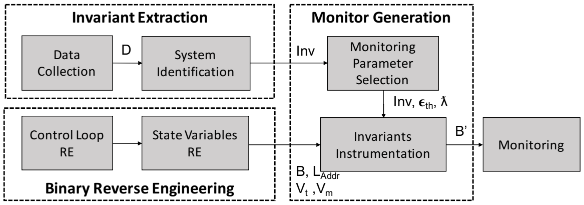


Figure 5: Overview of the CI framework

In the "invariant extraction" step, system identification (SI) is performed to instantiate the invariant equation matrices. First, a set of missions (i.e., flights or rides) to be performed by the subject RV are generated and executed. During the missions, we measure and record the runtime inputs (target states) and system states. These data will be used in SI to derive unknown coefficients.

We then set up a control model template for the target RV. Such a template includes equations of a certain degree/form with *uninstantiated* parameters (e.g., quadratic functions) and can be determined by the vehicle's physical properties and the type of control algorithm used. A family of RVs, for example, drones of the similar physical form (e.g., quadrotors), share the same template but have different parameters (e.g., weight, control gain, inertia, etc.). Quadrotors, hexarotors, and rovers belong to different families hence require different templates. With the model template and measurement data from the test missions, SI determines the optimal template parameters that best fit the data. The instantiated equations reflect the vehicle's control model and hence serve as its control invariants.

Next, to instrument the (binary) control program with invariant-checking logic, we locate the main control loop and state variables, which will be accessed by the invariant checking code when evaluating the invariant equations. This is achieved by dynamic program analysis. Specifically, the control loop is identified by observing the instruction sequences that are periodically executed. The *program variable* corresponding to a *model variable* is identified by comparing the value sequences of the program variable with those of the model variable. The latter are generated by running the model with the same mission.

In the "monitor generation" step, we determine the critical monitoring parameters: error threshold ϵ_{th} and monitoring window size λ. We first determine the window size by calculating the maximum temporal deviation between the actual state sequences (e.g., the attitude variations overtime) and the corresponding model-derived sequences via a sequence alignment algorithm for the training runs. Once the window is determined, we calculate the accumulated transient errors in each monitoring window and use the maximum observed error to set the error threshold.

Finally, we use detour-based binary rewriting [36] to insert the invariant-checking code into the control program binary.

Adversary Model. In this paper, we focus on attacks that interfere with RV operations by corrupting or injecting (actuation or sensor) signals through *external* means (e.g., distorting actuation signals or misleading sensors to generate erroneous readings). We assume that the attacker does not have access to the control program running onboard and hence cannot compromise/bypass the invariant-checking code. We note that the more traditional cyber attacks (launched via software/firmware) are not the focus of this paper as they can be effectively handled by existing software security techniques (e.g., CFI).

We assume that the attacker does not know at least one of the following three aspects about the vehicle: (1) the physical properties of the vehicle, such as weight and detailed frame shape specification; (2) the low-level control algorithm parameter setting; and (3) the maneuver commands from the auto-navigation system or human operator. The first two determine how the vehicle react to control signals and environment condition changes, whereas the third represents mission semantics of the vehicle. Finally, attacks targeting non-vehicle control logic (e.g., a vehicle's computer vision system) are outside the scope of this paper.

4 DESIGN

We continue to use the quadrotor as an example to describe the CI framework in detail. We point out that CI is generic and can be applied to a range of RVs, as shown in Section 5.

4.1 Control Invariant Extraction

Given a subject RV, we need to extract its control invariants that capture how its controller responds to commands and sensor inputs, based on its current state. The control invariants are largely determined by two aspects: vehicle dynamics and the underlying control algorithm.

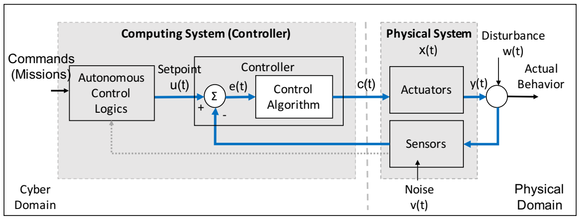


Figure 6: A typical closed-loop control system

Control Invariant. Figure 6 describes a typical RV control system, which consists of both cyber and physical components. The cyber component includes an autonomous control subsystem that takes commands or mission directives from the user and execute the controller program to determine the target state $u(t)$ at time t. The controller implements a control algorithm that compares the target state with the current state perceived by the sensors and determines the error $e(t)$. The control algorithm then computes the control signal $c(t)$ from $e(t)$. The control signal drives the actuators and produces output $y(t)$, which is affected by the external disturbance $w(t)$. The resulting state is perceived by the sensors and fed back to the control loop. The RV's control invariants are represented by a state-space model of the system, consisting of the state (1) and output (2) equations:

$$x' = Ax(t) + Bu(t) \tag{1}$$

$$y(t) = Cx(t) + Du(t) \tag{2}$$

where $u(t)$ (i.e., the target state) is system input and $y(t)$ is system output. As such, the two equations determine the next state and output of the system based on the current state and control signal. The goal of SI is to determine matrices A, B, C and D for the subject vehicle.

Data Collection. Intuitively, derivation of the matrices is equivalent to determining unknown parameters in a number of mathematical equations. To do so, we need to collect the subject vehicle's operation profile data, including the series of state (e.g., velocity), input (e.g., target attitude), and output (e.g., updated attitude) values. We develop a test generation tool that can produce random (but legitimate) missions with environmental effects. Details of the tool are in Appendix A. Note that the SI method only requires a small amount of data (i.e., data from a few flight/missions) to accurately derive the uninstantiated parameters, as shown in our evaluation (Section 5). The amount of data needed by our framework is much smaller than learning-based approaches (e.g., [2, 13, 40, 68]). Intuitively, we just need to collect enough data to solve a few equations with their templates known a priori.

System Identification (SI). The procedure of SI to extract control invariants works as follows. It takes a model template for the RV. Intuitively, a template contains some algebraic equations with unknown coefficients, describing the structure of the vehicle (with unknown metrics) and the nature of its control algorithm (with unknown parameters). We call the former *dynamics template* and the latter *control template*. It also takes the profile data that contain the state, input and output values recorded during the vehicle's SI missions. We then invoke the MATLAB System Identification Toolbox [49] to determine the coefficients that best fit the profile data.

We have two key observations from control engineering practice and our own experience: (1) *All vehicles of similar type/organization share the same dynamics template*. For example, all quadrotors share a dynamics template whereas (ground) rovers share another template, due to their different physical properties. Intuitively, vehicles with the same architecture operate in a similar fashion. The dynamics templates for standard vehicle types are readily available in textbooks and literature. (2) *The basic PID controller can approximate complex control algorithms reasonably well for external attack detection*, as shown in our evaluation (Section 5). Ideally, we would like to precisely model the control algorithm implemented in each subject vehicle. However, this is impractical as modern control algorithms are highly customized and complicated. Reverse engineering their implementations is highly challenging, not to mention that the source code may be unavailable. Although the PID controller is not as sophisticated as the control algorithms in real RVs, it controls the vehicle reasonably well and the errors it induces are correctable and of a much smaller scale – compared with those caused by external attacks. For example, a simple PID controller may lead to over-shoot when making a turn, which would not happen under a more advanced controller. But such (correctable) errors are much smaller compared to errors inflicted by external attacks. Conveniently, all PID controllers share the same equation template.

We further note that the basic PID model can sufficiently approximate *higher-order* dynamics – a common control engineering practice. Even for nonlinear control systems, the majority of control effort is from its linear (i.e., PID) portion. Since second-order response dominates RV dynamics, the lumped vehicle dynamics is a third-order system with the basic PID control. In summary, the PID model is sufficient to capture an RV's closed-loop behaviors. As such, we can use the dynamics template for the specific vehicle type and the PID control template during SI, avoiding manual, per-vehicle control model generation.

Detailed Example. In the following, we first briefly explain the dynamics of the quadrotor and the PID algorithm, which constitute the model template. We then explain how to instantiate the model.

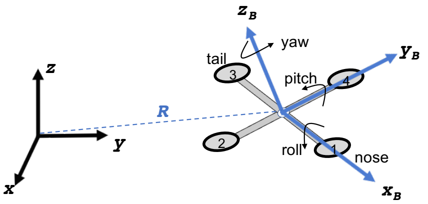


Figure 7: The inertial and body frames of a quadrotor

(1) Determining Quadrotor Dynamics. A quadrotor operates in two frames, the *body frame* and the *inertial frame*, as shown in Figure 7. The inertial frame (on the left) is determined by gravity, which points in the negative z direction. The body frame is defined by the orientation of the quadrotor, with the rotor axes pointing in the positive z_B direction and the arms pointing in the x_B and y_B directions. Intuitively, the thrusts of the rotors are computed in the body frames whereas their effects (e.g., linear and angular accelerations) can only be determined by projecting to the inertial frame.

The motion of a quadrotor is determined by its linear acceleration along the x, y, and z dimensions (of the inertial frame), denoted as $\ddot{x}$, $\ddot{y}$, and $\ddot{z}$, and its angular acceleration along the three dimensions in the body frame: *pitch*, *yaw*, and *roll*, denoted as $\dot{v_y}$, $\dot{v_z}$, and $\dot{v_x}$, respectively. The linear motion can be described by the following Newton-Euler equation.

$$m \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = Rk \begin{bmatrix} 0 \\ 0 \\ \Sigma_{i=1}^{4} \omega_i^2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + \begin{bmatrix} -k_d \dot{x} \\ -k_d \dot{y} \\ -k_d \dot{z} \end{bmatrix} \tag{3}$$

Variable m denotes the mass of the quadrotor, w_i the speed of the ith rotor, g the gravity, and R the conversion matrix from the body frame to the inertial frame. k and k_d are constant factors, and $\dot{x}$, $\dot{y}$, $\dot{z}$ are velocities.

The equation shows that the product of the mass and the accelerations (on the left) is equal to the sum of three terms on the right, denoting the thrust, gravity effect, and the drag force (i.e., air resistance), respectively. Observe that the thrust is along the z_B axis of the body frame (with the values along the x_B and y_B axises being 0), and proportional to the sum of squares of the rotor speeds. To reason about its effect in the inertial frame, it has to be transformed to the inertial frame by the R matrix. In the second term, the effect of the gravity is along the opposite direction of z (in the inertial frame) and hence there is a negative sign. The drag force also has negative signs and is proportional to the speed of the quadrotor. Intuitively, the larger the speed, the stronger the resistance. The linear velocities and positions of the vehicle can be computed from the accelerations and time. The angular motion can be described as follows:

$$\begin{bmatrix} \dot{v}_x \\ \dot{v}_y \\ \dot{v}_z \end{bmatrix} = \begin{bmatrix} lk(-\omega_2^2 + \omega_4^2)I_{xx}^{-1} \\ lk(-\omega_1^2 + \omega_3^2)I_{yy}^{-1} \\ b(\omega_1^2 - \omega_2^2 + \omega_3^2 - \omega_4^2)I_{zz}^{-1} \end{bmatrix} \tag{4}$$

where l is the distance between the rotor and the center of mass, k is a constant, and b is a constant related to drag force. $I_{xx/yy/zz}$ denote the rotational analogue to mass along the x_B, y_B, z_B axes, respectively. The larger the I_{xx} values, the more difficult it is to rotate around the x_B axis (and thus the smaller the $\dot{v}_x$ value). The equation essentially specifies that increasing the 4th rotor velocity and decreasing the 2nd rotor velocity causes rolling; increasing the 3rd and decreasing the 1st causes pitching; and changing all four rotors causes yawing.

(2) Instantiating PID Controller. A PID controller [56] can be described by the following formula.

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{\mathrm{d}e(t)}{\mathrm{d}t} \tag{5}$$

The first term is the proportional term P, which aims to adjust the control signal (e.g., the rotor currents) proportionally to the error. The second term is the integral term I, which aims to consider the history of the error. Intuitively, it compensates for P's inability to reduce the error in the previous rounds. The third term is the derivative term D, which aims to avoid changing the error too quickly (otherwise, the vehicle may overshoot), analogous to a brake. Different coefficient values of K_p, K_i, K_d result in different PID controllers.

By combining the aforementioned equations, specifically, computing $e(t)$ from the accelerations and time interval t and feeding it to the PID equation, we obtain a formula that computes the new states from the previous states, which is the control model template described earlier.

(3) Completing System Identification. Next, we apply the System Identification tool in MATLAB[49] to determine the values of the unknown coefficients in the invariant template.

```
1   for i = 1:N
2       data{i} = iddata(y{i}, u{i}, Ts)
3   end
4   tf = tfest(data, np, nz)
5   [num, den] = tfdata(tf)
6   [A,B,C,D] = tf2ss(num, den)
```

Figure 8: Simplified MATLAB code for system identification.

Figure 8 shows a simplified MATLAB code snippet for the procedure. In the first 3 lines (1-3), the program imports N time-domain datasets collected in N missions of the vehicle, each containing data sampled at a sequence of time instances. The data is combined into an IDDATA object `data` in the MATLAB workspace, which consists of input and output value matrices and a fixed sampling interval `Ts`. The input `u` and output `y` values collected in mission `i` are represented as vectors `y{i}` and `u{i}`.

At line 4, the `tfest` function (provided by the tool) identifies the optimal coefficients for the model template from the vehicle's profile data. Parameters `np` and `nz` represent the encoding of the model template after applying Laplace transformation to the template equations. The transformation turns a time-domain function into the frequency domain and hence substantially reduces the complexity of fitting the profile data. The details are elided as they are not centrally related to our problem. Interested readers are referred to [56]. At line 5, function `tfdata` accesses the resultant model: `num` and `den` that encode the model (with instantiated coefficients) in the frequency domain. They are essentially polynomials regarding the Laplace complex number variable `s`.

$$H(s) = \frac{6.395s^2 - 0.1866s + 66.45}{s^3 + 6.102s^2 + 10.54s + 63.71} \tag{6}$$

However, they are not directly usable as we do not want to produce state/output values in the frequency domain. Instead, we aim to estimate states/outputs in the time domain. At line 6, function `tf2ss` converts the model back to the time domain. The resulting A, B, C and D matrices concretize our model (i.e., the control invariants).

The following shows the example model of roll angle for our 3DR IRIS+ quadrotor with the ArduCopter controller obtained by SI. The output (roll angle) and the internal state of the system are denoted as $y(t)$ and $x(t)$, respectively.

$$x' = \begin{bmatrix} 0.9884 & -0.0493 & -0.0242 \\ 0.0025 & 0.9999 & 0 \\ 0 & 0.0025 & 1.0 \end{bmatrix} x(t) + \begin{bmatrix} 0.0025 \\ 0 \\ 0 \end{bmatrix} u(t) \tag{7}$$

$$y(t) = \begin{bmatrix} 1.8651 & 16.8655 & 10.0631 \end{bmatrix} x(t) + \begin{bmatrix} 0 \end{bmatrix} u(t) \tag{8}$$

The equations for other outputs and other RVs can be similarly derived and hence elided.

4.2 Monitoring Parameters Selection

The model constructed in the previous section represents the control invariants that will be monitored at runtime. Our model is an approximation of the real RV for the following reasons: (1) We use the same dynamics template for vehicles of the same type. However, as individual vehicles may have minor structural differences, our invariant extraction procedure may not be able to capture the small differences in the corresponding dynamics equations. (2) Our procedure does not model uncertain environmental perturbations, such as temperature and wind gusts. (3) We use the basic PID controller to approximate the more advanced controller (e.g., non-linear controller) implemented in the vehicle. All these factors may lead to errors during monitoring. We call them the *transient errors* induced by our approximation. Hence an important challenge is to distinguish transient errors from the errors caused by attacks, which we call *inflicted errors*.

With the assumption that attackers cannot keep accurate track of the vehicle controller's (internal) execution, our key observation is that transient errors are much smaller than externally inflicted errors as the attacker cannot generate malicious signals that *closely* follow the invariants for unknown target states (runtime inputs), without accurate knowledge about the controller program's execution. This implies that, on one hand we should not treat transient errors as indication of true attacks (e.g., the model may lead to overshoot when making a turn due to the simplicity of the PID controller; whereas the real vehicle will not); on the other hand, we do not want to miss or delay true attack detection. Our solution is to accumulate errors (between the model output and the real vehicle output) for a time window, called the *monitor window*, and compare the accumulated errors with a *threshold*. This section explains how to systematically determine the window size and the threshold.

Intuitively, our invariant model can be considered a less sophisticated version of the real system. It can (virtually) fulfill a given mission with a little extra latency. For example, assume the real vehicle needs x seconds to make a turn. The model may take $x + w$ seconds to make the same turn. Therefore, our idea of determining the monitor window is to look for the maximum w in all the primitive operations (e.g., take-offs, turns, and moving-to-waypoint). Once the window is decided, the error threshold is then computed from the maximum observed model-induced errors within the window.

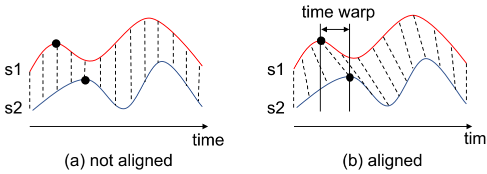


Figure 9: Time alignment of two time sequences. The dashed lines indicate the alignment

To determine w, we adapt the *dynamic time-warping* (DTW) technique [66] that was originally proposed for speech recognition to recognize words when they are pronounced by different persons with varying speeds [63]. Given two time series (e.g., sequences of output sample values over a period of time), time-warping looks for an order-preserving alignment of the timestamps of the sequences, so that the sum of the value differences at the aligned timestamps are minimal. Here, "order-preserving" means that if a timestamp t_1 precedes t_2 in a sequence, its alignment also precedes t_2's alignment in the other sequence. This procedure can be illustrated by Figure 9. Figure (a) shows two time series before DTW. Observe that the lower series $s2$ is a stretched and skewed version of the upper series $s1$. Figure (b) shows that DTW finds an alignment. Observe that the first peaks in the two series are aligned. We use the maximum difference between aligned timestamps, called the *time warp*, as the window size.

The window size and threshold are both vehicle-specific. However, similar to the SI procedure, the determination of the two parameters is highly automated. To achieve good precision, we use the data collection missions (Section 4.1) for this procedure. We note that our data collection missions cover a wide range of normal vehicle operation sequences and disturbances. With monitoring parameters set by such missions, unusual RV operations and severe disturbances would lead to alarms, which is reasonable. For the IRIS+/ArduCopter sample RV, our technique determines that the window size is 2.6 seconds and the error threshold is 91 degree for the roll angle. As shown in Section 5, it takes much shorter than 2.6s to detect attacks.

4.3 Control Program Reverse Engineering and Instrumentation

Based on the control invariants and monitoring parameters determined, our monitoring function, which will check and detect violation of the control invariants at runtime, needs to be inserted into the RV's control program. Commodity RVs may only provide binary executables of control programs without source code. Hence insertion of the monitoring function will have to be via binary code instrumentation. This raises three challenges: (1) We need to identify a location in the binary to insert the function so that it can be periodically executed as part of the control loop. (2) We need to locate the (program-level) control variables to be accessed by the control invariant checking function. (3) We need to perform ARM binary rewriting as most RVs' microcontrollers are ARM-based.

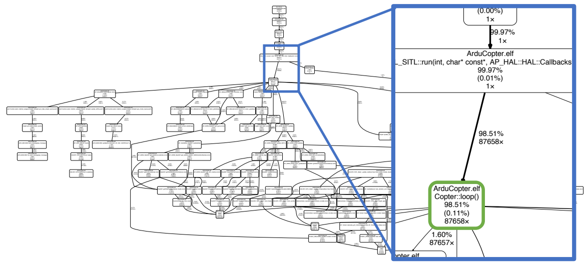


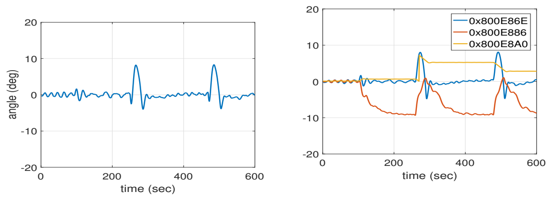
Figure 10: Call graph of ArduCopter with invocation counts

Control Loop Identification. Most RV control programs have a control loop that is regularly invoked to update system states and compute new control outputs. The loop dominates the execution of the program, with access to all critical state variables.

To identify the control loop, we leverage the following observation: A control loop does not manifest itself as a "looping" control flow structure such as for- or while-loop. Rather, it is a function regularly triggered by a timer. As such, the function will exhibit high execution frequency whereas its parent (in the call graph) will not. We note that, in some control programs, there may be some functions such as message callbacks which are triggered frequently. If they are not part of the control loop, their triggering frequency would be much lower than the control (hence invariant-checking) frequency and not as periodic. If those functions indeed perform control tasks with control frequency, our technique will identify them as part of the control loop body for invariant-checking function insertion. Based on this observation, we leverage the Callgrind tool [55] to construct the dynamic call graph annotated with function execution frequencies. Then we traverse the call graph in a top-down fashion to find the first function that has the aforementioned properties. Figure 10 shows an example call graph of ArduCopter (i.e., the control software for IRIS+ quadrotor) annotated with call counts and costs. Note that the enlarged area includes the control loop function `Copter::loop()` (the green box). The parent function calls the loop function 87658 times while the parent itself is executed only once.

Identifying Memory Locations for Critical State Variables. According to the control invariant equations (1) and (2), the current state $x(t)$ and input $u(t)$ (e.g., the target attitude) are needed to compute the new state x' and the output $y(t)$ (e.g., next attitude). Therefore, our control invariant check function needs to access the input value, compute the new state and compare it with the corresponding current state variables in the original control program. To identify the memory locations of these variables, we collect the value traces for all variables defined in the control loop and compare them with the value traces of the model variables generated by the invariant model under the same mission.

Specifically, we use Valgrind to instrument and trace all the memory writes that occur in the control loop function. Given a mission, a value trace is generated for all variable updates that happen inside the control loop function. We then partition the trace into multiple time series of values, each series containing all the updates for a unique memory location. On the other hand, MATLAB allows us to execute the control invariant model we have derived. Intuitively, it simulates the vehicle operations by computing all the state values according to the model equations. We instrument the MATLAB program to collect traces for the model variables and then execute it with the same input as for the real vehicle. For each model variable trace, we identify the program variable trace with the smallest Euclidean distance, which establishes the mapping between the model variable and the corresponding program variable (and its memory location).



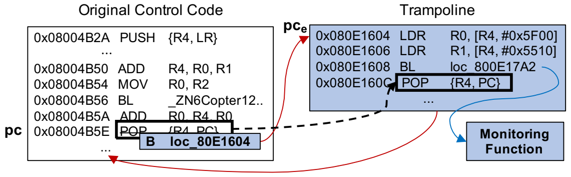
(a) A model variable

(b) Multiple program variables

Figure 11: State variable value traces at model and program levels

For example, Fig 11(a) shows a model variable value trace for the roll angle state and (b) three program variables' value traces. We can easily observe that the trace for memory location 0x800E86E (blue line) in (b) matches (a).

ARM Binary Rewriting. We apply trampoline-based binary rewriting [10, 36, 47] to insert the monitoring function and its invocation to the ARM binary of the control program. Specifically, the monitoring function (source) code is first compiled and made position-independent. The resulting code snippet is added to the end of the control program binary. Given a code location inside the main control loop, we add a jump – usually at the end of the control loop – to a small code snippet called a *trampoline* to invoke the monitoring function.



```
0x08004B2A  PUSH  {R4, LR}
            ...
0x08004B50  ADD   R4, R0, R1
0x08004B54  MOV   R0, R2
0x08004B56  BL    _ZN6Copter12..
0x08004B5A  ADD   R0, R4, R0
0x08004B5E  POP   {R4, PC}
                  B  loc_80E1604
            ...
```

```
0x080E1604  LDR   R0, [R4, #0x5F00]
0x080E1606  LDR   R1, [R4, #0x5510]
0x080E1608  BL    loc_800E17A2
0x080E160C  POP   {R4, PC}
            ...
```

Figure 12: Example of trampoline-based rewriting

Figure 12 shows an example. Suppose a long jump takes n bytes and we want to add an invocation to the monitoring function at position pc. We compose a trampoline code snippet that contains the monitoring function's invocation, followed by the n bytes of instructions starting at pc of the *original* binary and then a jump to location $(pc + n)$. The trampoline is attached at the end of the control program binary, say at location pc_e. Then the original n bytes at pc are replaced by a longjump to pc_e. At runtime, when the execution reaches pc, it jumps to pc_e to execute the trampoline, which will first invoke the monitoring function and then execute the original n bytes of instructions, before jumping back to location $(pc + n)$.

4.4 Runtime Control Invariant Monitoring

Algorithm 1 Runtime Control Invariant Monitoring

```
1:  $x$ control states of the real vehicle
2:  $u$ control input of the real vehicle
3:  $y$ control output of the real vehicle
4:
5:  procedure InvMonitor($x, u, y$)
6:      $x_p \leftarrow A \cdot x_p + B \cdot u$
7:      $y_p \leftarrow C \cdot x_p + D \cdot u$                    ▷ calculates expected output
8:      $s\_err \leftarrow |y - y_p|^2$
9:      $err\_sum \leftarrow err\_sum + s\_err$
10:     $error \leftarrow err\_sum/t$
11:     $t{+}{+}$
12:     if $error > threshold$ then                         ▷ runtime attack detected
13:         $alert()$
14:     end if
15:     if $t > monitor\_window$ then                        ▷ window expires
16:         $t \leftarrow 0$
17:         $err\_sum \leftarrow 0$
18:         $x_p \leftarrow x$
19:     end if
20: end procedure
```

Algorithm 1 describes the logic of the runtime control invariant monitoring code. It takes the current states, the control input and output of the real vehicle (identified by control program reverse engineering) as arguments. It then computes the predicted new state x_p and the predicted new output y_p, using the control invariant equations (lines 6 and 7). The squared error s_err is computed and aggregated. Note that squared error is sensitive to outliers (caused by attacks). At line 12, the algorithm compares the error with the pre-determined threshold. If the (accumulated) error exceeds the threshold, function $alert()$ will be invoked. Invocation of $alert()$ will further lead to attack response, which may be vehicle/mission-specific. For example, in response to an alert, a quadrotor may switch the flight mode to a fail-safe mode which involves aborting the mission and landing. Attack response/recovery is beyond the scope of this paper but we will briefly discuss it in Section 6. At line 15, the algorithm checks if the monitoring duration has exceeded the pre-determined monitor window size. If so, it resets the window counter t and accumulated error err_sum to zero; and the model (invariant) states to the real vehicle's states. Note that the monitoring algorithm/code does not use the real vehicle state for its own state prediction (lines 6 and 7), which means that the control invariant model basically executes independently of the real vehicle controller within a window. When the window expires and no alert is raised, the accumulated error is reset to zero to prevent further accumulation of transient errors (Section 4.2) and a new monitoring window starts with the real vehicle states x as the initial states (line 16-18).

Table 1: Subject Vehicles in Evaluation

Type	HW Vendor	Model	Controller Software
Quadrotor	3D Robotics	IRIS+	ArduCopter 3.4
Quadrotor	3D Robotics	IRIS+	PX4 Pro 1.6
Rover	Erle Robotics	Erle-Rover	APMrover2 3.2
Hexacopter	Ardupilot	APM SITL	ArduCopter 3.6
Quadrotor	Parrot	Bebop2 (JSBSim)	Paparazzi 5.12
Quadrotor	Erle Robotics	Erle-Copter (Gazebo)	ArduCopter 3.4
Quadrotor	3D Robotics	3DR Solo (Gazebo)	PX4 Pro 1.6
Quadrotor	Parrot	ARDrone2 (JSBSim)	Paparazzi 5.12
Rover	Ardupilot	APM SITL	APMrover2 2.5
Quadrotor	3D Robotics	3DR Solo	Ardupilot-solo 1.3.1
Quadrotor	Pixhawk-based	Self-built	ArduCopter 3.4

5 EVALUATION

5.1 Implementation

The implementation of the CI framework consists of the following: (1) a Valgrind-based dynamic analysis component for identifying control loop and important state variables; (2) a mission generator (for SI) in Python that takes state machine specification and parameter ranges as input, and generates random but realistic missions; (3) a profiler implemented based on MAVlink to collect measurement data for SI; (4) a control invariant extraction and parameter selection component implemented on MATLAB; (5) a monitoring function template implemented in C++ that takes the derived control invariant equation matrices and performs matrix computation; and (6) an ARM binary rewriter written in Python.

5.2 Subject Vehicles and Attacks

We use 11 different RVs of three types: quadrotors, hexarotors, and (ground) rovers.

Figure 13: Real RVs in evaluation: 3DR IRIS+, Erle-Rover, 3DR Solo, Self-built (left to right)

In particular, we have ten different vehicles among which four are real vehicles (shown in Figure 13). The others are virtual vehicles provided by various simulator packages. For example, Gazebo has a full-fledged simulator of the 3DR Solo quadrotor by 3D Robotics. Details of the vehicles are shown in columns 1-3 of Table 1. We use 5 different control programs (column 4). For vehicle simulation, we use Gazebo, JSBSim [39], and APM SITL [5] (column 3). We run the simulators on Ubuntu 64-bit with Intel(R) Xeon(R) CPU E5620 @ 2.40GHz x8 processor and 3.8 GB RAM.

Attacks. Most reported attacks against RVs that exploit physical channels/components can be classified into (1) sensor spoofing attacks [21, 35, 46, 58, 69, 72, 75–78], (2) control signal spoofing attacks [11, 45], and (3) parameter corruption attacks [19]. While it is difficult to implement all these attacks in real world due to lack of special attack devices (e.g., the equipment to emit acoustic noises), we are able to simulate these attacks without losing realism.

To simulate sensor spoofing, we choose to compromise inertial sensors and GPS sensors. These sensors are necessary for all the vehicles in our experiments. Specifically, we insert the attack simulation code at the interface between the (real) control program and sensor modules and manipulate sensor measurements by injecting malicious signals. To simulate control signal spoofing, we target the *motor pulse width modulation* (PWM) signals that are used to adjust the rotation of motors/rotors. Such signals are generated by the control program and emitted to the physical vehicle peripherals through a communication channel (e.g., bus). We insert a piece of signal-manipulation code into the PWM signal emission module of the control program. To simulate parameter corruption attacks, we add a piece of attack code to the control program that modifies the control parameters (e.g., the PID control coefficients) at runtime.

While we modify the real-world control programs to simulate the external physical attacks for experimentation convenience, we do assume that the attackers do not have access to the vehicle's internals including the control program and they do not have accurate knowledge about the vehicle's missions.

5.3 Experiments and Results

Our evaluation focuses on two aspects: efficiency and effectiveness. To evaluate efficiency, we measure the execution time of key steps of the CI framework, including the dynamic analysis that identifies the control loop and state variables, the SI procedure, and monitoring parameter determination. More importantly, we measure the overhead of control invariant checking at runtime.

To evaluate effectiveness, we conduct a variety of experiments: (1) We validate that the extracted control invariants can properly predict normal vehicle behaviors and do not raise false alarms during normal operation. (2) We measure the false negative rate of attack detection. (3) We show that control invariants are vehicle-specific hence the invariants extracted for vehicle *A* cannot be used for vehicle *B*. (4) We evaluate the effectiveness of our monitoring parameter setting techniques (Section 4.2) by showing that improperly set parameter values may lead to false positives and false negatives. (5) We measure error changes under various environmental conditions (i.e. wind) to show that our framework is effective even in unfavorable environments. We also vary the scale of attacks to show that our framework remains effective under different scale.

Efficiency. Table 2 summarizes the results of efficiency evaluation. We let each vehicle execute 20 missions and apply the SI method to extract its control invariants (Section 4.1). In particular, column RO shows the runtime overhead of the instrumented control program. Since the control invariant monitoring function mainly involves a small number of matrix multiplication and error calculation operations, it incurs very low runtime overhead (below 2.3%). The profiling overhead (PO) is large but it is offline. The system identification time (ST) is less than 1 minute.

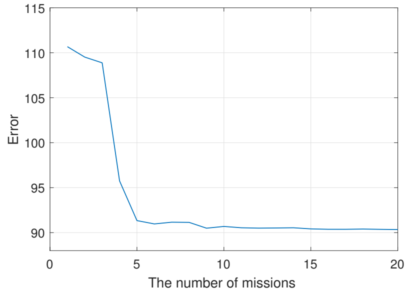


Figure 14: Convergence of system identification

Figure 14 shows the convergence of the SI procedure for IRIS+ /ArduCopter relative to the number of missions (flights) conducted. The y-axis shows the average distance between the measured output and expected output from our model. Observe that the SI-generated model reaches a fix-point at about 5 missions, indicating that it only takes a few missions to achieve reasonable accuracy. The results for other vehicles are similar and hence elided.

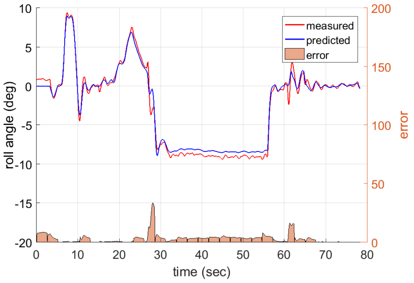


Figure 15: Match between real behavior and model prediction

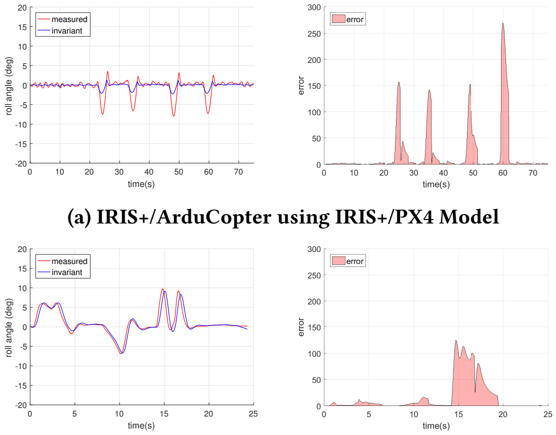
Effectiveness. In the first experiment, we use our mission generator (details in Appendix) to generate a new set of 20 normal missions (not the ones used in SI) for each vehicle. We then let the instrumented vehicles execute these missions, during which the control invariant monitor does *not* raise any attack alarm. Figure 15 shows how closely the control invariants' prediction matches the real (normal) behavior of the IRIS+/ArduCopter vehicle – in a real flight. In the figure, the red curve denotes the measured roll angle and the blue curve denotes the predicted values. Only small errors exist between the two curves (i.e., the orange area at the bottom). The larger errors at the peaks/dips are due to the approximation nature of our framework.

Table 2: Summary of Efficiency/Overhead Results

System	CS (KB)	PS (KB)	PO (%)	ST (sec)	Target	ID	WS (s)	TH	SO (%)	RO (%)
IRIS+/ArduCopter	772	1,212	285	29.7	Roll angle	3 x 3	2.6	91.0	0.04	1.87
IRIS+/PX4 Pro	857	719	1,096	28.2	Roll rate	3 x 3	4.4	6.0	0.04	1.01
Erle-Rover/APM:Rover2	1,164	552	115	13.0	Steering rate	3 x 3	4.2	2.5	0.08	0.53
APM SITL/ArduCopter	14,125	1,174	280	33.6	Roll angle	3 x 3	2.0	43	0.08	0.66
Bebop2/Paparazzi	2,337	2,848	603	18.3	Roll rate	3 x 3	1.8	5.6	0.48	1.23
Erle-Copter/ArduCopter	14,640	606	1,400	16.1	Roll angle	3 x 3	2.5	45.6	0.09	0.55
Solo/PX4 Pro	14,599	750	1,480	39.3	Roll rate	3 x 3	3.6	8.2	0.08	2.23
ARDrone2/Paparazzi	2,539	2,449	598	18.1	Roll rate	3 x 3	2.0	6.3	0.45	1.12
APM SITL/Rover	11,128	1,155	124	12.8	Steering rate	3 x 3	1.7	1.54	0.11	0.94
3DR Solo/ArduCopter	660	1,097	1909	21.5	Roll angle	3 x 3	0.5	13.5	0.06	1.31
Custom/ArduCopter	738	1,146	1399	14.8	Roll angle	3 x 3	4.7	20.7	0.06	1.40

[*] CS: Code Size, PS: Profile Size, PO: Profiling Overhead, ST: SI Time, ID: Invariant Dimension (A matrix), WS: Window Size, TH: Threshold, SO: Code Size Overhead, RO: Runtime Performance Overhead

In the second experiment, we launch attacks during 20 missions (of each vehicle) and record the number of attacks that are detected. Our framework detects *all* the attacks within an average of 0.2 second after they are launched (i.e., zero false negative rate with detection timeliness).



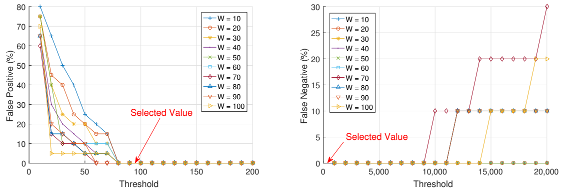
(a) IRIS+/ArduCopter using IRIS+/PX4 Model

(b) SITL/HexaCopter using IRIS+/ArduCopter Model

Figure 16: Applying different models on different vehicles

In the third experiment, we use the model extracted from IRIS+ /PX4 to predict behaviors of IRIS+/ArduCopter and use the model extracted from IRIS+/ArduCopter (a quadrotor) to predict the behaviors of SITL/ArduCopter (a hexarotor). The first pair involves the same vehicle with two different control programs and the second pair involves two different vehicles using the same control program. The results are shown in Figures 16a and 16b, respectively. Observe that the errors are non-trivial. This observation confirms that control invariant models are vehicle (including control algorithm)-specific.

In the fourth experiment, we measure the FP and FN detection rates under different monitoring parameter (window and threshold) values. To measure FPs, we run 20 normal missions. To measure FNs, we launch attacks during 20 missions and observe how many attacks are missed, under different parameter values. Figures 17a and 17b show the results. We observe that: (1) under the same threshold, a larger window generally leads to fewer FPs and more FNs and (2) for the same window size, a larger threshold leads to fewer FPs and more FNs. (1) is because the accumulated error is normalized (i.e., divided by the time lapse within the window) before comparison with the threshold such that a larger window leads to smaller normalized errors. We also observe that FNs only occur when the threshold is set to a very large value. This experiment highlights the importance of our monitoring parameter determination technique (Section 4.2), which achieves zero FP and zero FN.



(a) False positive rates

(b) False negative rates

Figure 17: FP and FN under different parameters

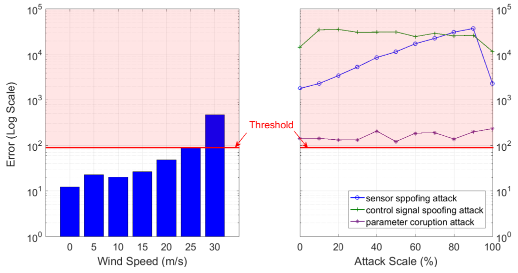


Figure 18: Error under different wind speed and attack scale

In the fifth experiment, we measure the error (i.e., the result of control invariant check) under different wind speed and attack scale. We set up a mission in which a quadrotor makes a sharp turn, which is highly sensitive to environmental conditions and injected noises. Figure 18 shows the results. First, in the left sub-figure, the error significantly increases at 30m/s wind speed, while the other cases (i.e., below 25m/s) result in small errors. The quadrotor is not able to take off when the wind speed is higher than 35m/s. Note that the 25m/s wind speed corresponds to the "storm force" (i.e., Beaufort Scale 10 [73]) and manufacturers would not recommend flying a drone under such an extreme condition.

We also measure the error under various attack scale on the roll angle. Specifically, 0% attack means that we set the roll angle to 0, 10% attack means that we set it to a random value in [-10%× 360, 10%×360] and 100% attack means that we set it to [-360,360]. The quadrotor performs the same mission. The right sub-figure in Figure 18 shows the maximum error during three different attacks under different scales. Note that the errors are substantially larger compared with the error threshold (=91) for this vehicle and our control invariant check detects all the attacks. The errors under the parameter corruption attacks are relatively smaller because the vehicle has its own internal protection that caps control gains, making a larger noise impossible.

5.4 Case Studies

In this section, we present case study with two *real* RVs under six attacks. The subject vehicles include the IRIS+ quadrotor and Erle (ground) rover running ArduCopter and APMrover2 control programs, respectively. We launch three attacks on the IRIS+ and three attacks on the Erle-rover.

In the first case, we let the IRIS+ fly a mission in which it first takes off from the home position to an altitude of 2 meters and then turns left and right. We launch the sensor spoofing attack during the flight. The attack is launched at time instance 5.7s (Figure 19a), when the inertial sensor reading is disrupted and then the roll measurement is compromised, leading to the quadrotor's crash. Our framework detects the attack only 100ms after it is launched. Figure 19g shows the invariant check error between the measured and predicted roll values under the attack. A video of the attack and its detection can be viewed at [23].

In the second case, the IRIS+ performs the same mission as the first case. During the flight, we launch a control signal spoofing attack. The IRIS+ is equipped with four MN2213 950kV DC motors, actuated by motor pulse width modulation (PWM) signals to adjust the motors' rotation and hence control the speed and attitude of the vehicle. We launch the attack at time instance 2.4s (Fig 19b), when one of the signals is maliciously replaced by a constant value. The quadrotor loses the roll control and then crashes. Figure 19h shows the error between the invariant-predicted and measured roll values. Our framework detects the attack 100ms after it is launched. A demo video is at [24].

In the third case, we corrupt a control parameter in the same mission. This leads to a control gain change (by a factor of 6). We launch the attack at 10.5s (Figure 19c). Upon the attack, the quadrotor flies in a circle and gradually loses balance. Figure 19i shows the invariant check error. Our framework detects the attack 1.1s after it is launched. A demo video is at [25]. Note that this attack does not substantially violate the invariants at once as it takes some time for the effect of the compromised parameter to manifest itself physically. Therefore, it takes longer time to detect the attack. However, the detection time is still short enough for possible recovery as the impact of this attack is milder than the earlier ones.

In the fourth case, the Erle-rover performs a mission in which it departs from the home position, follows a rectangular track, and then comes back to the home location. We launch the motor input spoofing attack during the mission. The attack module modifies the value of a steering servo which is generated by the controller and controls the steering rate. We launch the attack at 7.7s (Figure 19d). The rover then fails to follow the track and gets stuck in a circle pattern. Figure 19j shows the error between the invariant-predicted and measured steering rates. Our framework detects this attack 100ms after it is launched. A video is at [26].

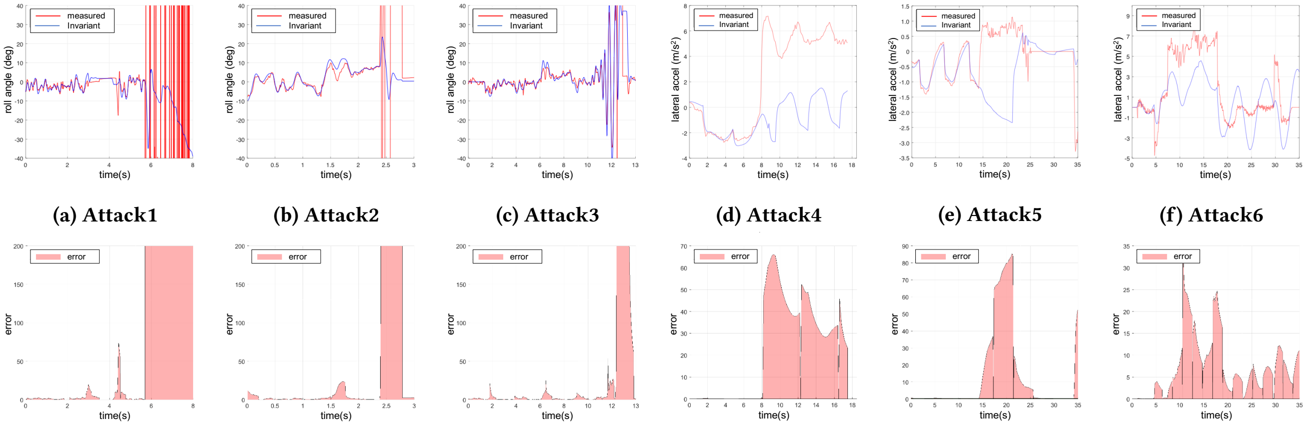
In the fifth case, the Erle-rover performs the same mission under GPS spoofing. We launch the attack at 14.1s (Figure 19e). The rover then deviates from the track and moves to a non-home location at the end of the mission. Our framework detects the attack 600ms after it is launched. Figure 19k shows the error and a video is at [27].

In the sixth case, we aim to reproduce an attack similar to that in [76], in which the attacker can manipulate the sensor signal in a non-random fashion. The Erle-rover performs a mission where it follows the straight line. During the mission, an attacker controls the yaw sensor in a sophisticated fashion and manages to change the measurements to +30, 0 and -30 deg at the 4th, 17th and 28th second, respectively (Figure 19f). The rover then deviates from the original straight line. As shown in Figure 19l, the errors during the attack are significant. This case demonstrates that, even if the attacker can manipulate signals in a delicate way, without knowing the accurate motion plan of the vehicle, the invariant check errors are still sufficient for detection. A demo video can be found at [28].

6 DISCUSSION

Mimicry Attacks. The CI framework can effectively detect external attacks against RVs that exploit physical channels/vulnerabilities. In theory, we cannot rule out the possibility that an attacker closely mimics the behaviors of the target vehicle (e.g., by following the model of a similar vehicle). For example, a sophisticated attack can be launched such that the compromised sensor readings largely respect the vehicle's dynamics and laws of physics while generating small errors. We expect that such attacks are difficult to implement, when the attacker cannot directly manipulate the target states/values and instead has to rely on indirect physical channels (e.g., affecting gyroscope sensors by acoustic noises). Moreover, as shown in our experiments (Section 5), control invariants are vehicle-specific hence setting a high bar for high accuracy approximation. More importantly, the behaviors of a vehicle are determined by three factors: physics, control algorithm and parameters, and mission plan and user commands (runtime inputs). Even if the attacker manages to grasp the first two, missing the third factor would still expose the attack as shown in the last attack case in Section 5.4. Note that we can even equip the vehicle with a proactive self-validation procedure that is executed regularly to detect attacks. Specifically, the control software can switch to a different set of pre-defined control parameters and then make a few maneuvers. Since these new parameters are unknown to the attacker, substantial errors between the perceived motions (under attacker's control) and predicted motions (from control invariants) are expected.

More Adaptive Detection. We rely on proper monitoring parameters (monitoring window and accumulated error threshold) to distinguish attacks from transient errors. However, under highly



(a) Attack1 **(b) Attack2** **(c) Attack3** **(d) Attack4** **(e) Attack5** **(f) Attack6**

(g) Attack1 error **(h) Attack2 error** **(i) Attack3 error** **(j) Attack4 error** **(k) Attack5 error** **(l) Attack6 error**

Figure 19: Case study: attacks on IRIS+/ArduCopter and Erle-rover

unfavorable environmental conditions not experienced during training, false positive detection may happen. One possible solution is to make those parameters adaptive to the environment during a real mission. The physical properties of a vehicle (i.e., its mass under different payloads) may also change, so should its model. A possible solution is to pre-define a number of vehicle configurations and construct a model for each of them.

Attack Response. After an attack is detected, a proper response to it is essential. While attack response is outside the scope of this paper, we note that many response/recovery mechanisms exist, such as deploying a parachute (for aerial vehicles) and stopping the vehicle (for ground vehicles). More advanced attack recovery techniques also exist that maintain the vehicle's non-stop operation via controller redundancy [30, 81] or checkpointing [44].

7 RELATED WORK

System Identification. System Identification (SI) [61] is a mature and widely practiced method in control engineering to infer the control model of a subject system. The model describes the relation between the system's input and output. Besides typical model construction [8, 48], SI is also applied to disturbance handling [50, 80], worst case analysis [33], and so on. Different from those application scenarios, we in this paper customize SI for RV attack detection.

Attacks against RVs. Many external attacks against RVs have been reported. In [11, 45], the authors demonstrate the feasibility of infiltrating internal vehicle networks. In [38], the authors exploit a car tire pressure sensing system, which utilizes Radio Frequency (RF)-based wireless motes. GPS spoofing [35, 75, 78] by sending interfering signals is a typical active physical sensor attack. Optical sensor input spoofing [21] involves obtaining an implicit control channel by tricking optical flow sensors with a physically altered ground plane. In [72], the authors propose a gyroscopic sensor attack with intentional acoustic noise to crash drones. Later the authors of [76] compromise accelerometers by injecting acoustic noise in a controlled manner, as a more advanced form of the attack. Anti-lock Braking System (ABS) attack [69] involves injecting magnetic fields to spoof wheel speed sensor. In [34], it is shown that an attacker with an antenna and a malicious ground station can compromise a benign UAV by sending malicious packets. In [59], the authors propose attacks on a camera-based ground vehicle by relaying and spoofing signals. In [54], the authors analyze the effects of false data injection attacks on control systems.

Attack Detection. Attack detection for RVs [2, 6, 31, 32, 40, 41, 52, 53, 68, 81, 82, 84] can be based on the following methodologies: signature, learning, system redundancy and specification. Signature-based detection [32, 41] monitors the target system and compares it with pre-determined attack patterns known as attack signatures. It generally achieves low false positive rate, but it needs to maintain an up-to-date attack dictionary and cannot handle zero-day attacks effectively. Redundancy-based techniques [30, 31, 82] duplicate important system components (e.g., controller) and cross-check their states/outputs at runtime [81] to detect attacks/anomalies. The redundancy can be in the form of software and/or hardware. However, this approach, by definition, incurs additional cost and system complexity (e.g., for implementing multiple versions of the same controller). The learning-based approach [2, 13, 40, 68] monitors abnormal behaviors using a machine learning-based model. The normality can be defined by unsupervised and supervised training. However, in the physical domain, it is hard to obtain large sets of normal and attack training data. Although unsupervised learning eliminates the need for attack data, it may be susceptible to a high false positive rate. Our CI framework is not dependent on learning from a large amount of data. We only need to instantiate control invariant parameters with standard model templates, based on profiling data from just a few test missions. Behavioral rule-based techniques [6, 52, 53, 84] use a specification to describe normal system operations. These techniques model program state transitions or execution time constraints, whereas our CI framework models control invariants based on a standard controller (e.g., PID) and physics, without having to reverse engineer the specific control algorithm of a vehicle. Moreover, external physical attacks may not cause any program-level anomaly.

8 CONCLUSION

We have presented a new comprehensive framework CI for detecting external physical attacks against RVs, based on the definition, derivation, and monitoring of control invariants for the vehicles. The control invariants are derived from the physical dynamics and control model of a subject vehicle. The corresponding invariant-checking logic is implanted in the vehicle's binary control program. Our framework does not require control program source code or per-vehicle control algorithm reverse engineering. Our evaluation of the CI framework with 11 physical or simulated RVs – all running real-world control programs – demonstrates high attack detection accuracy and low runtime overhead.

ACKNOWLEDGEMENTS

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

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *the 12th ACM conference*. ACM Press, New York, New York, USA, 340–353.

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

[3] Amazon Prime Air Delivery 2016. Amazon Prime Air. https://www.amazon.com/Amazon-Prime-Air/b?node=8037720011.

[4] ArduPilot 2017. ArduPilot :: Home. http://ardupilot.org/.

[5] ArduPilot Dev Team 2016. SITL Simulator (Software in the Loop). http://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html.

[6] Stanley Bak, Karthik Manamcheri, Sayan Mitra, and Marco Caccamo. 2011. Sandboxing controllers for cyber-physical systems. In *Proceedings of the 2011 IEEE/ACM Second International Conference on Cyber-Physical Systems*. IEEE Computer Society, 3–12.

[7] Jason Bau and John C Mitchell. 2011. Security modeling and analysis. *IEEE Security & Privacy* 9, 3 (2011), 18–25.

[8] George A Bekey. 1970. System identification-an introduction and a survey.

[9] Samir Bouabdallah, Pierpaolo Murrieri, and Roland Siegwart. 2004. Design and control of an indoor micro quadrotor. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, Vol. 5. IEEE, 4393–4398.

[10] Bryan Buck and Jeffrey K Hollingsworth. 2000. An API for runtime code patching. *The International Journal of High Performance Computing Applications* 14, 4 (2000), 317–329.

[11] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, Tadayoshi Kohno, et al. 2011. Comprehensive Experimental Analyses of Automotive Attack Surfaces.. In *USENIX Security Symposium*. San Francisco.

[12] Feng Chen and Grigore Roşu. 2007. Mop: an efficient and generic runtime verification framework. In *Acm Sigplan Notices*, Vol. 42. ACM, 569–588.

[13] Yuqi Chen, Christopher M Poskitt, and Jun Sun. 2018. Learning from Mutants: Using Code Mutation to Learn and Monitor Invariants of a Cyber-Physical System. *arXiv preprint arXiv:1801.00903* (2018).

[14] Abraham A Clements, Naif Saleh Almakhdhub, Khaled S Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. 2017. Protecting Bare-metal Embedded Systems With Privilege Overlays. In *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 289–303.

[15] CNN 2012. Self-driving cars now legal in California. http://www.cnn.com/2012/09/25/tech/innovation/self-driving-car-california/index.html.

[16] Frederick B Cohen. 1993. Operating system protection through program evolution. *Computers & Security* 12, 6 (1993), 565–584.

[17] comma.ai 2018. commaai/openpilot: open source driving agent. https://github.com/commaai/openpilot.

[18] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. 1998. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.. In *USENIX Security Symposium*, Vol. 98. San Antonio, TX, 63–78.

[19] Ang Cui, Michael Costello, and Salvatore J Stolfo. 2013. When Firmware Modifications Attack: A Case Study of Embedded Exploitation. (2013).

[20] Ang Cui and Salvatore J Stolfo. 2011. Defending embedded systems with software symbiotes. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 358–377.

[21] Drew Davidson, Hao Wu, Robert Jellinek, Vikas Singh, and Thomas Ristenpart. 2016. Controlling UAVs with Sensor Input Spoofing Attacks.. In *WOOT*.

[22] Onur Demir, Wenjie Xiong, Faisal Zaghloul, and Jakub Szefer. 2016. Survey of Approaches for Security Verification of Hardware/Software Systems. *IACR Cryptology ePrint Archive* 2016 (2016), 846.

[23] Demo Video 2018. Attack Case 1: Sensor Spoofing Attack on IRIS+. https://bit.ly/2Kb6TcK.

[24] Demo Video 2018. Attack Case 2: Control Signal Attack on IRIS+. https://bit.ly/2Ka5PpG.

[25] Demo Video 2018. Attack Case 3: Control Parameter Corruption Attack on IRIS+. https://bit.ly/2LQTTOo.

[26] Demo Video 2018. Attack Case 4: Control Parameter Corruption Attack on Erle-Rover. https://bit.ly/2LBpK6l.

[27] Demo Video 2018. Attack Case 5: Motor Input Spoofing Attack on Erle-Rover. https://bit.ly/2LFPKOk.

[28] Demo Video 2018. Attack Case 6: Sensor Manipulation Attack on Erle-Rover. https://bit.ly/2NXJDRQ.

[29] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* 27, 2 (2001), 99–123.

[30] Fan Fei, Zhan Tu, Ruikun Yu, Taegyu Kim, Xiangyu Zhang, Dongyan Xu, and Xinyan Deng. 2018. Cross-Layer Retrofitting of UAVs Against Cyber-Physical Attacks. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2018)*.

[31] Paul M Frank. 1990. Fault diagnosis in dynamic systems using analytical and knowledge-based redundancy: A survey and some new results. *automatica* 26, 3 (1990), 459–474.

[32] Wei Gao and Thomas H Morris. 2014. On cyber attacks and signature based intrusion detection for modbus based industrial control systems. *The Journal of Digital Forensics, Security and Law: JDFSL* 9, 1 (2014), 37.

[33] Guoxiang Gu and Pramod P Khargonekar. 1992. A class of algorithms for identification in $H\infty$. *Automatica* 28, 2 (1992), 299–312.

[34] Kate Highnam, Kevin Angstadt, Kevin Leach, Westley Weimer, Aaron Paulos, and Patrick Hurley. 2016. An uncrewed aerial vehicle attack scenario and trustworthy repair architecture. In *Dependable Systems and Networks Workshop, 2016 46th Annual IEEE/IFIP International Conference on*. IEEE, 222–225.

[35] Todd E Humphreys, Brent M Ledvina, Mark L Psiaki, Brady W O'Hanlon, and Paul M Kintner Jr. 2008. Assessing the spoofing threat: Development of a portable GPS civilian spoofer. In *Proceedings of the ION GNSS international technical meeting of the satellite division*, Vol. 55. 56.

[36] Galen Hunt and Doug Brubacher. 1999. Detours: Binary Interception of Win32 Functions. In *3rd usenix windows nt symposium*.

[37] IEEE 2014. *Cyber-attack detection based on controlled invariant sets*. IEEE.

[38] Rob Millerb Ishtiaq Roufa, Hossen Mustafaa, Sangho Ohb Travis Taylora, Wenyuan Xua, Marco Gruteserb, Wade Trappeb, and Ivan Seskarb. 2010. Security and privacy vulnerabilities of in-car wireless networks: A tire pressure monitoring system case study. In *19th USENIX Security Symposium, Washington DC*. 11–13.

[39] JSBSim 2009. JSBSim Open Source Flight Dynamics Model. http://jsbsim.sourceforge.net/.

[40] Khurum Nazir Junejo and Jonathan Goh. 2016. Behaviour-based attack detection and classification in cyber physical systems using machine learning. In *Proceedings of the 2nd ACM International Workshop on Cyber-Physical System Security*. ACM, 34–43.

[41] Sanmeet Kaur and Maninder Singh. 2013. Automatic attack signature generation systems: A review. *IEEE Security & Privacy* 11, 6 (2013), 54–61.

[42] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P Godfrey. 2012. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 49–54.

[43] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. 2018. Securing Real-Time Microcontroller Systems through Customized Memory View Switching. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS '18)*. The Internet Society.

[44] Fanxin Kong, Meng Xu, James Weimer, Oleg Sokolsky, and Insup Lee. 2018. Cyber-physical system checkpointing and recovery. In *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*. IEEE Press, 22–31.

[45] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, et al. 2010. Experimental security analysis of a modern automobile. In *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 447–462.

[46] Denis Foo Kune, John Backes, Shane S Clark, Daniel Kramer, Matthew Reynolds, Kevin Fu, Yongdae Kim, and Wenyuan Xu. 2013. Ghost talk: Mitigating EMI

signal injection attacks against analog sensors. (2013), 145–159.

[47] Michael A Laurenzano, Mustafa M Tikir, Laura Carrington, and Allan Snavely. 2010. Pebil: Efficient static binary instrumentation for linux. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*. IEEE, 175–183.

[48] Lennart Ljung. 1991. Issues in system identification. *IEEE Control systems* 11, 1 (1991), 25–29.

[49] MATLAB 2017. System Identification Toolbox - MATLAB. https://www.mathworks.com/products/sysid.html.

[50] Mario Milanese and Gustavo Belforte. 1982. Estimation theory and uncertainty intervals evaluation in presence of unknown but bounded errors: Linear families of models and estimators. *IEEE Transactions on automatic control* 27, 2 (1982), 408–414.

[51] Military.com 2018. Drones | Military.com. http://www.military.com/equipment/drones.

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

[53] Robert Mitchell and Ray Chen. 2015. Behavior rule specification-based intrusion detection for safety critical medical cyber physical systems. *IEEE Transactions on Dependable and Secure Computing* 12, 1 (2015), 16–30.

[54] Yilin Mo and Bruno Sinopoli. 2010. False data injection attacks in control systems. (01 2010).

[55] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices* 42, 6 (2007), 89.

[56] Katsuhiko Ogata and Yanjuan Yang. 2002. *Modern control engineering*. Vol. 4. Prentice hall India.

[57] Open Source Robotics Foundation 2014. Gazebo. http://gazebosim.org/.

[58] Young-Seok Park, Yunmok Son, Hocheol Shin, Dohyun Kim, and Yongdae Kim. 2016. This Ain't Your Dose: Sensor Spoofing Attack on Medical Infusion Pump.. In *WOOT*.

[59] Jonathan Petit, Bas Stottelaar, Michael Feiri, and Frank Kargl. 2015. Remote attacks on automated vehicles sensors: Experiments on camera and lidar. *Black Hat Europe* 11 (2015), 2015.

[60] Lee Pike, Pat Hickey, Trevor Elliott, Eric Mertens, and Aaron Tomb. 2016. Trackos: A security-aware real-time operating system. In *International Conference on Runtime Verification*. Springer, 302–317.

[61] Ales Prochazka, NG Kingsbury, PJW Payner, and J Uhlir. 2013. *Signal analysis and prediction*. Springer Science & Business Media.

[62] PX4 Dev Team 2017. Open Source for Drones - PX4 Pro Open Source Autopilot. http://px4.io/.

[63] Lawrence R Rabiner and Biing-Hwang Juang. 1993. *Fundamentals of speech recognition*. Vol. 14. PTR Prentice Hall Englewood Cliffs.

[64] ROS 2017. ROS.org | Powering the world's robots. http://www.ros.org/.

[65] Grigore Roşu, Wolfram Schulte, and Traian Florin Şerbănuţă. 2009. Runtime verification of C memory safety. In *International Workshop on Runtime Verification*. Springer, 132–151.

[66] Hiroaki Sakoe and Seibi Chiba. 1978. Dynamic programming algorithm optimization for spoken word recognition. *IEEE transactions on acoustics, speech, and signal processing* 26, 1 (1978), 43–49.

[67] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*. ACM, 298–307.

[68] Qikun Shen, Bin Jiang, Peng Shi, and Cheng-Chew Lim. 2014. Novel neural networks-based fault tolerant control scheme with fault alarm. *IEEE transactions on cybernetics* 44, 11 (2014), 2190–2201.

[69] Yasser Shoukry, Paul Martin, Paulo Tabuada, and Mani Srivastava. 2013. Non-invasive spoofing attacks for anti-lock braking systems. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 55–72.

[70] Sergei Skorobogatov. 2009. Local heating attacks on Flash memory devices. In *Hardware-Oriented Security and Trust, 2009. HOST'09. IEEE International Workshop on*. IEEE, 1–6.

[71] Sooel Son, Seungwon Shin, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. 2013. Model checking invariant security properties in OpenFlow. In *Communications (ICC), 2013 IEEE International Conference on*. IEEE, 1974–1979.

[72] Yunmok Son, Hocheol Shin, Dongkwan Kim, Young-Seok Park, Juhwan Noh, Kibum Choi, Jungwoo Choi, Yongdae Kim, et al. 2015. Rocking Drones with Intentional Sound Noise on Gyroscopic Sensors.. In *USENIX Security Symposium*. 881–896.

[73] Storm Prediction Center, NOAA / National Weather Service 2017. Beaufort Wind Scale. http://www.spc.noaa.gov/faq/tornado/beaufort.html.

[74] The Guardian 2016. First passenger drone makes its debut at CES. https://www.theguardian.com/technology/2016/jan/07/first-passenger-drone-makes-world-debut.

[75] Nils Ole Tippenhauer, Christina Pöpper, Kasper Bonne Rasmussen, and Srdjan Capkun. 2011. On the requirements for successful GPS spoofing attacks. In *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 75–86.

[76] Timothy Trippel, Ofir Weisse, Wenyuan Xu, Peter Honeyman, and Kevin Fu. 2017. WALNUT: Waging doubt on the integrity of mems accelerometers with acoustic injection attacks. In *Security and Privacy (EuroS&P), 2017 IEEE European Symposium on*. IEEE, 3–18.

[77] Zhengbo Wang, Kang Wang, Bo Yang, Shangyuan Li, and Aimin Pan. 2017. SONIC GUN TO SMART DEVICES. *Black Hat USA*.

[78] Jon S Warner and Roger G Johnston. 2002. A simple demonstration that the global positioning system (GPS) is vulnerable to spoofing. *Journal of Security Administration* 25, 2 (2002), 19–27.

[79] Waymo 2017. Waymo (formerly the Google self-driving car project). https://waymo.com.

[80] Chen-Wei Xu and Yong-Zai Lu. 1987. Fuzzy model identification and self-learning for dynamic systems. *IEEE Transactions on Systems, Man, and Cybernetics* 17, 4 (1987), 683–689.

[81] Man-Ki Yoon, Bo Liu, Naira Hovakimyan, and Lui Sha. 2017. VirtualDrone: virtual sensing, actuation, and communication for attack-resilient unmanned aerial systems. In *Proceedings of the 8th International Conference on Cyber-Physical Systems*. ACM, 143–154.

[82] Man-Ki Yoon, Sibin Mohan, Jaesik Choi, Jung-Eun Kim, and Lui Sha. 2013. SecureCore: A multicore-based intrusion detection architecture for real-time embedded systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*. IEEE, 21–32.

[83] Feng Zhu and Jinpeng Wei. 2014. Static analysis based invariant detection for commodity operating systems. *Computers & Security* 43 (2014), 49–63.

[84] Christopher Zimmer, Balasubramanya Bhat, Frank Mueller, and Sibin Mohan. 2010. Time-based intrusion detection in cyber-physical systems. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*. ACM, 109–118.

APPENDIX

A Data collection

Our test generation tool produces random missions with environmental effects. The mission consists of a sequence of operation commands which respect certain happen-before relations (e.g., a quadrotor cannot go to a waypoint before it takes off). Mission generation involves two steps: generating a sequence of operation commands and populating command parameters. We denote a test input as a pair $I = \{t, w\}$ where t is a sequence of operation commands with parameter values and w is the environmental effects (e.g., wind). Algorithm 2 describes the procedure of the profile data collection.

Algorithm 2 Data Collection

```
$Ex \leftarrow$ Executable of target system
$T \leftarrow$ The number of total experiments
$N \leftarrow$ The maximum number of actions in a mission
$M \leftarrow$ Mission state machine
$V_p \leftarrow$ Input range vector of mission parameters
$V_e \leftarrow$ Input range vector of environmental parameters
$D \leftarrow null$                                              ▷ collected data

while $T >= 0$ do
    $(A, Env) \leftarrow GenerateActions(M, N, V_p, V_e)$        ▷ generate a mission
    $trace \leftarrow Execute(Ex, A, Env)$                      ▷ run simulation
    $D \leftarrow D \cup trace$
    $T \leftarrow T - 1$
end while
return $D$

procedure GenerateActions($M$, $N$, $range_p$, $range_e$)
    $cmdlist \leftarrow Traverse(M, N)$                         ▷ command sequence
    $alist \leftarrow GenMission(cmdlist, range_p)$             ▷ a mission
    $env \leftarrow GenEnvEffect(range_e)$                      ▷ environment
    return $alist, env$
end procedure
```

Mission Sequence Generation. A mission is a series of preprogrammed high-level operations. For example, a quadrotor starts a mission with vertical takeoff (*T*) from the home position, and then performs a sequence of operations, such as waypoint (*WP*), loitering (*LO*), and hovering (*HO*). The mission completes when the quadrotor safely lands at the target position with the land (*L*) operation. For each type of vehicle, we construct an operation state machine from our domain knowledge. This is a one-time effort for each type of vehicles. Furthermore, it does not have to be comprehensive as our goal is to derive control invariants that are determined by vehicle weights, shapes, rotor gains, etc. These factors hardly vary and thus we do not need to cover all the possible sequences of operations in order to extract the invariants.

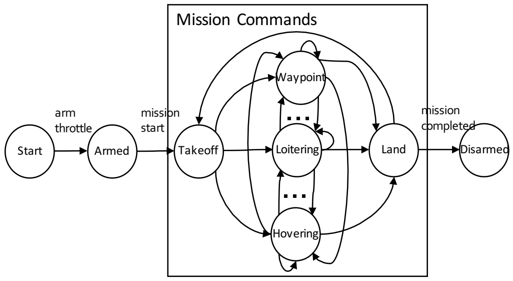


Figure 20: Operation state transition diagram of quadrotor

Table 3: Randomly generated missions.

Test No.	Sequence of operation commands
T1	$T \rightarrow WP \rightarrow LO \rightarrow WP \rightarrow L \rightarrow T \rightarrow LO \rightarrow L$
T2	$T \rightarrow HO \rightarrow WP \rightarrow WP \rightarrow L$
T3	$T \rightarrow LO \rightarrow L \rightarrow T \rightarrow L \rightarrow T \rightarrow WP \rightarrow L$

Figure 20 shows the state transition diagram for a quadrotor, from which we generate sequences of operation commands in a random manner. In particular, our test generator performs random walk of the state machine, starting from the initial state and ending at the final state. Each traversed path represents a mission as in Table 3.

Mission Parameter Value Generation. Every operation command has its own set of parameters. For example, Micro Air Vehicle Communication (MAVLink), the communication protocol between a quadrotor and a ground control station, supports at most 7 parameters for each command (e.g., the *WP* command requires latitude, longitude and altitude as the parameters). Since we test vehicle behaviors within a *virtual fence*, each parameter has a value range. We sample a parameter value from its range following the uniform distribution.

While we run a number of real missions to collect profile data for system identification, we cannot afford running all missions in the real world as it entails a large amount of human efforts. We hence run a lot of missions in simulators as well. We run the simulation in Gazebo [57], a widely-used universal robotics simulation platform. In order to mimic a realistic environment, we further simulate various environmental factors. For example, for drones, we simulate both wind and wind gust effects through Gazebo plugins. Specifically, the *wind* plugin allows to simulate the direction and force of the wind, while the *Windgust* plugin allows simulating other factors, such as direction, duration, force, and start time.

Profile Data Acquisition. Most RVs have their own log module that records runtime operation data. For the missions executed in the real world, real time operation information can be transmitted to the ground station (e.g., via the MAVlink protocol) or stored on its storage (e.g., flash memory). We hence piggy-back the profile data collection on the existing logging components. Details are elided.

For missions executed in simulators, we collect profile data as follows. We set up Gazebo with Robot Operating System (ROS) [64]. ROS is a middleware for robot software development, which provides libraries and tools designed to create robotic applications in modular architecture. Gazebo provides comprehensive simulation environments for a wide spectrum of physical vehicles and third-party control software. Users can also create new vehicles (for simulation) in Gazebo using the Universal Robotics Description Format (URDF) language. The third-party control software (e.g., Ardupilot) communicates with a ground control system via MAVLink. Users can send control messages and missions to the subject vehicle through the ground station during simulation. We implemented a test driver in Gazebo to systematically load and execute the generated missions and collect runtime profile for system identification.