Software-based Realtime Recovery from Sensor Attacks on Robotic Vehicles

Hongjun Choi[1], Sayali Kate[1], Yousra Aafer[2], Xiangyu Zhang[1], and Dongyan Xu[1]

[1]Purdue University
[2]University of Waterloo
[1]{choi293, skate, xyzhang, dxu}@purdue.edu
[2]yaafer@uwaterloo.ca

Abstract

We present a novel technique to recover robotic vehicles (RVs) from various sensor attacks with so-called *software sensors*. Specifically, our technique builds a predictive state-space model based on the generic system identification technique. Sensor measurement prediction based on the state-space model runs as a software backup of the corresponding physical sensor. When physical sensors are under attacks, the corresponding software sensors can isolate and recover the compromised sensors individually to prevent further damage. We apply our prototype to various sensor attacks on six RV systems, including a real quadrotor and a rover. Our evaluation results demonstrate that our technique can practically and safely recover the vehicle from various attacks on multiple sensors under different maneuvers, preventing crashes.

1 Introduction

Robotic Vehicles (RVs) are complex cyber-physical systems (CPS) that continuously change their physical states based on sensor measurements. Specifically, various sensors monitor the current system's physical states and the environment. Based on the measurements, the control components generate actuation signals to control the vehicle for stable operations according to the planned behaviors. RVs, such as drones, ground rovers, and underwater robots [2, 6, 50], utilize multiple sensors of different types. For example, a gyroscope sensor measures angular velocities, an accelerometer measures linear accelerations, a GPS provides geographic position information, and a barometer measures the pressure outside the vehicle which is used for altitude calculation. Unlike the traditional cyber attacks, attackers aiming at RVs can compromise sensor readings through external and physical channels. Since RVs operate based on sensors, the security of RV sensors has become a primary requirement and challenge.

Along with the wide deployment of safety-critical RVs, many physical sensor attacks have been reported recently. For instance, GPS spoofing [51, 54] is a typical physical sensor attack to deceive GPS receiver by injecting incorrect GPS signals. Gyroscopic sensor attack on UAV systems through sound noises [49] can disrupt attitude measurements and lead to crashes. Attackers can manipulate the measurements of MEMS accelerometers via analog acoustic signal injection in a controlled manner [52]. In optical sensor spoofing [8], attackers can acquire an implicit control channel by deceiving the optical flow sensor of a UAV with a physically altering ground plane. Attackers in [47] corrupt automobile's Anti-lock Braking System (ABS) by injecting magnetic fields to wheel speed sensors. In [42], researchers presented remote attacks on camera and LiDAR systems in a self-driving car by introducing false signals with a cheap commodity hardware. These physical sensor attacks pose new challenges because the traditional techniques to protect software are deficient.

To defend the external attacks, many methods have been published recently [5, 22, 26, 36, 37, 57]. However, they only focus on attack detection rather than attack resilience, which is not a complete solution. A canonical counter-measure for attack recovery is to leverage hardware redundancy [29], where critical components are multiplicated to provide attack resilience. For instance, triple module redundancy (TMR) uses three sensors to measure the same physical properties and produces a single output by majority-voting or weighted average. This approach requires additional cost to deploy and sustain the redundant hardware. Additionally, an adversary can still attack multiple sensors as all these sensors are exposed to the same compromised physical environment.

We propose a novel *software sensor* recovery technique for multi-sensor RVs to be resilient to *physical sensor attack*. Instead of using duplicated hardware, our approach uses so-called *software sensors* as the backup of the corresponding physical sensors. Our method can recover when multiple sensors of the same kind or different kinds are under attack. Unlike the transitional physical control systems, the emergence of computationally powerful CPS allows new opportunities to deploy more complex software-based control and recovery components. With these advantages, a software sensor continuously computes and predicts the reading of the corresponding physical sensor. When an attack is detected on some physical sensor(s), the corresponding software sensor(s) allow to isolate and replace the compromised sensors, and recover the system from corrupted internal states to prevent serious attack consequences (e.g., crashes and physical damages). Since our approach is purely software-based, not requiring any additional hardware (e.g., HW duplication and mechanical shielding), it can be deployed not only at design time but also to patch existing systems (e.g., legacy systems).

Specifically, our technique builds a precise state-space model of the vehicle that allows us to predict its physical states (i.e., expected physical behaviors). The model is largely determined by the gravity, control algorithms used, and the physical characteristics of the vehicle (e.g., motor specification, weight and frame shape). We then construct a set of *software sensors*, one for each physical sensor, by transforming the predicted physical states (i.e. the model output) into the appropriate sensor readings using the mathematical conversion equations. In practice, the predicted sensor readings tend to deviate from the real sensor readings due to various reasons. Therefore, to compensate for the intrinsic errors (*conversion*, *model*, and *external errors*), we further develop a number of error correction techniques.

Software sensor readings and physical sensor readings are continuously monitored and compared. In normal operations, both readings are almost identical. Substantial discrepancies indicate that the corresponding physical sensor is under attack. The compromised sensor is hence replaced with its software version. Note that software sensors do not interact with the (compromised) physical environment, thus allowing the vehicle to continue normal operation (for a certain time duration) in the presence of the attacks.

Contribution. Our contributions are summarized as follows.

- We propose a novel software-based technique, *software sensors*, for recovery from various physical and external sensor attacks. This is the first work on the sensor recovery for RVs with the software sensors.
- We address a number of prominent challenges, including how to generate software sensors using system identification; how to recover from individual sensor failures; and how to improve software sensor accuracy considering external disturbances for practical usage.
- We conduct a set of comprehensive experiments on multiple RVs, including simulated RVs and two real ones (a quadrotor and a rover), using various kinds of attacks on one or multiple sensors. The results show that with low overhead, our framework can successfully recover from all the attacks considered for all the RVs, effectively preventing physical damage to the subject vehicles.

Adversary Model. We target physical sensor attacks that maliciously corrupt sensor signals though *external* channels. Additionally, we assume that the attacker can compromise multiple sensors at the same time with different attack techniques,

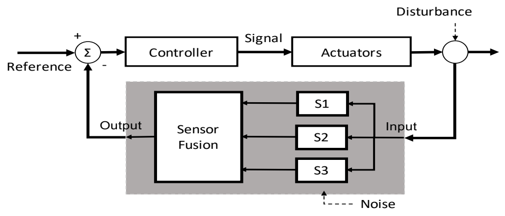


Figure 1: Feedback control loop with sensor redundancy

resulting in disrupted sensor readings. The state-of-the-art attacks (Section 6) can inject both noise or sensor values in a controlled manner. However, we assume that the attacker does not have access to the internal states of subject RV, such as the true sensor readings and the intended navigation plan, thus cannot generate constant deviation (smaller than any predefined threshold). We argue that this is reasonable for the following. (1) We target physical attack channels, for example, acoustic noise (to attack inertia sensors), under which achieving fine-grained manipulation is difficult. (2) To ensure the applied error is consistently smaller than the detection threshold, the attacker needs to have precise estimation of the RV internal sensor readings, in the 400Hz (2.5ms) time unit, which is practically hard by observing external behaviors. Note that while the attacker may use external observation and modeling to estimate RV internal sensor readings when the RV operates normally and has a predictable navigation plan, such estimation becomes infeasible when the navigation plan is not predictable and the RV's internal states have been corrupted by the attack itself. Without precise estimation, the injected error may exceed the threshold and will be detected by our technique (see Section 4.3 for an example).

We do not consider traditional attacks on software or firmware in the cyber domain since those attacks can be effectively handled by existing software security techniques [7,43]. Thus, we assume that our recovery framework – running as part of the control program – is safe against cyber attack vectors.

2 Motivation and Background

In this section, we first introduce control loop with hardware redundancy as background. Later, we use an example to illustrate the physical sensor attack and recovery with the proposed approach. This example simulates a sensor spoofing attack on a real quadrotor by artificially inserting malicious signal data.

2.1 Background

A common control mechanism in RVs shown in Figure 1 is the feedback-loop control which takes system outputs (i.e., current physical state) as the input in the loop. The controller adjusts its control signal to make the vehicle reach the reference state over the loop. Most RVs utilize multi-sensor measurements to obtain a more accurate view of physical state since a single sensor cannot provide reliable data in a real environment (due to sensor noises, possible sensor failure, etc.). In quadrotors, multiple redundant or heterogeneous sensors (e.g., gyroscopes, accelerometers, and GPS) enable the controller to recognize the current physical state and the environment, and then accordingly control motor signals for a stable flight.

Sensor fusion [4] is a very common practice in control engineering. The technique combines multi-sensor data to produce enhanced results. Figure 1 shows typical sensor fusion with the triple modular redundancy (TMR). A single physical property is measured by multiple sensors, and a fusion algorithm combines the redundant information to generate a single output with high accuracy in a competitive way (e.g., voting) or a complementary way (e.g., weighted average). Sensor fusion is not limited to the same type of sensors. Complex sensor fusion algorithms (e.g., extended Kalman filter) often utilize heterogeneous sensor data to reduce uncertainty and produce more accurate measurements. Although sensor fusion can improve accuracy and tolerate failures of a subset of sensors (of a specific kind), it is not effective for physical attacks. For example, the sensor fusion with TMR utilizes the majority voting technique in which, if any one out of the three sensors is compromised or faulty, the other two sensors can identify and mask the faulty one. However, if two sensors (the majority of the sensors) are compromised at the same time, it is difficult to identify which sensors are problematic, which is the Byzantine agreement problem [31]. In case of sensor fusion with the weighted average technique, any single compromised sensor can significantly degrade control performance.

2.2 Motivating Example

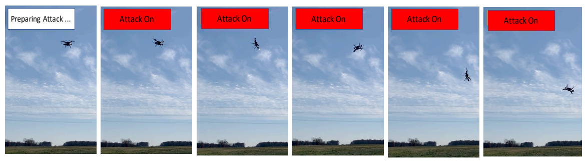


Figure 2: Sequential snapshots from the video of the gyroscope sensors attack (the full video is available at [11]).

Sensor spoofing attack [8, 30, 41, 47, 49, 51–54] is a popular physical attack on RVs. The adversaries maliciously disrupt sensor measurements by perturbing the physical environment or directly compromising sensor internals with physical means. In our example, we use a real commodity quadrotor, 3DR Solo. The vehicle is equipped with three Inertia Measurement Units (IMU), each including a gyroscope, an accelerometer, and a magnetometer. Among the different sensors, we aim to disrupt multiple gyroscope readings with a simulated acoustic sensor attack, leading to a physical crash. In particular, the attacker intentionally injects acoustic noises at the resonant frequency of the gyroscopes, causing the gyroscopes to generate erroneous angular rates. Here, we assume that the attacker cannot access the internals of the target system, but can know the resonant frequency by investigating the sensors used by a similar system beforehand.

We illustrate our example in three steps: first, we show the actual crash of the quadrotor under the example attack with a video; second, we explain the low-level data flow compromised by the attack using a code snippet; and lastly, we demonstrate how our framework effectively recovers the quadrotor under the attack with a graph of internal state value changes.

Figure 2 shows the video snapshots of the attack consequence. During the stable flight (the first snapshot), the attack, launched from the second snapshot, compromises the gyroscope sensor measurements of the current angular rate, corrupts the attitude, and causes a sudden increase in the attitude angle. Specifically, the attack corrupts the roll rate to 0.8 rad/s, and then the controller incorrectly tries to change the roll rate to -0.8 rad/s, as it "thinks" 0.8 rad/s is the current measurement. Subsequently, in the next snapshots, the quadrotor under the attack turns over and crashes.

```
1  main_loop() {
2
3      // determines vehicle states
4      angles = read_AHRS();
5
6      // generates target values
7      targets = navigation_logic();
8
9      // generates actuation signal
10     inputs = attitude_controller(targets, angles);
11
12     // sends signals to actuators
13     motor.update(inputs);
14 }
15 read_AHRS() {
16
17     // read IMU sensor measurements
18     for(i=0; i<num_gyro; i++) {
19         gyros[i] = gyro_sensors[i].read(); // *attack*
20
21         // *inserted code for attack recovery*
22         if(abs(soft_gyro[i] - gyros[i]) > k)
23             gyros[i] = soft_gyro[i];
24
25         // weighted sum
26         gyro += w[i] * gyros[i];
27     }
28     // return angles
29     angles = convert2angle(gyro);
30     return angles;
31 }
```

Figure 3: Control loop and sensor reading monitor

Next, Figure 3 shows the simplified code snippet in the quadrotor's control program. The function `main_loop` shows the main control loop, which has a typical feedback control loop structure [21]. Especially, `read_AHRS()` shows a fusion process of gyro sensor readings. It acquires the readings of the multiple gyro sensors via the sensor hardware interface at line 19, and consolidates the information by weighted average at line 26 (various algorithms may use different weights). The data is then processed to obtain the angles (i.e., internal state values) which are returned at line 30.

The sensor attack on the gyroscope compromises the angular rate measurements at line 19. Since `attitude_controller()` generates motor inputs based on the angles from `read_AHRS()`, any compromised gyroscope reading would disrupt the entire control loop. For example, in a stable hovering operation, errors between the current and target angles are minimal. However, the attack compromises the current angles causing an instant increase in the errors. The controller then generates motor input to reduce these fake errors, and consequently introduces unwanted maneuvers. Since the compromised sensor cannot provide the actual valid measurements, the error accumulates over loop iterations.

Our Recovery Approach. Motivated by the sample attack, we propose a *software sensor* based defense technique. We first construct a *system model* that models the behaviors of the controller, actuators, vehicle physics and dynamics. Specifically, it predicts the next physical state given the system input (i.e., reference) and the current state. Software sensors do not interact with the physical environment such that they are immune to physical attacks. Instead, they "*measure*" the states produced by the system model. In the closed feedback loop as shown in Figure 4, the software sensors are used as standbys: they work in synchrony with the real sensors, and are prepared to take over any time. The recovery switch determines an attack by monitoring the difference between the real and the software sensors measurements, and replaces the real sensors with the corresponding software sensors in the event of an attack. Additionally, if an attack is transient, the switch determines when the attack ends by continuously monitoring the difference, and switches back to the real sensors. Our design is particularly suitable for handling diverse attack scenarios, e.g., attacking one sensor, two sensors (of the same kind), all sensors (of the same kind), and multiple sensors of different kinds, because it detects the ones that misbehave and replaces them with the software version. In contrast, traditional sensor fusion based fault tolerance techniques [4, 16] (e.g., Extended Kalman Filter (EKF) [27]) rely on real physical sensors, including the compromised ones. Therefore, they have difficulties dealing with attacks that corrupt majority sensors of the same kind.

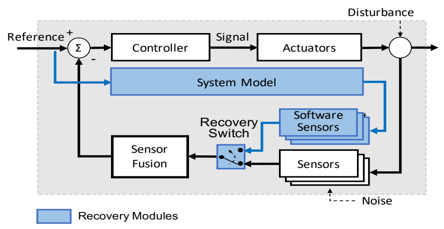
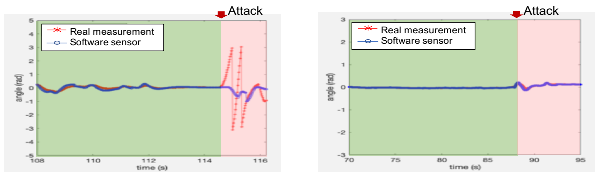


Figure 4: Feedback control loop with our recovery modules

To describe how the recovery modules work, we inserted the attack recovery code at lines 22-23 in Figure 3. The code is placed right after the real sensors readings. At runtime, the code checks if the difference exceeds a pre-defined threshold k, and if so, uses the software sensor measurements instead. The details of software sensor generation and how we distinguish attacks from non-deterministic environmental condition changes will be described in Section 3.2 and Section 4.2.2.

With our recovery modules, the quadrotor can be recovered



(a) Attack without recovery (b) Attack with recovery

Figure 5: Roll changes under the attack

from the attack. Figure 5 shows the changes of roll angle (one of the attitude angles) under the attack with and without the recovery modules during the same mission. The red (star marker) and blue (circle marker) line show the real and software sensor reading, respectively. Before the attack is launched (green area), both are almost identical. However, after the attack, the roll angle is dramatically increased without any recovery action in Figure 5a, whereas with the recovery module in Figure 5b, the software sensor masks and replaces the compromised real sensor measurement. Thus, the recovery modules enable the quadrotor to maintain stable attitude.

Technical Challenges. We should address several prominent technical challenges to use the approach in practice on multi-sensor vehicles: we need to (1) efficiently generate multiple sensor predictions to recover from multi-sensor attacks; (2) consider intrinsic errors such as model inaccuracy and external disturbances (wind, noises, etc.); (3) isolate the specific sensor under attack in time not to propagate corrupted measurements to the vehicle's internal; (4) set proper recovery parameters such as the recovery switch threshold.

3 Design

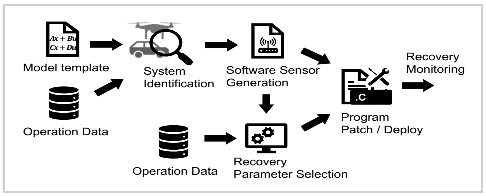


Figure 6: Overview of our recovery framework

Figure 6 presents a high level work-flow of our proposed recovery framework. Each kind of RV, such as quadrotor, hexrotor, and rover, has the same system model template, for example, a polynomial with a specific order and unknown coefficients. The system models of different vehicles (of the same kind) can be considered as various concrete instantiations of the template, that is, polynomials with concrete coefficients. Hence, as the first step of our technique (Section 3.1), given a model template and operation data (i.e., state logs) for a target RV, we leverage system identification [32], a widely used technique to derive system models for the RV. Intuitively, one can consider that it is a training procedure to derive the unknown coefficients such that the model behaviors have minimal errors with the operation log. These coefficients are jointly determined by the RV's physical attributes (e.g. weight and shapes), its control algorithm, and the laws of physics. Once the system model is derived, the framework constructs software sensors (Section 3.2) that operate on model responses. Mathematically, these software sensors are also polynomials that take the physical states predicted by the model as input and produce the corresponding sensor readings. For example, the framework employs the model's angular velocity states to predict the gyroscope sensor's measurements, thus creating a software-based gyroscope sensor; the reading of air pressure sensor is derived from the altitude prediction of the system model. Software sensor is an approximation and has inherent errors (Section 3.3). Such errors accumulate over time (*drifting*). Hence, we synchronize the predicted states with the real states periodically. Also, our recovery switch utilizes the historical (i.e., accumulated) errors to prevent false alarms and to limit the impact of stealthy attacks by using a small time window. In the next step, we determine the appropriate time window size. The window size is RV specific and hence requires analysis. In addition, we determine the threshold for the recovery switch, which is RV specific and sensor specific. Finally, the framework patches the original control program by inserting the recovery code (Section 3.4) right after sensor reading acquisition.

3.1 System Model Generation

Operation Data Pre-processing. To generate a system model, i.e., a mathematical model reflecting RV's behavior, we first collect a large corpus of input and output data of the target RV under normal operations; where inputs are the target states, and outputs are the perceived states for the given inputs over time. For the derivation of accurate model, we collect and pre-process the data as follows: First, the data is collected under different maneuvers to appropriately capture various control properties and dynamics. Our mission generator produces random missions systematically based on Mavlink [35] commands. However, since we constrain the model with a template known a priori, the amount of data needed by our approach is much less than an ML-based learning approach [26,46] – only those involved in the template are needed. Second, the data is collected at a high sampling rate to adequately reflect the highly reactive behavior of the RV to the surrounding physical environment. However, as the log system uses substantial system resources for saving values to memory (e.g., flash card or disk), the typical log update rates are lower than the control loop frequency with a limited number of variables. Specifically, various RV components have different update frequencies - e.g., 400Hz sampling rate for critical sensors, 100Hz sampling rate for non-critical sensors and RC modules, and 10Hz update rate for the RV's own log module. As such, aligning these different data streams is a prominent challenge. To address the problem, we convert various data streams to the same target frequency using a resampling technique. It interpolates new data points within the range of existing sample points by minimizing overall curvature, resulting in a smooth line that passes the existing samples. Here, we use spline interpolation, to avoid Runges' phenomenon [3] which causes oscillation in high degree polynomials. Offline resampling enables us to obtain high frequency data without additional runtime overhead.

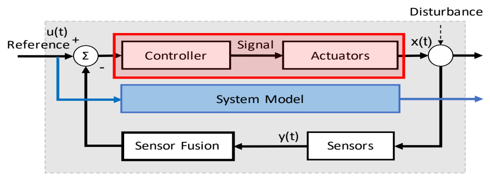


Figure 7: System model in closed loop

Model Construction. The template of an RV's system model consists of the state and output equations, i.e., Eq. (1) and Eq. (2), respectively:

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

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

where $u(t)$ is system input (i.e., the target state as shown in Figure 7), and $y(t)$ is system output. Output $y(t)$ is measured by sensors. The model specifies how the physical states $x(t)$ of the system respond to external inputs and control signals with the underlying control algorithm and system dynamics. As shown in Figure 7, the system model (in the blue box) can be considered a counterpart of the combination of control algorithm, actuators, and vehicle dynamics (in the red box). We leverage the system identification (SI) technique [32] for deriving the system model, which is widely used in different applications [5, 56]. Given the model template (Eq. (1), (2)) and a large set of collected operation data, SI instantiates the A, B, C, D matrices so that the resulting equations produce the best fit for the data. We use the SI Toolbox by MATLAB [34]. Note that the system model is not a software sensor. The output of the model should be accordingly converted to the individual sensors with our on-the-fly operation and error correction technique.

Example. For a quadrotor system, we can generate the models for individual state variables, defined by the following:

$$x = \begin{bmatrix} x & y & z & \phi & \theta & \psi & \dot{x} & \dot{y} & \dot{z} & p & q & r \end{bmatrix} \tag{3}$$

where $[x\ y\ z]$ is the position vector, $[\phi\ \theta\ \psi]$ is the attitude angles (roll, pitch, yaw), $[\dot{x}\ \dot{y}\ \dot{z}]$ is the vehicle velocity, and $[p\ q\ r]$ is the vehicle angular velocity. For each variable, we first determine the state and output template equations. Specifically, we use a discrete-time state-space model template, encoding a PID controller and dynamics equations known a priori for the family of the subject RV. Then, for each variable, we specify a model order (i.e., the degree of polynomial equations). We then employ SI to instantiate the unknown coefficients of the template using an iterative prediction error minimization algorithm [32]. SI in our technique is not limited to the linear state-space modeling. A non-linear model can also be derived from the known template. However, for our purpose (e.g., rigid body RVs), the linear-model is sufficient to approximate the actual higher-order close-loop dynamic, since the dominating system dynamic is a second-order system. Even for an advanced non-linear control algorithm, the control effort is mainly from the linear portion, namely proportion, derivative, and integral [24, 55].

We note that our model construction is generic, since the same model template can be used to instantiate the models for a family of vehicles with a similar physical structure. Besides, our methodology is efficient. Given the profile data and known model template, SI can optimize the coefficients and derive a state-space model that accurately predicts the next states with reasonable computation time (Section 4.2). Our model construction is different from most SI applications in control systems, which often focus on modeling the vehicle dynamics, whereas ours models both dynamics and control algorithm.

3.2 Software Sensor Construction

Software sensor is a software-based virtual sensor which generates the prediction of the corresponding real sensor measurement. It predicts real sensor reading based on the system model output, i.e., the predicted new physical state. Since the physical state prediction is completely model-based, software sensors are independent of real sensor measurements that are vulnerable to physical attacks. Specifically, at runtime, software sensor readings are compared with real sensor measurements. Once an attack is detected on some physical sensor (i.e., its real measurement differs significantly from the predicted one), the corresponding software sensor is used to replace the real one. An RV often has many kinds of physical sensors. Their software version may require non-trivial derivation from the system model outputs. In the following, we explain the mathematical conversions entailed by software sensors.

Conversion Operation. We provide the conversion operations for various sensors (accelerometer, gyroscope, barometer, magnetometer, GPS).

An accelerometer measures linear acceleration of the vehicle. However, the outputs of our example model contain only 12 states that do not directly include acceleration information. Therefore, a conversion operation (Eq. (4)) is required.

$$a(t) = c_k \frac{v(t) - v(t-k)}{k \cdot \Delta t} \tag{4}$$

where v is the velocity, Δt is the sampling time interval (temporal distance between two samples), c_k is a constant coefficient and k is the number of equidistant sample points; k is usually much larger than 1 to tolerate the noise induced by high frequency sampling.

A gyroscope measures angular velocities which are critical in maintaining stable movement, especially for aerial vehicles. To obtain accurate measurements, the gyroscope in IMU operates with a high sampling rate. Other sensors further help to correct gyroscope sensor errors to estimate accurate orientation state (i.e., attitude angles). Gyroscope intrinsically has *drift error* over time due to an integration operation over angular velocities for obtaining angles. In the recoverability test of our approach under the different combinations of attacks on multiple sensors (see Section 4.2.2), gyroscope sensor is the most sensitive and requires accurate prediction for recovery. In this case, it turns out that using software gyroscope alone is not sufficient (leading to reduced stability and operation time) when all the physical gyros are compromised. As such, we introduce a compensation approach to improve accuracy by leveraging other types of real sensors. The details are shown in Section 3.3.

A barometer measures atmospheric pressure, which is necessary to determine altitude. We use Eq. (5) to calculate air pressure from altitude (position z in the system model states).

$$P_h = P_0 \cdot exp\left[\frac{-g_0 \cdot M \cdot (z - h_0)}{R \cdot T_0}\right] \tag{5}$$

where P_0 is the base air pressure (Pa), g_0 is the gravitational acceleration ($9.87 m/s^2$), M is the molar mass of Earth's air ($0.02896 kg/mol$), h_0 is the base altitude, R is the universal gas constant ($8.3143 N \cdot m/mol \cdot K$), T_0 is the base temperature (K), and z is the current altitude from the model states.

A magnetometer, also known as compass, measures the strength of the Earth's magnetic fields in 3-axis, which is used to calculate orientation (heading) information. The following equation shows the transformation of the magnetic fields to orientation status (i.e., the heading direction of the RV):

$$\begin{aligned} H = atan2(&-m_y \cdot cos\phi + m_z \cdot sin\phi, \\ &m_x \cdot cos\theta + m_y \cdot sin\theta \cdot sin\phi + m_z \cdot sin\theta \cdot cos\phi) \end{aligned} \tag{6}$$

where H is the heading direction yaw, and m_x, m_y, m_z are the magnetic field measurements along each axis. Control systems do not directly use the magnetic field measurements, but rather rely on the extracted orientation. Therefore, instead of converting the system model responses to raw magnetic field sensor measurements, we directly use the orientation states from the system model.

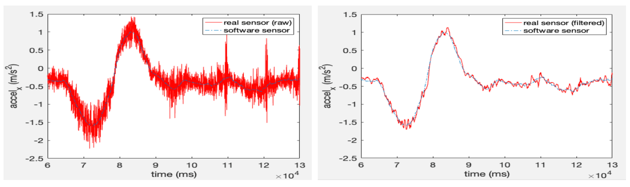
Global Position System (GPS) measures geometric positions and velocities which collectively enhance the position and attitude estimation along with other sensors. GPS measurements can be directly acquired from the system model.

Coordinate System Transformation. Based on the system model responses and conversion equations, we can approximate sensor measurements. Note that internal state variables and sensor measurements may be aligned with different reference frames. Intuitively, each frame can be considered a different coordinate system. Information can be exchanged/aggregated only after they are projected to the same coordinate system [33]. Hence during software sensor conversion, we have to perform frame canonicalization. Specifically, for an RV with a rigid body, we commonly use different reference frames for describing its position and orientation (i.e., pose). The inertial and body frames are used to provide the pose in the global and local coordinate systems, respectively. The inertial frame is an earth-fixed frame, whereas the body frame is aligned with the vehicle's body (hence the sensors). The sensor measurements are usually related to the body frame where the sensors are attached to, and must be converted from the body frame to the inertial frame and vice versa. Frame conversion is accomplished by multiplying with constant conversion matrices. The detailed equations are in Appendix A.

3.3 Error Correction

Software sensors aim to closely predict real sensor measurements. However, the errors between software and real sensors are intrinsic for the following reasons: (1) the conversion (from model states to sensor readings) introduces *conversion errors*, (2) the system model provides only an approximation of the real states and hence introduces *model errors* over time, (3) external disturbances and noises affect the accuracy of model prediction, which introduce *external errors*.

Obtaining an accurate prediction model - thus avoiding the above errors - through precise modeling of complex real-world effects for a specific system is neither practical nor generic. Instead, we choose to tolerate model inaccuracies through integrating additional error correction techniques to compensate for the errors. Note that our recovery does not aim to replace the real sensors permanently when the attack is continuous, but rather aims to isolate the compromised sensor and provide the needed feedback to the control loop for a certain time duration so that we can ensure continuous stable operation for some time without catastrophic consequences (e.g., immediate crashes) or take an appropriate emergency action (e.g. safe landing).



(a) Raw measurement (b) Filtered measurement

Figure 8: Raw and filtered acceleration measurements with software sensor output

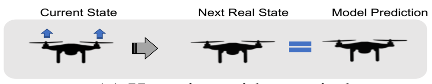
Conversion Errors. Although the concept of comparing software sensor readings and real readings is straightforward, direct comparison is problematic. Specifically, on the software sensor side, higher-order state variables (e.g, acceleration) may contain noise at high-frequency. The conversion process introduces additional errors. As such, directly comparing such software sensor readings with the real ones leads to numerous false alarms. To address the problem, we leverage error reduction techniques [40]. Specifically, to mitigate output inaccuracy caused by numerical differentiation, we can use a simple finite differentiation method like Eq. (4). However, with this method, the output tends to be close to zero in the presence of high frequency noise. To tackle this, we implement a smooth noise-robust differentiator that provides noise suppression [25].

On the real sensor side, raw measurements have various kinds of errors, such as noise, bias, and time lags. Figure 8a shows that a highly fluctuating raw signal is not ideal for comparison with the software sensor signal. Therefore, we smooth it out with a basic filter (see Figure 8b). Specifically, we apply the low-pass filter [40] which is a standard filter to attenuate high frequencies with a pre-selected cutoff.

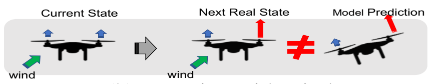
Model Errors. The system model approximates the real RV states. Such approximation contains intrinsic *model error*. This is because the model is constructed from a universal template (for a family of vehicles), which does not describe the details and nuances of a concrete RV. In addition, the model assumes a simple linear PID controller whereas real RVs may use non-linear control algorithms. To mitigate intrinsic model errors, we introduce periodic synchronization and error reset. Although model errors are marginal at any time instance, they tend to aggregate overtime, namely *prediction drift*. Thus, errors should be corrected periodically.

Specifically, our solution regularly resets software sensor errors, by synchronizing with the real sensor readings to remove prediction drift during normal operations. To reset errors, we partition the entire operation duration to small time windows of a fixed duration and synchronize the software sensor readings with the real ones at the start of each window. Note that the synchronized readings are then fed to the system model, eliminating errors in the predicted system states.

Recovery Parameters. We select the recovery parameters (i.e., window sizes and recovery thresholds) systematically. The window size (N) for historical error is an important parameter. If N is too large, there can be a significant accumulation of the error which could cause false alarms. Conversely, if N is too small, the synchronization of the software sensors with the real sensors will be so frequent that it would lead to false negatives. Moreover, the conversion might introduce a small delay in the generation of software sensor measurement, causing it to not align with the real sensor. Therefore, to achieve the measurement synchronization at correct timesteps, N should be more than a potential time-displacement between the software and real sensor signals. We choose N to be the maximum time-displacement computed from the large set of operation data using the dynamic time-warping algorithm [45] that computes the optimal alignment of two data sequences. Once the window size is determined, we calculate the maximum error between two signals within each window in the large set of operation data. We select the threshold $T = e_{max} + m$, where e_{max} is the maximum accumulated error and m is a margin parameter. For example in the 3DR Solo quadrotor, the main control loop is invoked at every 2.5ms and we use the 575ms (i.e., 230 loop counts) as the window size which is chosen as described above. We evaluate the effect of different recovery parameters in Section 4.2.



(a) Hovering with no wind



(b) Hovering with wind

Figure 9: External force and *external error* (state discrepancy)

External Errors. So far, our technique does not model external forces that may introduce errors. For example, when the wind speed is 5 mph west, it introduces external forces that move the aerial vehicle to the east. To adapt software sensors to external disturbances, we calculate an estimation of external force by measuring errors within the previous window. The key observation is that external forces are likely static across two consecutive windows in practice as long as the wind is not drastically changing. Specifically, the time unit is 400Hz (2.5ms), and the window is 2Hz (500ms) less. Within 0.5s, the forces are highly likely unchanged.

Based on our observation, we calculate the average error by comparing the real state and model prediction within each window. Figure 9 illustrates a simple external error. The quadrotor maintains a stable attitude during hovering. Without wind (a), the real next states and model prediction are the same since both cases are affected only by thrust force (the blue up-arrow). However, with the wind (b), to maintain a stable state without tilting, the controller increases the right thrust (in red) such that the drone does not tilt right, whereas the model (without the wind force) thinks the vehicle tilts left with the increased right thrust. We use the average error from the previous window as an estimate of external forces in the next window. The correction result can be found in Section 4.2.2.

Supplementary Compensation. Certain sensor types and usage scenarios require very accurate measurements. As such, using software sensors alone may not be sufficient, especially for lengthy operations. To increase an accuracy of these special sensors and to extend the operation time under the recovery mode, we employ additional error correction techniques. Specifically, we leverage other types of sensors - less sensitive ones - to reduce the estimation errors. Under this approach, we can provide real sensor readings in model prediction. Since model prediction is based on the model and real input, the real sensor measurement (converted from different sensors) would contain more realistic feedback with real disturbances factored in. We present an example of this approach that estimates angle status from the accelerometer and magnetometer (where angle status is typically measured by gyro) in Appendix B. Note that this compensation approach is not necessary for majority of the sensors. In most cases, using our software sensors for recovery - without other real sensors - is sufficient. In our evaluation (Section 4.2.2), among all the studied scenarios, we leverage this technique only when all the gyros are compromised at the same time, which is rare.

Algorithm 1 Runtime Recovery Monitoring

```
 1: $u$ control input of the real vehicle
 2: $m$ sensor measurement
 3: $x$ control states of the real vehicle
 4:
 5: procedure RecoveryMonitor(u, m)
 6:     $y \leftarrow C \cdot x + D \cdot u$                         ▷ calculates model response
 7:     $x \leftarrow A \cdot x + B \cdot u$
 8:     $m \leftarrow filter(m)$
 9:     $m_s \leftarrow convert(y)$
10:     $t++$
11:     if $!recovery\_mode$ && $t > window$ then                  ▷ checkpoint
12:         $t \leftarrow 0$
13:         $r \leftarrow 0$
14:         $e \leftarrow error\_estimation(r, m, m_s)$
15:         $m_s = m$
16:     end if
17:     $m_s \leftarrow m_s - e$                                    ▷ error compensation
18:     $r \leftarrow r + |m - m_s|$
19:     if $r > T_{on}$ then                                        ▷ checks residual
20:         $recovery\_mode \leftarrow true$
21:         $safe\_count \leftarrow 0$
22:     end if
23:     if $recovery\_mode$ then
24:         $m \leftarrow m_s$                                      ▷ recovers sensor
25:         if $r < T_{off}$ then
26:             $safe\_count++$
27:         end if
28:         if $safe\_count > K$ then                               ▷ switches back
29:             $recovery\_mode \leftarrow false$
30:         end if
31:         $recovery\_action()$                                    ▷ optional action
32:     end if
33: end procedure
```

3.4 Recovery Monitoring

Algorithm 1 describes our proposed recovery procedure. The `recovery_monitor()` function is inserted right after the sensor reading code in the main control loop. It takes runtime inputs and actual sensor measurements as the parameters. It then computes the predicted new state (x) and output (y) from the previously predicted states (line 6 and 7). The real sensor measurements (m) are first filtered to attenuate noises (line 8). The model output is then converted into sensor prediction (m_s) according to the sensor type (line 9). In lines 11-16, when the current time is a checkpoint, that is, the start of a new window, the error (e) is calculated using the function `error_estimation()` that estimates the model and external error (see Section 3.3). Error compensation is applied to the sensor prediction within the window (line 17). At line 18, the cumulative difference (residual r) is computed by comparing the values with the real measurement. If the difference exceeds the recovery threshold T_{on}, then it changes to recovery mode and starts new $safe\_count$ (line 19-22). In the recovery mode, the real sensor measurement (m) is replaced by the sensor prediction (m_s) (line 24). At the same time, the difference is continuously checked by the recovery-off threshold T_{off} (usually, $T_{off} < T_{on}$) and when the difference is smaller than the threshold, the $safe\_count$ is increased (lines 25-27). When the difference is below the threshold for more than K times, we resume using the real sensors, assuming the attack is over (lines 28-30). If there is predefined recovery action, we trigger it through calling the function *recovery_action*.

4 Evaluation

We have developed a prototype that includes a mission generator based on Mavlink [35], a customized log module using Dataflash log system, and a system model construction component implemented in Matlab. The recovery module is implemented in C/C++ and includes the software sensors, recovery switch, error correction modules (i.e., differentiator, low-pass filter and supplementary compensation). The model validation and parameter selection components using the profile data are implemented in Matlab. Additionally, we implemented attack modules to simulate physical sensor attacks that maliciously modify the sensor measurements via a remote trigger at runtime.

4.1 Evaluation Setting

We evaluate our framework with both simulated and real-world RVs, including quadrotor, hexarotor, and rover. Table 1 shows the subject vehicles. We first evaluate the effectiveness of our technique under various simulated environmental conditions, since it is difficult to realize different wind effects/conditions in real-world. We then confirm the results with real vehicles including a 3DR Solo quadrotor and an Erle-Rover in real-world conditions.

Table 1: Subject Vehicles in Evaluation

Type	Model	Controller Software	Number of Sensors				
			G	A	M	B	P
Quadrotor	APM SITL	ArduCopter 3.4	2	2	1	1	1
Hexacopter	APM SITL	ArduCopter 3.6	2	2	1	1	1
Rover	APM SITL	APMrover2 2.5	2	2	1	1	1
Quadrotor	Erle-Copter	ArduCopter 3.4	2	2	1	1	1
Rover	Erle-Rover†	APMrover2 3.2	1	1	2	1	1
Quadrotor	3DR Solo†	APM:solo 1.3.1	3	3	3	2	1

* G: gyroscope, A: accelerometer, M: magnetometer, B: barometer, P: GPS
† Real Vehicles

Real Testbed. Our real testbed consists of two commodity RVs: a 3DR Solo [1] and an Erle-Rover [18]. 3DR Solo is a typical commercial quadrotor that leverages heterogeneous and redundant sensors for flight stability. The aerial vehicle is highly dynamic and can be easily affected by environmental factors. The 3DR Solo system is implemented in Pixhawk 2 from the open-source autopilot project Pixhawk [44], and uses APM:Copter, an open-source flight controller based on the MAVlink protocol and part of the ArduPilot project [2]. Erle-Rover is equipped with various sensors and is a representative ground RV. Erle-Rover is implemented with Erle-Brain 3, a linux-based system provided by Erle Robotics. We use the open-source control software APMrover 2 for the rover. Table 2 lists the sensors in 3DR Solo and Erle-Rover. 3DR has 12 sensors and the rover has 6. Note that many sensors are replicated with different hardware to avoid the same type of failures. For example, 3DR has three gyroscope sensors manufactured by different vendors. To compromise the entire set of sensors of the same type, the attacker should have different attack techniques.

Attack and Recovery Setting. To generate the physical sensor attacks discussed earlier (See Section 1), we insert attack modules into the firmware. Since it is difficult to implement the actual hardware attacks which require special devices, we simulate the same effects with the attack modules - but the actual attack does not access internals. Specifically, we add a piece of malicious code into the sensor interface that transmits the sensor measurements to the main closed control loop. The attacks modify sensor measurements (through attack code) to mimic the effect of real "controlled attacks" that control sensor readings (e.g, a sinusoidal wave, random or selected values). Moreover, we consider continuous attacks rather than instantaneous attacks since temporary attacks can be easily recovered by our method. We map Mavlink commands to various attack types to remotely trigger via the ground control.

We say recovery is successful, when after an attack is launched, the technique detects it and triggers the recovery logic to ensure the current states are within a certain error bound of the expected states for a certain period of time:

$$R_{succ} := |Y_t - \bar{Y}_t| \leq \varepsilon, t \in [1...k] \tag{7}$$

where Y_t is the real output, $\bar{Y}_t$ is prediction, ε is the error margin, t is the timestamp in the recovery mode, and k is the maximum time to decide recovery success. For example, $\varepsilon = 3$ and $k = 10$ indicate that a RV performs missions within 3 meters error for 10 seconds under the recovery mode.

Note that our recovery technique does not consider the previous maneuvers (at $t \leq 0$) since our software sensors accurately predict the real measurements in the "various maneuvers" (Figure 12). As long as recovery starts with the accurate initial states via software sensors, subsequent sensor feedbacks are precise and the control loop can obviously control the vehicle to stable states and recover. Also, our goal is to prevent immediate crash and provide the transition time for emergency operation (e.g., manual mode), not to replace the compromised sensor permanently. Therefore, after recovery mode on, the vehicle would conduct stable operation (e.g., hovering) before changing to the emergency operation.

4.2 Experiments and Results

4.2.1 Efficiency

In terms of the space overhead, we measure the firmware size before and after our recovery code is inserted. For the runtime overhead, we compare the execution time of the main control loop before and after. Specifically, we first measure the (space and runtime) cost of the original code as a baseline, which

Table 2: Sensors in 3DR Solo quadrotor and Erle-Rover

Vehicle	Sensors	Manufacturer	Model	Location	Measurement	Data Type	Frequency
3DR Solo	Gyroscope1	InvenSense	MPU6000	Pixhawk 2 (onboard)	Angular Rate	Angular Motion	400Hz
	Gyroscope2	InvenSense	MPU6000	Pixhawk 2	Angular Rate	Angular Motion	400Hz
	Gyroscope3	STMicroelectronics	L3GD20	Pixhawk 2	Angular Rate	Angular Motion	400Hz
	Accelerometer1	Measurement Specialties	MPU6000	Pixhawk 2	Acceleration	Linear Motion	400Hz
	Accelerometer2	InvenSense	MPU6000	Pixhawk 2	Acceleration	Linear Motion	400Hz
	Accelerometer3	STMicroelectronics	LSM303D	Pixhawk 2	Acceleration	Linear Motion	400Hz
	Magnetometer1	Honeywell	HMC 5983	Pixhawk 2	Magnetic Field	Angular Position	100Hz
	Magnetometer2	STMicroelectronics	LSM303D	Pixhawk 2	Magnetic Field	Angular Position	100Hz
	Magnetometer3	Honeywell	HMC 5983	Body (Leg)	Magnetic Field	Angular Position	100Hz
	Barometer1	Measurement Specialties	MS5611	Pixhawk 2	Air Pressure	Linear Position	50Hz
	Barometer2	Measurement Specialties	MS5611	Pixhawk 2	Air Pressure	Linear Position	50Hz
	GPS	u-blox	NEO-7M	Body (Head)	Position	Linear Position	50Hz
Erle-Rover	Gyroscope	Erle Robotics	Erle-Brain	Erle-Brain 3 (onboard)	Angular Rate	Angular Motion	50 Hz
	Gravity Sensor	Erle Robotics	Erle-Brain	Erle-Brain 3 (onboard)	Acceleration	Linear Motion	50 Hz
	Compass1	Erle Robotics	Erle-Brain	Erle-Brain 3 (onboard)	Magnetic Field	Angular Position	10 Hz
	Compass2	u-blox	Neo-M8N	External (roof)	Magnetic Field	Angular Position	10 Hz
	Pressure Sensor	Erle Robotics	Erle-Brain	Erle-Brain 3 (onboard)	Air Pressure	Linear Position	10 Hz
	GPS	u-blox	Neo-M8N	External (roof)	Position	Linear Position	50 Hz

does not include the recovery code. Then, for each sensor, we insert the recovery code including the required libraries that correspond to the sensor (e.g., filters and utility functions) and measure the overhead. Finally, we insert all the recovery code for all sensors to obtain the total overhead (for all the simulated and real RVs).

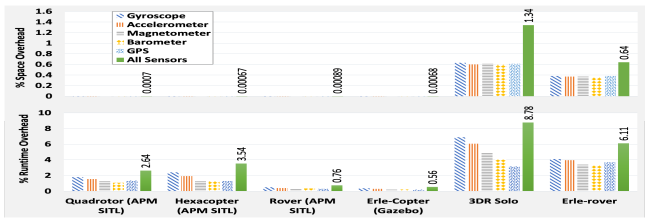
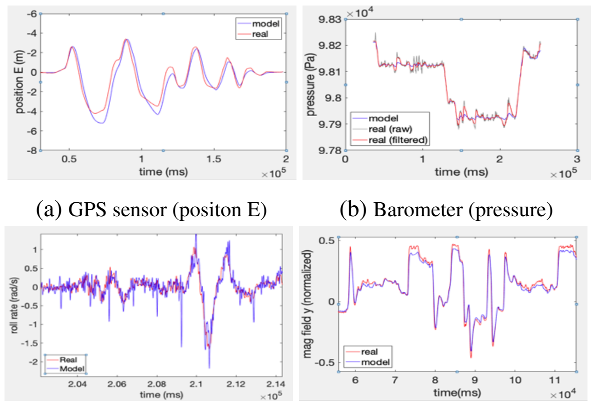


Figure 10: Space and Runtime overhead

Space Overhead. Unlike traditional computing systems, RVs usually have limited memory space. As such, code size is an important performance factor. As shown in Figure 10, the increase of code size (i.e., additional firmware size needed) incurred by our recovery modules is marginal. The space overhead is at most 1.3% when all software sensors are loaded and less than 0.7% for individual sensors. Note that some code pieces are shared across software sensors. The simulated vehicles have negligible overhead since the executables are relatively larger than those of the real vehicles.

Runtime Overhead. We measure the average per-iteration execution time of the main loop which includes various control functions and auxiliary tasks. In ArduCopter and APM-rover2, the system loop execution frequency is 400HZ and 50Hz respectively. Every 2.5ms or 20ms, the scheduler executes the control functions, and then schedules auxiliary tasks using the remaining time in the epoch. Basically, all the tasks should be completed within the hard deadline (i.e., 2.5ms or 20ms). Figure 10 shows the results. The runtime overhead introduced by the recovery module for single sensor recovery is at most 6.9%, whereas, for multiple sensor recovery, the total overhead is at most 8.8%. We also consider the CPU utilization rate (for real vehicles), which is the iteration execution time over the hard deadline. For the 3DR Solo, the rate



(a) GPS sensor (positon E)
(b) Barometer (pressure)
(c) Gyroscope (roll rate)
(d) Magnetometer (normalized)

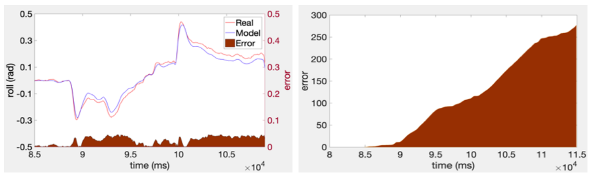
Figure 11: Sensor prediction

increases from 63.32% to at most 67.68% (i.e., by 4.36%) for single sensor recovery, and to 68.88% (i.e., by 5.56%) for multiple sensor recovery. For the Erle-rover, the rate increases from 26.7% to at most 27.8% (i.e., by 0.9%), and to 28.4% (i.e., by 1.7%) for single and multiple sensor recovery, respectively. Note that the observed overhead does not impact normal operations, as the per-iteration runtime does not exceed the hard deadline. Real recovery cases in Section 4.3 demonstrate that our technique is practically effective.

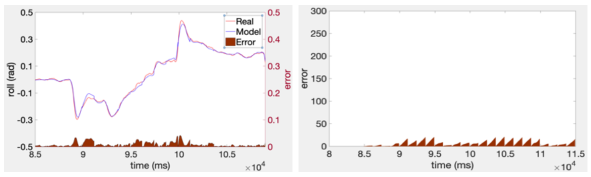
4.2.2 Effectiveness

We evaluate effectiveness as follows. (1) We first show software sensors can precisely predict real sensor measurements under various maneuvers; (2) we show that the error correction techniques can effectively attenuate the prediction errors; (3) we demonstrate that parameter selection is effective; (4) we show that our framework can successfully recover from multiple attacks with real vehicles in real environments; (5) last, we further evaluate our technique under various environmental conditions and attack scales.

Software Sensors. Figure 11 shows how closely software sensors predict (blue lines) the real readings (red lines) in the *various maneuvers* of 3DR Solo. The figures for other systems are similar and hence omitted. It can be observed that there are errors (e.g. drift and external error) between the predictions and the real measurements, which we will remove using the error correction techniques demonstrated below.



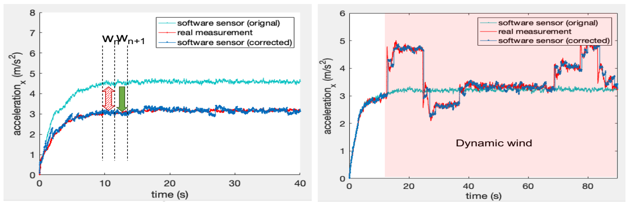
(a) Roll prediction and accumulated errors without correction



(b) Roll prediction and error correction with synchronization

Figure 12: Drift correction with synchronization and error reset

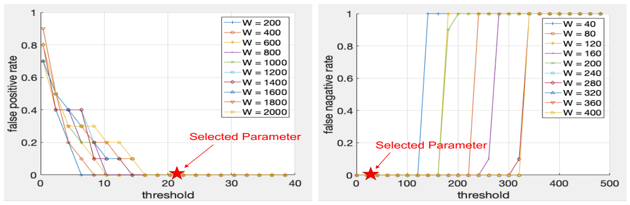
Error Corrections. Figure 12 shows the drift in the roll angle prediction before and after error correction. We measure the roll value and the prediction error during a real flight of the quadrotor (left of (a)). As shown in (b), at each window start, the initial state is synchronized, and the accumulated error is reset. As such, the accumulated error is significantly reduced (right of (b)). In this experiment, we used 1.0s window size with the main sampling rate $T_s = 2.5ms$. The results for other sensors and vehicles are similar.



(a) Constant wind (b) Dynamic wind

Figure 13: External force (wind) corrections for different winds

Figure 13 shows the external force estimation and correction for when there is wind. Here, a simulated APM quadrotor is flying north. We use a simulated RV as we need to create different windy conditions. First, we generate an artificial wind towards the south with a constant velocity 29mph (i.e., a strong breeze in Beaufort scale 6). As the wind pushes the vehicle to the opposite direction, the real acceleration measurement is lower than the software sensor which does not model the wind. We correct this external error in software sensor by subtracting the average error in a window from the following window predictions (see Figure 13a). Similarly, we present the error correction under a dynamic wind - composed of randomly generated wind portions with a random selection of speed ($[10...30]mph$), direction (N,S,E,W) and the portion duration ($[2..10]s$) - in Figure 13b. Observe that software sensors can produce accurate predictions after correction.



(a) False positives rates (b) False negative rates

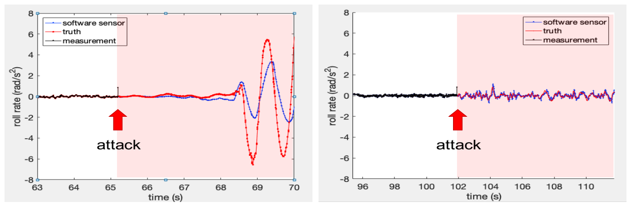
Figure 14: Different recovery parameters and FP/FN rates

Parameter Selection We study the effect of recovery parameters (i.e., window size and recovery switch threshold) on the recovery mode activation. We generate 20 missions (i.e., a sequence of primitive moves like straight fly, turns, etc.) with no attack, to measure the FP rates (i.e. how many times recovery is activated), and 20 missions with the injected attacks to measure the FN rates (i.e. how many times recovery activation is missed). Figure 14 shows the results for different parameter values. Observe that, (1) for a given window, the larger threshold raises less FPs and more FNs; (2) for a given threshold, the larger window causes more FPs and less FNs. Also note that the values chosen by our parameter selection strategy (see Section 3.3) lead to zero FPs and FNs.

Table 3: Attack combination and recovery result

Test#	GPS	Barometer 1 2	Gyroscope 1 2 3	Recovered
C1	Compromised	Benign	Benign	✓
C2	Benign	Compromised	Benign	✓
C3	Benign	Benign	Compromised	✓†
C4	Compromised	Compromised	Benign	✓
C5	Compromised	Benign	Compromised	✓†
C6	Compromised	Compromised	Compromised	✓†

✓: success, †Supplementary Compensation Applied



(a) without compensation (b) with compensation

Figure 15: all-gyroscopes attack recovery and compensation

Multiple Sensor Attacks and Results. We perform combinatorial attacks on heterogeneous sensors of 3DR. For the same type of sensors, we attack the entire set of sensors at the same time. Table 3 shows the results. In this experiment, we observe that the sensors have different sensitivity and importance. When GPS and Barometers are compromised, we can recover the vehicle. However, when all gyros are attacked (C3, C5, and C6), the recovery duration was short (3 sec).

To further investigate the gyroscope recovery case, we perform different attacks on the available gyroscope sensors. Table 4 shows the results and the comparison with a traditional fail-safe mechanism (i.e., TMR). We can recover from

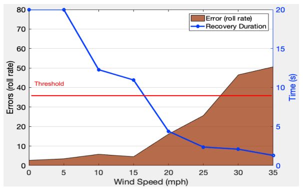
Table 4: Attack on attitude (gyro) sensors and recovery result

Test#	Gyroscope 1	Gyroscope 2	Gyroscope 3	Proposed	TMR
T1	Compromised	Benign	Benign	✓	✓
T2	Compromised	Compromised	Benign	✓	✗
T3	Compromised	Compromised	Compromised	✓[1]	✗

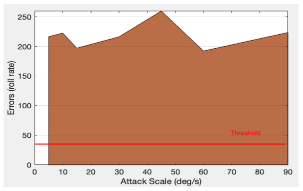
✓: success, ✗: fail to recover, 1: recovery with supplementary compensation

the attack on a single gyro (Test T1) and the attack on the majority gyros (more than half – Test T2) without supplementary compensation. When all gyro sensors are compromised, we can recover from the attack with the complementary approach, leading to increased recovery duration. In comparison, the traditional fail-safe mechanism fails to recover when the majority of gyro sensors are compromised. In general, some RV systems are equipped with failsafe modes (e.g., emergency landing or manual mode). However, those approaches still rely on the remaining benign sensors, or otherwise they undergo immediate crashes even with the failsafe mode.

To increase the recovery duration, we leverage our compensation approach described in Section 3.3. Specifically, to compensate for the accuracy loss, the gyroscope readings are combined with readings from other types of sensors. Figure 15 shows that the internal state (i.e., roll rate) changes during recovery *without and with* compensation. The red curve represents the actual physical state (ground truth roll rate) of the vehicle in real-world. Under attack, the software sensor (blue curve) replaces the real measurement (black curve). However, without compensation, a small error in the software sensor prediction accumulates over time and causes the actual roll rate to oscillate significantly after a few seconds (see Figure 15a). Note that our recovery technique prevents an immediate crash even without compensation applied. Our compensation approach increases the software sensor accuracy by adding the supplementary measurements, leading to a more stable roll rate and a longer recovery duration (see Figure 15b). The video is available at [12]. More discussion of the recovery cases can be found in Section 4.3.



(a) Wind Effects



(b) Attack Scale

Figure 16: Errors under different wind speeds and attack scale

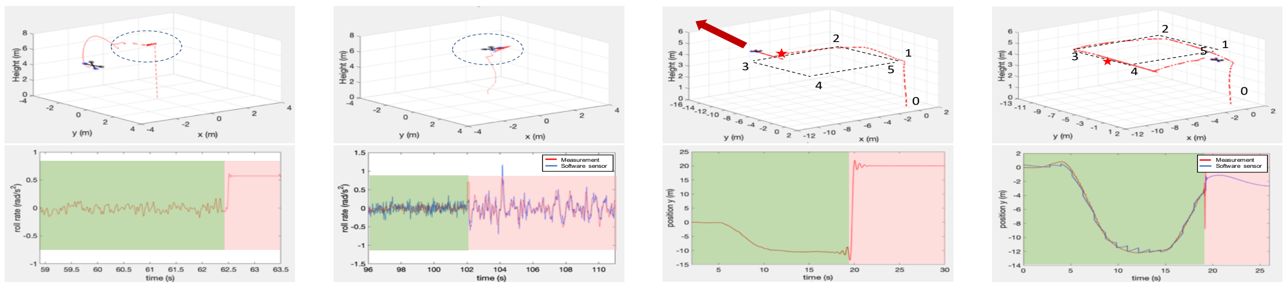
Wind Effects and Attack Scale We evaluate the error (i.e., the difference between software and real sensors) under different wind speeds and attack scales while the vehicle performs missions including straight flies and turns. We also measure the duration of stable operation after recovery, called the *recovery duration*. Specifically, we artificially inject wind in the simulation with different speeds from 0 to $35mph$. For the different attack scales, we change the maliciously injected roll rate from 5 to $90deg/s$. To measure the recovery duration, we find the value of k in Eq. (7) with $\varepsilon = 3$ and maximum $t = 20$. Figure 16a shows that the error (brown area) is small with small (0-$7mph$) and moderate (8-$15mph$) wind, and significantly increases with strong wind (above $20mph$). Observe that the errors are lower than our recovery threshold (=38) with in most case. Only the very strong wind (above $27mph$ - a gale in Beaufort scale) generates errors exceeding the threshold. However, commodity drones are recommended not to fly in wind speed exceeding $22mph$. And, Figure 16b shows the error during the gyroscope attack. Observe that it is significantly larger than the recovery threshold (=38) for all the attack scales. The magnitudes of errors in the two figures demonstrate that the our selected threshold value can distinguish between wind and attack. Lastly, as shown by the blue curve in Figure 16a, the recovery duration is at least 10 seconds with small/moderate wind. When the wind is strong ($> 20mph$), the duration is significantly reduced.

4.3 Case Studies

We present case studies with the two real RVs with four different attacks under various movements: gyroscope and GPS attacks on the 3DR Solo, two GPS attacks on the Erle-rover.

Gyroscope recovery on 3DR Solo. In this attack, 3DR Solo takes off from home and maintains its position at a predefined altitude (i.e., hovering). Then, we launch an attack on its available gyroscope sensors (3 in total). Specifically, we insert constant values (under the attacker's control) to disrupt the gyroscope roll rate measurements. Without our recovery framework, the vehicle instantly deviates from its hovering condition and crashes. Figure 17a show the trajectory in red (top) and roll rate changes (bottom) during the mission. The green region indicates the normal operation without the attack while the red region shows the roll rate changes under the attack. When the attack is launched, the controller uses the compromised roll rate ($\sim 0.6rad/s^2$) and accordingly tries to reduce the rate to match the target state (zero). This conversely decreases the roll rate leading to overturning and consequently crashing the vehicle. With recovery, the software sensor prevents the crash by providing the proper feedback to the control loop under the attack. Figure 17b shows the real trajectory (top) and roll rate changes (bottom) during the attack and recovery. During the normal operation, the measurement (blue curve) takes the real sensor readings whereas under the recovery mode (after the attack), the measurement takes the software sensor (red curve). Observe the software sensor introduces a reasonable amount of oscillation that was not seen under the real benign sensors because of inevitable inherent errors. However, despite the errors, it still allows the vehicle to maintain the hovering position under the attack, preventing the crash. Videos are available at [9].

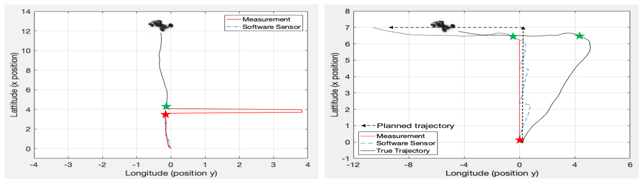
GPS recovery on 3DR Solo. We compromise the GPS sensor reading with a more complex mission. The 3DR Solo performs a waypoint navigation mission where it flies through



(a) Gyro attack (b) Gyro attack recovery (c) GPS attack without recovery (d) GPS attack recovery

Figure 17: Attack and recovery under the sensor attacks on 3dr Solo

five waypoints in a square shape trajectory after take-off. We launch the attack by modifying the longitude positional information of the GPS measurement - set to 20 meters left from the actual position. As shown in Figure 17c, the vehicle deviates from the expected trajectory (black line) and flies to the right (red star), due to the compromised measurements. With recovery, Figure 17d shows that the vehicle continues its planned mission (with a marginal deviation) as the compromised measurements were replaced by the software sensor. Videos can be found at [10] and [15], respectively.



(a) Random Attack (b) Controlled Attack

Figure 18: GPS attacks and recovery on Erle-rover

GPS recovery on Erle-rover. We conduct two different GPS attacks on Erle-Rover: *random* and *controlled* GPS attacks.

In the *random* attack, we inject random signals to compromise positional information (i.e., longitude and latitude) while the vehicle is driving straight. Figure 18a shows the attack (red star). As depicted, right after the attack, the compromised measurements (red curve) is replaced with the software sensor (blue curve), allowing the vehicle to continue its intended trajectory (i.e., straight line). A video is available at [14].

In the *controlled* GPS attack, the attacker maliciously crafts the injected signal based on her/his estimation of the rover's current physical states. In this case, the rover performs a more complex mission in which it moves straight, makes a sharp turn to the left and then drives straight again. During the first straight movement, the attacker injects a signal that closely follows the estimated trajectory but has a small constant error, which misdirects the rover gradually. Specifically, we inject the estimated longitude values with around 0.00001 degrees error, making the attack quite stealthy as it cannot be directly detected. Under this scenario, the vehicle's controller thinks that the vehicle is deviating from the planned mission as the sensed positional measurement are slightly incorrect. To correct the error, the vehicle adjusts its behavior by slightly moving right. To avoid detection, the attacker makes sure that the error is less than our threshold by accurately estimating the changing real states according to the planed move, and also the effect of the attack. However, during the next maneuver (sharp turn), which is unknown to the attacker, she fails to precisely estimate the real states. Consequently, our recovery monitoring successfully detects the attack and activates the software sensor. Figure 18b shows the planed trajectory (black dotted line), the actual trajectory (black line), the measurements (red curve) used in the controller, the software sensor measurements (blue curve), the attack (red star) and recovery (green star). As shown, during the straight move, the attacker maintains small error by estimating the planned states and the vehicle moves right gradually. However, during the turn, the attack is detected. In this case, the software sensor stops the drift to the right, but it cannot compensate the error introduced by the lengthy attack. In practice, additional emergency steps can be taken, such as reboot or estimate GPS location through external channel (e.g., surroundings and nearby RVs). A video of this attack and recovery is available at [13].

5 Discussion

Recovery Duration. The drift during recovery is inevitable since the software sensor is an approximation. Although our recovery technique successfully prevents sensor attacks temporarily, software sensors cannot replace physical sensors permanently or for a prolonged duration due to the drift effect. Our experiment in Section 4.2 shows that the operation time of the recovery mode can be at least 10 seconds in most cases, enough for launching emergency operations (e.g., emergency landing, manual mode switch with an alert) to avoid devastating incidents. The empirical studies in [17, 23] shows that the takeover time to resume control from a highly automated vehicle is around 3 to 7s. Our recovery duration is 10 seconds.

Advanced Attack. Our experiment in Section 4.3 shows that a small-error attack (carry-off attack) can be detected and prevented. The first reason is that the attacker has no access to the internal sensor readings. However, if the attacker can precisely model such readings, she/he may be able to manipulate the error in a way to evade our defense. For example, if the navigation plan of an RV is extremely simple (e.g., straight-line), the attacker can estimate the internal GPS reading even after it is contaminated by the attack through observing the RV's velocity and considering the accumulated errors introduced so far, and accordingly applies the attack continuously until the goal is reached. One way to defend against this is to avoid any predictable navigation plan, for example, by proactively maneuvering the RV in a specific and secret way unknown to the attacker. Second, our detection mechanism utilizes historical error changes rather than an instant error. This approach can limit the stealthy attack. Specifically, the error between real and model states are calculated with accumulated deviation during a certain time duration. This is distinct from a simple bad-data detector or estimator.

6 Related Work

Our approach is inspired by both traditional hardware and software fault-tolerant techniques. The traditional redundancy-based approach [29] can recover a system when *less than half* of the components have a failure. Moreover, hardware replication requires additional hardware costs. There has been a lot of work regarding physical attacks on RVs in recent years. Many external attacks [8, 47, 49, 51, 52, 54] have been proposedAt the same time, corresponding attack detection techniques [5, 20, 22, 26, 28, 36, 37, 57] have been proposed. However, they focus only on attack detection (i.e., significant anomalies) and do not provide a recovery mechanism for continuous operations. As such, the RV may still crash even though it detects the attack.

State estimation [39] has been well researched in control engineering, aiming to improve the accuracy of noisy sensors. Especially, secure state estimation [19, 38, 48] was introduced to handle sensor attacks. However, it mostly utilizes the remaining benign sensors or sensor redundancy to securely estimate system states in the presence of significant noises or attacks. They restrict attackers to corrupt only a subset of sensors in which case the estimation needs to rely only on the benign sensors. In comparison, our approach uses software sensor based on system modeling regardless of the set of sensors under attack, which is practical and generic.

System identification [32] is used to detect attacks [5] and debug RV failures [56]. Similar to our method, they build models for RVs with SI. However, they only detect extreme deviations and cannot provide accurate feedback to the control loop after detection. In addition, we precisely model individual sensor readings while they cannot.

7 Conclusion

We propose a novel sensor attack recovery technique for multi-sensor RVs. The technique uses generic state-space model based software sensors as a safe backup of the physical sensors. Software sensors can precisely predict physical sensor readings while they are largely isolated from the (malicious) environment. Evaluation with real RVs demonstrates our technique can recover from single and multi-sensor attacks.

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] Drone - 3DR Solo, 2017. https://www.3dr.com.

[2] ArduPilot :: Home, 2010. http://ardupilot.org/.

[3] Brian Bradie. *A friendly introduction to numerical analysis*. Pearson Education India, 2006.

[4] Richard R Brooks and Sundararaja S Iyengar. *Multi-sensor fusion: fundamentals and applications with software*. Prentice-Hall, Inc., 1998.

[5] Hongjun Choi, Wen-Chuan Lee, Yousra Aafer, Fan Fei, Zhan Tu, Xiangyu Zhang, Dongyan Xu, and Xinyan Xinyan. Detecting attacks against robotic vehicles: A control invariant approach. In *Proceedings of the ACM CCS*, pages 801–816. ACM, 2018.

[6] Self-driving cars now legal in California, 2012. https://cnn.it/2ZJDnEN.

[7] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security, 1998*.

[8] Drew Davidson, Hao Wu, Robert Jellinek, Vikas Singh, and Thomas Ristenpart. Controlling uavs with sensor input spoofing attacks. In *WOOT*, 2016.

[9] Case study1: Multiple gyroscope sensor (1,2,3) recovery. https://youtu.be/SQ1liFlncB4.

[10] Case study2: GPS sensor attack. https://youtu.be/e8J6ixvtXqQ.

[11] Gyroscope sensor attack and crash. https://youtu.be/_bOLxzvyu5c.

[12] Multiple gyroscope sensor (1,2,3) recovery. https://youtu.be/1MidmMlDEMo.

[13] Rover controlled GPS attack. https://youtu.be/gu1NVssiJls.

[14] Rover random GPS attack. https://youtu.be/S04iL8diRh8.

[15] Case study2: GPS sensor recovery, 2019. https://youtu.be/oC3SOGT_XDY.

[16] W Elmenreich. Sensor fusion in time-triggered systems phd thesis. *Institut fur Technische Informatik, Technischen Universitat Wien*, 2002.

[17] Alexander Eriksson and Neville A Stanton. Takeover time in highly automated vehicles: noncritical transitions to and from manual control. *Human factors*, 59(4):689–705, 2017.

[18] Rover Home — Rover documentation, 2016. http://erlerobotics.com/blog/erle-rover/.

[19] Hamza Fawzi, Paulo Tabuada, and Suhas Diggavi. Secure estimation and control for cyber-physical systems under adversarial attacks. *IEEE Transactions on Automatic control*, 59(6):1454–1467, 2014.

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

[21] Gene F Franklin, J David Powell, Abbas Emami-Naeini, and J David Powell. *Feedback control of dynamic systems*, volume 3. Addison-Wesley Reading, MA, 1994.

[22] Wei Gao and Thomas H Morris. 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):37, 2014.

[23] Christian Gold, Daniel Damböck, Lutz Lorenz, and Klaus Bengler. “take over!” how long does it take to get the driver back into the loop? In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, volume 57, pages 1938–1942. SAGE, 2013.

[24] Jingqing Han. From pid to active disturbance rejection control. *IEEE transactions on Industrial Electronics*, 56(3):900–906, 2009.

[25] Pavel Holoborodko. Smooth noise robust differentiators.

[26] Khurum Nazir Junejo and Jonathan Goh. 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*, pages 34–43. ACM, 2016.

[27] Rudolf Emil Kalman et al. Contributions to the theory of optimal control. *Bol. soc. mat. mexicana*, 5(2), 1960.

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

[29] Israel Koren and C Mani Krishna. *Fault-tolerant systems*. Elsevier, 2010.

[30] Denis Foo Kune, John Backes, Shane S Clark, Daniel Kramer, Matthew Reynolds, Kevin Fu, Yongdae Kim, and Wenyuan Xu. Ghost talk: Mitigating emi signal injection attacks against analog sensors. pages 145–159, 2013.

[31] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[32] L Ljung. System identification-theory for the user, prentice hall, upper saddle river n. *System identification: Theory for the user. 2nd ed. Prentice Hall, Upper Saddle River, NJ.*, 1999.

[33] Kevin M.. Lynch and Frank Chongwoo Park. *Modern Robotics: Mechanics, Planning, and Control*. Cambridge University Press, 2017.

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

[35] Micro Air Vehicle Communication Protocol, 2017. http://qgroundcontrol.org/mavlink/start.

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

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

[38] Yilin Mo and Bruno Sinopoli. Secure estimation in the presence of integrity attacks. *IEEE Transactions on Automatic Control*, 60(4):1145–1151, 2015.

[39] Katsuhiko Ogata and Yanjuan Yang. *Modern control engineering*, volume 4. Prentice hall India, 2002.

[40] Tom O’Haver. A pragmatic introduction to signal processing. *University of Maryland at College Park*, 1997.

[41] Young-Seok Park, Yunmok Son, Hocheol Shin, Dohyun Kim, and Yongdae Kim. This ain’t your dose: Sensor spoofing attack on medical infusion pump. In *WOOT*, 2016.

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

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

[44] Pixhawk, 2019. https://pixhawk.org/.

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

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

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

[48] Yasser Shoukry, Pierluigi Nuzzo, Alberto Puggelli, Alberto L Sangiovanni-Vincentelli, Sanjit A Seshia, and Paulo Tabuada. Secure state estimation for cyber-physical systems under sensor attacks: A satisfiability modulo theory approach. *IEEE Transactions on Automatic Control*, 62(10):4917–4932, 2017.

[49] Yunmok Son, Hocheol Shin, Dongkwan Kim, Young-Seok Park, Juhwan Noh, Kibum Choi, Jungwoo Choi, Yongdae Kim, et al. Rocking drones with intentional sound noise on gyroscopic sensors. In *USENIX Security*, pages 881–896, 2015.

[50] First passenger drone makes its debut at CES, 2016. https://bit.ly/2O0zYft.

[51] Nils Ole Tippenhauer, Christina Pöpper, Kasper Bonne Rasmussen, and Srdjan Capkun. On the requirements for successful gps spoofing attacks. In *Proceedings of the 18th ACM CCS*, pages 75–86. ACM, 2011.

[52] Timothy Trippel, Ofir Weisse, Wenyuan Xu, Peter Honeyman, and Kevin Fu. Walnut: Waging doubt on the integrity of mems accelerometers with acoustic injection attacks. In *EuroS&P*, pages 3–18. IEEE, 2017.

[53] Zhengbo Wang et al. SONIC GUN TO SMART DEVICES. 2017.

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

[55] Bin Yao and Chang Jiang. Advanced motion control: from classical pid to nonlinear adaptive robust control. In *AMC*, pages 815–829. IEEE, 2010.

[56] Enyan Huang Qixin Wang Yu Pei Haidong Yuan Zhijian He, Yao Chen. A system identification based oracle for control-cps software fault localization. In *Proceedings of ICSE*. ACM, 2019.

[57] Christopher Zimmer, Balasubramanya Bhat, Frank Mueller, and Sibin Mohan. Time-based intrusion detection in cyber-physical systems. In *Proceedings of ICCPS*, pages 109–118. ACM, 2010.

Appendix

A Quadrotor Model and Frames

The Figure 19 shows a quadrotor with two frames: inertia and body frame. The linear position of the quadrotor is defined in

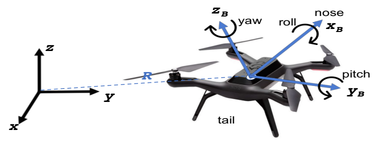


Figure 19: Inertial and body frame

the inertial frame with ξ (x, y, z). The attitude (i.e., angular position) is defined in the inertial frame with η (ϕ, θ, ψ). The roll(ϕ), pitch(θ), yaw (ψ) angle (i.e., Euler angle) determine the rotational angles around the x, y, z axis, respectively. The origin of the body frame is defined in the center of mass of the quadcopter. The linear velocities in the body frame is defined with V_B and angular velocities determined by ω (p,q,r). The rotation matrix R from the body frame to the inertial frame is denoted as:

$$R = \begin{bmatrix} C_\psi C_\theta & C_\psi S_\theta S_\phi - S_\phi C_\phi & C_\psi S_\theta C_\phi + S_\psi S_\phi \\ S_\psi C_\theta & S_\psi S_\theta S_\phi + C_\psi C_\phi & S_\psi S_\theta c_\phi - C_\psi S_\phi \\ -S_\theta & C_\theta S_\phi & C_\theta C_\phi \end{bmatrix} \quad (8)$$

where $S_x = sin(x)$, $C_x = cos(x)$. R is orthogonal thus $R^{-1} = R^T$. The R^T is for rotation from the inertial frame to body frame. The transformation matrix W_η for angular velocities from the inertial frame $\dot{\eta}$ to the body frame ω is:

$$\omega = W_\eta \dot{\eta}, \quad \begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} 1 & 0 & -S_\theta \\ 0 & C_\phi & C_\theta S_\phi \\ 0 & -S_\phi & C_\phi C_\theta \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \quad (9)$$

Also, the Euler angular velocity is then

$$\dot{\eta} = W_\eta^{-1} \omega, \quad \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & S_\phi T_\theta & C_\phi T_\theta \\ 0 & C_\phi & -S_\phi \\ 0 & S_\phi / C_\theta & C_\phi / C_\theta \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix}, \quad (10)$$

B Supplementary Compensation.

We present an example of the supplementary compensation approach that estimates angle status from the accelerometer and magnetometer. Note that angle status is typically measured by gyro. The conversion equation is the following.

$$\begin{aligned} \phi_{acc} &= atan2(y_{acc}, \sqrt{x_{acc}^2, z_{acc}^2}) \\ \theta_{acc} &= atan2(x_{acc}, \sqrt{y_{acc}^2, z_{acc}^2}) \\ \psi_{mag} &= atan2(-y_{mag} \cdot cos\phi + z_{mag} \cdot sin\phi, \\ &\quad x_{mag} \cdot cos\theta + y_{mag} \cdot sin\theta \cdot sin\phi + z_{mag} \cdot sin\theta \cdot cos\phi) \end{aligned} \quad (11)$$

The conversion errors and the real sensor errors (e.g. sensor noise, bias) cause fluctuations in the equations' output. We use low-pass filter to smooth the outputs. Combining the outputs from Eq. (11) and our software sensor (using weighted sum), we acquire more accurate measurements.