SAVIOR: Securing Autonomous Vehicles with Robust Physical Invariants

Raul Quinonez, *University of Texas at Dallas;* Jairo Giraldo, *University of Utah;* Luis Salazar, *University of California, Santa Cruz;* Erick Bauman, *University of Texas at Dallas;* Alvaro Cardenas, *University of California, Santa Cruz;* Zhiqiang Lin, *Ohio State University*

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

This paper is included in the Proceedings of the 29th USENIX Security Symposium.

August 12–14, 2020

978-1-939133-17-5

Open access to the Proceedings of the 29th USENIX Security Symposium is sponsored by USENIX.

SAVIOR: Securing Autonomous Vehicles with Robust Physical Invariants

Raul Quinonez
UT Dallas

Jairo Giraldo
University of Utah

Luis Salazar
UC Santa Cruz

Erick Bauman
UT Dallas

Alvaro Cardenas
UC Santa Cruz

Zhiqiang Lin
Ohio State University

Abstract

Autonomous Vehicles (AVs), including aerial, sea, and ground vehicles, assess their environment with a variety of sensors and actuators that allow them to perform specific tasks such as navigating a route, hovering, or avoiding collisions. So far, AVs tend to trust the information provided by their sensors to make navigation decisions without data validation or verification, and therefore, attackers can exploit these limitations by feeding erroneous sensor data with the intention of disrupting or taking control of the system. In this paper we introduce SAVIOR: an architecture for securing autonomous vehicles with robust physical invariants. We implement and validate our proposal on two popular open-source controllers for aerial and ground vehicles, and demonstrate its effectiveness.

1 Introduction

Autonomous Vehicles (AVs) including aerial, ground, and sea vehicles are becoming an integral part of our life [1]. Unmanned aerial vehicles are projected to have an 11.2 billion dollar global market by 2020 [2] with applications ranging from agricultural management to aerial mapping and freight transportation [3]. Currently, most AVs trust sensor data to make navigation and other control decisions. In addition, they trust that the control command given to actuators is executed faithfully. While trusting sensor and actuator data without any form of validation has proven to be an effective trade-off in current market solutions, it is not a sustainable practice as AVs become more pervasive and sensor attacks continue to mature in their sophistication.

There are two main threats to AV sensors: *GPS spoofing* and *transduction attacks*. GPS spoofing attacks have occurred in real-world systems. For example, several instances of GPS spoofing attacks affecting the navigation of more than 24 vessels in the Black Sea have been reported [4] (experts believe these GPS attacks are anti-drone measures), and while there is debate on whether a foreign nation spoofed a military-grade GPS to capture a U.S. Unmanned Aerial Vehicle [5], launching the same AV takeover attack in commercial GPS systems is quite straightforward [6–9].

Another notable attack against AVs are *transduction attacks* [10], which often inject out-of-band signal to sensors or actuators [11]. Sensors are transducers that translate a physical signal into an electrical one, but these sensors sometimes have couplings between the property they want to measure and the analog signal that can be manipulated by the attacker. For example, sound waves can affect accelerometers and make them report incorrect movement values [12], and radio waves can trick pacemakers into disabling pacing shocks [13]. These attacks have been shown to be effective on AVs by using sound to affect gyroscopes [14], lasers to affect camera sensors in drones [15], lasers to affect lidar sensors in cars [16], and intentional electromagnetic interference to manipulate actuators in drones [17].

Classical security mechanisms such as software security, memory protection, authentication, or cryptography are not enough to protect these cyber-physical systems as transduction attacks represent a new class of attacks that are not effectively handled by classical software security [10]. In order to identify these new attacks, there is growing interest in Physics-Based Attack Detection (PBAD) [18].

PBAD consists of two steps: the first step is performed offline and extracts physical invariants of the system to create a model that captures the expected correlations between sensors (also known as sensor fusion), and between actuators and sensors (i.e., between the inputs and the outputs to the system). The second step is an online anomaly detection algorithm that compares predictions with observed states and raises an alarm when the accumulated discrepancy between predicted and observed states exceeds a threshold.

PBAD has been explored in water control systems [19, 20], state estimation in the power grid [21, 22], chemical processes [23, 24], autonomous vehicles [25], and a variety of other cyber-physical systems [18]. However, one of the key weaknesses of PBAD is that it is vulnerable to stealthy attacks [26]. A fundamental reason for the existence of stealthy attacks is that any control of a physical system would not

need sensors if we knew exactly the physical evolution of the process given the control commands (this is called open-loop control). Meanwhile, almost all control algorithms run in "closed-loop" because model uncertainties and perturbations prevent us from knowing exactly the evolution of a physical process. This uncertainty allows malicious users to create attacks that behave seemingly like the physical process under control, but create a small deviation that over time can be catastrophic. Unfortunately, none of the prior efforts on autonomous vehicles have considered stealthy attacks [25].

Therefore, in this paper we design a new system considering the robustness of PBAD for AVs against stealthy attacks. In particular we design and evaluate the system against sophisticated attackers that can craft worst-case impacts to the system while remaining undetected. In addition, we provide a detailed study of how to implement and evaluate PBAD as a fundamental component for a future security reference monitor for aerial and ground vehicles. We argue that in order to study the role of PBAD for AVs, we need to consider three aspects: (1) algorithms for attack detection, (2) adversary models that include powerful stealthy attackers, and (3) an implementation that shows minimal performance overhead in real-hardware. Correspondingly, we provide contributions in each of these aspects:

1. We provide a detailed study of which physical models are optimal for capturing the behavior of aerial and ground vehicles, and which statistical anomaly detection algorithm works best to detect attacks. Our results show that our algorithms outperform state-of-the-art PBAD tools for AVs (e.g., [25]) by (a) detecting more attacks, (b) detecting attacks *faster*, and (c) having less false alarms than previous proposals.

2. We study in detail *stealthy* attacks against AVs by showing that PBAD tools are never perfect (if we knew the exact behavior of a drone, we would not need sensors), and show how an attacker can leverage this imperfection to launch stealthy attacks. To our best knowledge, no previous work on drones has considered stealthy attacks and we argue in this paper how previous proposals are insecure against PBAD attacks (the attacker can crash a drone without being detected) while our methods are more resilient to this strong type of attacker.

3. We provide a detailed implementation of our system in two popular open-source projects for autonomous vehicles (PX4 and ROS). We also show our implementation in real hardware (Intel Aero drone and Traxxas Ford Fiesta ST Rally Car), showing minimal performance impact. Our source code is openly available at `https://github.com/Cyphysecurity/SAVIOR.git`.

We call our general framework SAVIOR: Securing Autonomous Vehicles wIth rObust physical invaRiants. Our SAVIOR framework consists of the following key insights: (1) use well-known physical invariants, (2) learn the parameters of these invariants via system identification, (3) use change detection algorithms to keep track of historical anomalies, and (4) evaluate PBAD with stealthy attacks in order to find the worst-case performance of our defenses.

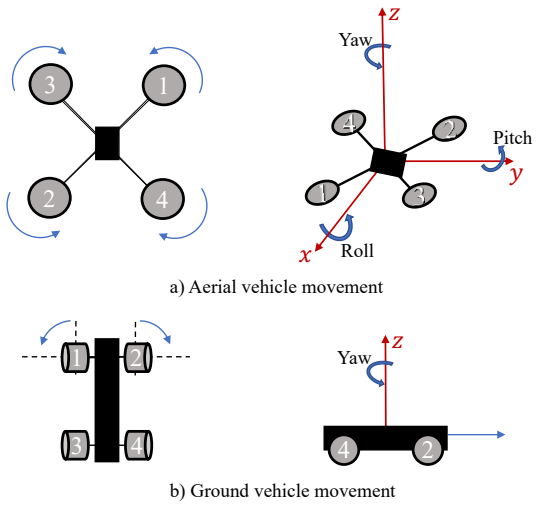


Fig. 1: All vehicles are free to rotate in three dimensions: yaw, pitch, and roll. Ground vehicles can only control their yaw, but IMUs can report pitch and roll if the vehicle is on an inclined plane.

2 Background and Motivation

AVs use a variety of sensors ranging from cameras to GPS and Inertial Measurement Units (IMU). An IMU is a standard component in AVs and includes *accelerometers*, *gyroscopes*, and *magnetometers*. Accelerometers measure the acceleration of a vehicle, gyroscopes measure the angular velocity of a vehicle, and magnetometers act like a compass for the vehicle. A typical configuration includes one accelerometer, one gyroscope, and one magnetometer per axis of the vehicle. The three axes are *pitch* (rotating a vehicle upwards or downwards), *roll* (rotating the vehicle sideways), and *yaw* (rotating the orientation of the vehicle). Examples of these axes for a quadcopter and a ground vehicle are shown in Fig. 1.

2.1 Threat Model

We assume an adversary that can inject false signals in one of the sensors (or actuators) used by AVs. For example, in addition to IMUs, AVs typically use other sensors like GPS receivers for location information, RADARs, LiDARs or ultrasonic sensors to detect nearby obstacles, and cameras. Unfortunately, all of these sensors are vulnerable to transduction attacks including IMU [12, 14, 27], RADAR [28], LiDAR [16, 29, 30], ultrasonic [28], and camera [15, 28, 29] sen-

sor measurements. In addition, GPS signals can be spoofed to hijack vehicles [7–9, 31]. The threat model in our paper is similar to the threat model in all of these previous research efforts.

It is important to note that while GPS and transduction attacks started mostly as denial of service attacks (e.g., [14]), the ability of the attacker to launch these kinds of attacks is improving. Not only can these attacks be launched from longer distances [11], but recent research has shown how GPS spoofing and transduction attacks can achieve a high level of accuracy in the way the attackers can manipulate the sensor signal [11] and GPS takeovers [9].

The level of access for successfully launching these attacks is diverse. It can range from physically placing an actuator next to a sensor, to flying an attack drone near to the target drone to launch these attacks. For example, a ground vehicle in front of the target vehicle can spoof LiDAR signals causing the vehicle to perceive nonexisting obstacles or ignore existing ones [16].

The end result of these attacks is that the sensor signal y is replaced with an attacked signal y^a. In this paper we consider a variety of attacks, including bias attacks, where $y^a = y + bias$ or stealthy attacks, where y^a is selected so that the signal causes damage to the system, but the attack is not detected by a PBAD algorithm.

While the main motivation for our work is the growing sophistication of transduction attacks, our defenses simply assume a signal injection attack, which can also be done through software attacks (malware). The implementation of PBAD against software-based attacks needs to be done as part of a trusted computing base, for example in the kernel of the operating system (assuming the attacker does not have access to it) or at the firmware-layer (again, assuming the attacker cannot change the firmware).

We consider as out of scope attacks that can inject signals to all sensors and actuators. Our attack-detection mechanisms needs to have at least one sensor/actuator combination that can reveal the anomalies injected by the adversary. Fully characterizing the attack detectability of PBAD to signal injection attacks is an active research area [32] and it depends on the nature of the system under consideration and where to capture the sensor and actuator signals, the physical properties of the system, etc.

2.2 Linear Physical Invariants

To detect transduction (or even software) attacks to these sensors (and actuators) there is a growing body of literature on PBAD [18]. PBAD algorithms have two parts: the first part builds a model of the physical invariants of the system and can be done offline. In the second part, an online tool monitors predicted and observed measurements to see if they fit our expectation on the correlations between sensors, and the correlations between sensors and actuators. In this subsection we briefly explain how the first part of PBAD (extracting physical invariants) was done in previous work.

It is possible to represent physical processes in a compact form using matrices and vectors (i.e., a linear system) that indicate the relationship between the control inputs and the system variables. For example, if you have a vehicle with an initial speed of v_0 m/s, the position p_1 after 0.1 sec is dictated by the initial position p_0 plus the change caused by the initial velocity after 0.1 sec, i.e., $p_1 = p_0 + 0.1v_0$. Similarly, if the vehicle has an initial acceleration a_0 m/s^2, the velocity v_1 evolves according to $v_1 = v_0 + 0.1a_0$ by assuming zero friction and aerodynamic drag. Finally, suppose that only the position can be measured at each time instant. These simple systems can be generalized using matrices as follows: Let $x_k = [p_k, v_k]^\top$, $u_k = a_k$, $A = \begin{bmatrix} 1 & 0.1 \\ 0 & 1 \end{bmatrix}$, $B = \begin{bmatrix} 0 \\ 0.1 \end{bmatrix}$.

Since only the position is measured, we define the sensor readings $y_k = Cx_k$, where $C = [1, 0]$ such that

$$x_{k+1} = Ax_k + Bu_k, \quad \text{and} \quad y_k = Cx_k. \tag{1}$$

Equation (1) is known as a Linear Dynamical State-space (LDS) system and is widely used in system dynamics and control. Matrices A, B, C are the system matrices and are unique for each physical process.

Choi et al. [25] recently proposed the use of linear equations to describe the physical invariants of the vehicles. Linear state-space models (like the ones used in their work) are one of the most popular models in control systems because they can capture the dynamics of a wide range of systems and avoid the expensive detailed nonlinear models. However, quadcopters, rovers, and other vehicles have well-known nonlinear physical invariants [33–35]. With a more accurate model of the system, we can expect better attack detection and fewer false alarms in PBAD systems. In the next section we show the general equations describing the physical invariants of any quadcopter, and ground vehicles, but similar equations exist for other AVs such as hexacopters.

3 Designing PBAD for AVs

Fig. 2 gives an overview of how we design our PBAD for AVs. Our design consists of three main components: 1) an offline stage where we learn the parameters of the physical invariants of the AV, 2) an online stage where we use the model we learned offline to predict sensor measurements and compare them to observations (and raise an alert if there is a persistent anomaly), and 3) a definition of stealthy adversaries to help us evaluate the security of our algorithms against sophisticated signal injection attacks.

The pre-processing stage in Fig. 2 uses a **data-fusion** algorithm that combines the gyroscope readings of angular velocities with the accelerometer or magnetometer measurements to calculate the intrinsic bias of the gyroscope and then

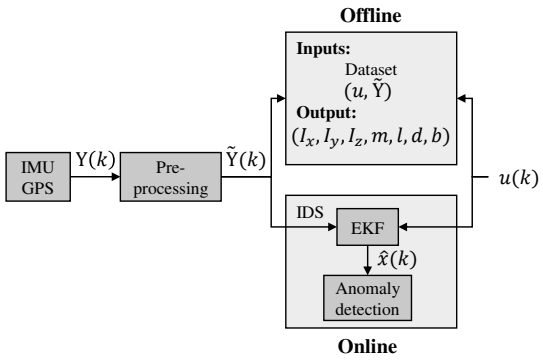


Fig. 2: Our first step is to pre-process sensor data to obtain the states $\tilde{x}$ needed in our nonlinear equations. We then collect a dataset of inputs to drone rotors (u) and outputs (observed states $\tilde{Y}$) to learn the parameters of our nonlinear model. During runtime, we use the model learned to make a prediction using the Extended Kalman Filter (EKF) and compare it to the observed state. We then run an anomaly detection test to see if the differences between what we observe and what we expect is statistically significant over time.

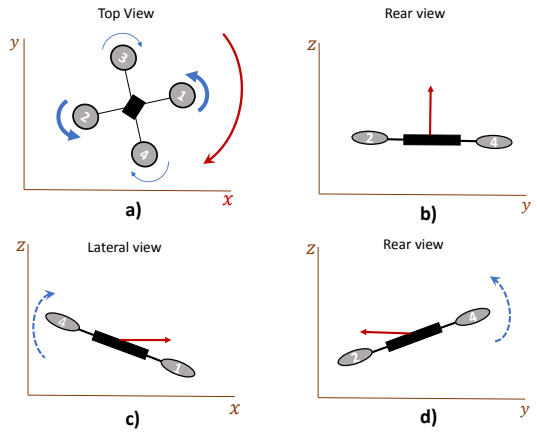


Fig. 3: Movements of a quadcopter: a) yaw rotation is obtained when motors 1 and 2 move faster than 3 and 4; b) vertical lift when all propellers have the same speed; c) forward movement is caused by pitch rotation; d) movement to the left caused by the roll rotation.

generate accurate roll, pitch, and yaw angle readings. The algorithm is based on a simple linear Kalman filter that exploits geometric properties of the accelerometer and magnetometer.

In this section we describe why our specific methods achieve better results than previous work. In particular, (1) we use nonlinear physical invariants, which capture better the model of the system, and (2) we use a better online statistic to keep track of anomalies and raise alerts if necessary. In particular we use a CUSUM statistic, which is based on optimal change detection theory (instead of using fixed time windows), and which allow us to detect attacks faster, and more accurately than previous work.

3.1 Nonlinear Physical Invariants

Choi et al. used linear equations to describe the physical invariants of the vehicles; however, quadcopters, rovers, and other vehicles have well-known nonlinear physical invariants [33–35]. In our experiments, we show why considering linear invariants leads to PBAD systems that are insecure because stealthy attackers can take advantage of this incorrect assumption (linear vs. nonlinear) to launch attacks that can crash the drone or cause other safety problems.

All quadcopters are uniquely defined as having four motors rotating in opposite directions. Motors one and two rotate counter clock-wise and motors three and four rotate clockwise. These motors receive signals from the flight controller to execute different maneuvers such as take off, landing, and route following commands. The quadcopter uses the thrust created by the propellers to rise in a vertical direction when all propellers have the same speed. All the other maneuvers are possible thanks to the roll (move left or right), pitch (move forward or backward), and yaw (change orientation), which correspond to the rotation along the x, y, and z axes respectively. Fig. 1 shows the overall model of the device.

When motors 2 and 4 spin faster than 1 and 3, a tilt along the y-axis (pitch) is achieved causing a forward movement (the opposite will cause the drone to fly backwards), and the velocity of the drone is proportional to the difference between the speeds of the rear and the front propellers (which is also proportional to the pitch angle). Similarly, when the propellers on one side (i.e., 1 and 4) are faster than the other side (2 and 3), a tilt along the x-axis (roll) will cause the drone to fly to the left. Rotation along the z-axis (yaw) is achieved when the rotation speed of diametrically opposing pairs of motors are increased or decreased, varying the torque in the direction of rotation of that pair (recall that diametrically opposing motors in a quadcopter rotate in the opposite direction), causing the quadcopter to rotate in the opposite direction of the increased torque. The four types of movements are summarized in Fig. 3.

The physical invariants of a quad-copter model can be described by 12 nonlinear differential equations that exploit Newton and Euler equations for the 3D motion of a rigid body. These equations keep track of position, speed, angles, and angular speed of the quadcopter.

Six states define the position of the system in the three dimensional space described by the Cartesian coordinate (x, y, z), which points to the center of gravity of the quadcopter. Their time derivative (v_x, v_y, v_z) defines the speed of the center of gravity relative to the earth. Six states define the attitude of the system: Euler angles (θ, ϕ, ψ) represent the roll, pitch, and yaw angles respectively, and their time derivatives $(\omega_\theta, \omega_\phi, \omega_\psi)$ describe the rotation speed of the quadcopter.

The dynamics of the quadcopter are given as follows [34, 35]:

$$\begin{aligned}
&\dot{\phi} = \omega_\phi, && \dot{x} = v_x \\
&\dot{\theta} = \omega_\theta, && \dot{y} = v_y \\
&\dot{\psi} = \omega_\psi, && \dot{z} = v_z \\
&\dot{\omega}_\phi = \frac{U_\phi}{I_x} + \dot{\theta}\dot{\psi}\left(\frac{I_y - I_z}{I_x}\right), && \dot{v}_x = \frac{U_t}{m}\left(\cos\phi\sin\theta\cos\psi + \sin\theta\sin\psi\right) \\
&\dot{\omega}_\theta = \frac{U_\theta}{I_y} + \dot{\phi}\dot{\psi}\left(\frac{I_z - I_x}{I_y}\right), && \dot{v}_y = \frac{U_t}{m}\left(\cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi\right) \\
&\dot{\omega}_\psi = \frac{U_\psi}{I_z} + \dot{\phi}\dot{\theta}\left(\frac{I_x - I_y}{I_z}\right), && \dot{v}_z = \frac{U_t}{m}\cos\phi\cos\theta - g
\end{aligned} \tag{2}$$

where I_x, I_y, I_z are the moments of inertia, m is the mass of the quadcopter, and g is the gravity.

To control the device, a flight controller changes the torque produced by the rotors of the quadcopter. U_ϕ, U_θ, U_ψ are the torques produced by the rotors and U_t is the thrust force. The behavior of the quadcopter is controlled by changing the torques and thrust induced by the rotors velocity. Let Ω_i^2 denote the square of the speed of each rotor $i = 1, \ldots, 4$. Then we have the following relations

$$\begin{cases}
U_t = b\left(\Omega_1^2 + \Omega_2^2 + \Omega_3^2 + \Omega_4^2\right) \\
U_\phi = bl\left(\Omega_2^2 - \Omega_4^2\right) \\
U_\theta = bl\left(\Omega_3^2 - \Omega_1^2\right) \\
U_\psi = d\left(\Omega_1^2 + \Omega_2^4 - \Omega_3^2 - \Omega_2^2\right)
\end{cases} \tag{3}$$

where l is the distance between any rotor and the center of the drone, b is the thrust factor, and d is the drag factor. Notice from equation (3) that the thrust, which dictates the vertical movement, depends on the sum of the velocities of all four rotors. Similarly, forward and lateral movements come from the differences between the speed of the rotors that cause pitch or roll changes, as summarized in Fig. 3.

These equations can be used to model any commercially available quadcopter. There are parameters of the equations that will change from drone to drone. In particular the moments of inertia I_x, I_y, I_z; the mass m; the distance between any rotor and the center of the drone l; the thrust factor b; and the drag factor d. Learning the values of these parameters can be done offline and needs to be done only once per drone.

We can learn all these parameters by using a system identification tool. A system identification algorithm is a machine learning tool used by control engineers to find the values of parameters for their models. In our case we have to learn the values of I_x, I_y, I_z, m, l, b, and d from a dataset of inputs (control actions to the rotors of the quadcopter) and outputs (sensor measurements from IMUs and GPS).

Nonlinear models are also well-known for other AVs. For example, the dynamics of a four-wheel vehicle are described as follows [36]:

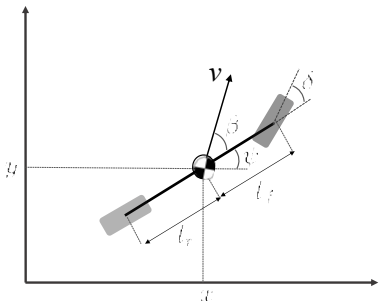


Fig. 4: Ground vehicle bicycle model.

$$\begin{aligned}
\beta &= \tan^{-1}\left(\frac{l_r}{l_r + l_f}\tan(\delta)\right) \\
\dot{x} &= v\cos(\psi + \beta) \\
\dot{y} &= v\sin(\psi + \beta) \\
\dot{\psi} &= \frac{v}{l_r}\sin(\beta) \\
\dot{v} &= a.
\end{aligned} \tag{4}$$

This model describes the interaction between the actuators, which are the steering angle δ and the acceleration a, and the states/sensors, which are the velocity v, the orientation (i.e., yaw angle ψ), and the position x, y, according to Fig. 4.

In the next subsection we will show how to learn the parameters of these two models.

3.2 Offline Learning of Nonlinear Invariants

There are different learning tools for parameter estimation. In our case, we use nonlinear-least squares data fitting [37] which can be summarized as follows: Suppose we have a dataset with input/output data, U/Y, respectively. We have prior approximate knowledge about the physical dynamics of the system in terms of the set of differential equations $F(\cdot)$ with unknown parameters $\mathcal{P} = \{p_1, p_2, \ldots\}$. Given the input/output dataset and the differential equations $F(\cdot)$, our goal is to find the parameters $\mathcal{P}$ that better fit the data. The optimization problem can be formulated as a least-squares problem

$$\min_{\mathcal{P}} \sum_{t=1}^{T} (H_t(\mathcal{P}, U_t) - Y_t)^2$$

where $H_t(\mathcal{P}, U_t)$ is the estimated output at each sampling instant t for the given parameters $\mathcal{P}$ and the input U_t, and it is obtained from the solution of the differential equations $F(\cdot)$. The objective is then to find the set of parameters $\mathcal{P}$ that minimize the least square error between the estimated output $H_t(\mathcal{P}, U_t)$ and the measured output Y_t. This is an optimization problem that requires algorithms such as the Levenberg-Marquardt [38] or the interior-reflective Newton method [39].

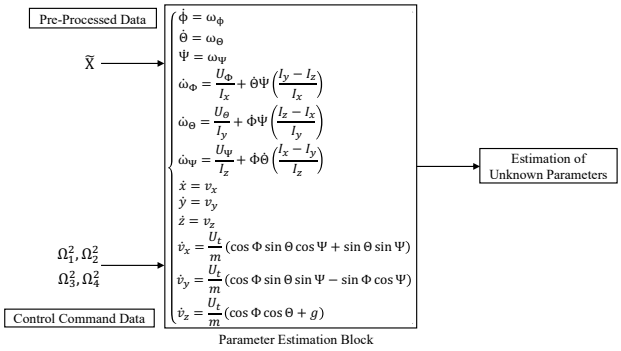


Fig. 5: Parameter estimation block that takes input/output data and an approximate nonlinear model to estimate unknown parameters. In this example, the thrust and drag factors are estimated.

The complexity increases with the number of parameters to estimate and the number of differential equations.

For example, offline learning for our drone was done once the flight controller was modified to capture actuator data (inputs to the system) and sensor data (outputs to the system). We executed several missions with the drone in order to capture a dataset of inputs (control signals to the rotors of the quadcopter) and outputs (sensor values). For instance, in a quadcopter, we collect sensor and control information when the drone is taking-off and reaches a specific height, and then moves forward to a desired location. We run different missions to collect this dataset. With this dataset, we can learn the unknown coefficients from Equation (3) using the online learning mechanism described in Section 3.2. Fig. 5 describes the parameter estimation block for the quadcopter.

In particular, we use the function *nlgreyest* from the System Identification Toolbox of Matlab to find the unknown parameters using the collected data and the nonlinear model. This function can execute the Levenber-Marquardt or interior-reflective Newton methods.

The advantage of this methodology with respect to general machine learning is that we exploit our knowledge about the physical dynamics of the system to create prediction models. For instance, learning a neural network of a drone would not give us a guarantee that the model we have learned is sound (the learned model can add dynamics that do not exist in a real drone), and in addition, neural networks are a black box (they are not a generative model explaining the data like our differential equations). Therefore, an alert will be uninformative and it will be difficult to determine the specific event that caused the alarm. On the other hand, with our approach, we know beforehand that the AV is subject to specific physical laws that are summarized in the differential equations, and then the prediction model is simpler to implement (e.g., by using the Euler integration method, which is not computationally expensive).

3.3 Online Anomaly Detection

In the previous step, we found the parameters of a set of nonlinear equations for our AVs using input/output data. Now, we use this models to generate predictions of the physical process that can be compared with the pre-processed sensor readings in order to identify signal injection attacks.

3.3.1 Predicting AV Behavior with EKF

The Kalman filter is an algorithm that uses noisy sensor measurements to estimate unknown variables of physical processes (e.g., temperature, velocity, pressure) based on prior knowledge of the dynamic equations of the process. It has many applications in robotics, navigation, guidance, and signal processes and econometrics [40].

With *linear* systems, the typical way to predict the next sensor observation is to use the *linear* Kalman filter (which is generally referred simply as the Kalman filter, dropping the "linear" part), but since we are using nonlinear equations, our prediction needs to be done by the Extended-Kalman Filter (EKF). The Extended-Kalman Filter is the more general version of the Kalman filter for systems with more complex dynamic equations (i.e., nonlinear equations). In this case, the transition and observation matrices at each iteration k are defined in terms of the Jacobians (i.e., partial derivatives of a vector-valued function with respect to each of its variables. More details about the EKF can be found in Appendix B.

3.3.2 Anomaly Detection with CUSUM

In order to detect the presence of cyber-attacks, we take the pre-processed sensor readings $\widetilde{Y}(k)$ to generate the prediction $\hat{Y}(k+1)$ using the EKF algorithm described above. Then, in the next iteration we compute the residuals associated to each sensor as follows

$$r_i(k) = \widetilde{Y}_i(k) - \hat{Y}_i(k). \tag{5}$$

If the observations we get from the i^{th} sensor $\widetilde{Y}_i(k)$ are significantly different from the ones we expect (over a period of time) then we generate an alert. The question is how to decide that the deviation is significant, or how long should we observe the anomaly?

There are several detection strategies that take the residuals and compute a detection statistic that quantifies the deviation. For example, Choi et al. [25] used a time-window to keep track of the anomaly and raise an alert if the residuals during the time window exceeded a given value. However, in our previous research [26, 41], we have shown that change detection algorithms such as the CUSUM or the SPRT will outperform other attack-detection algorithms that use time windows.

that strategies that keep track of the historical changes of the residuals without a fixed time window (to prevent the adversary from hiding its attack in between windows of time) have a better performance, especially for persistent threats [26].

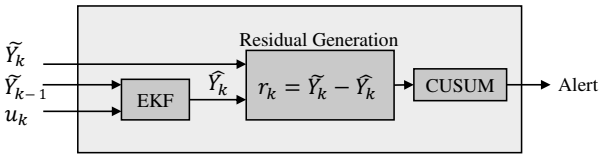


Fig. 6: Anomaly detection: EKF uses our nonlinear model to verify the consistency of our sensors, and the CUSUM algorithm keeps a historical record of the anomalies.

For this reason, instead of using fixed time windows, we use the non-parametric CUSUM statistic, which is described by the following recurrent equation

$$S_i(k+1) = (S_i(k) + |r_i(k)| - b_i)^+ \tag{6}$$

where $S_i(0) = 0$ and $b_i > 0$ is a parameter selected to prevent $S_i(k)$ from increasing when there are no attacks. When $S_i(t_k) > \tau_i$, then an alarm associated to sensor i is triggered. The summary of our detection block is given in Fig. 6.

3.4 Stealthy Attacks

Recall that our attacker can replace a subset of sensor signals Y with a desired Y^a. As a first evaluation of the accuracy of our anomaly detector, we can launch simple attacks, such as bias attacks, where the sensor signal is replaced with the original signal and a fixed bias b: $Y^a = Y + b$. We will use these attacks to evaluate the accuracy of our classifier and other baseline approaches; however, we cannot rely only on this attack, as it may represent an optimistic expectation of what an adversary may do.

A good security principle for evaluating new algorithms is to show that the proposal is resilient against a powerful adversary in order to make sure the new mechanism is secure, even against less sophisticated adversaries. Therefore in this section we present the worst type of attacks that can be launched by a sophisticated attacker, in the hopes of guaranteeing secure operations to less powerful attackers.

In our previous research [26] we argued that the most powerful adversary against PBAD algorithms is one who launches stealthy attacks that maximize the damage to the system without being detected. For this reason, we also evaluate the performance of our defense by analyzing how much deviation of an AV an attack can cause while remaining stealthy.

Let Y^a denote the signal injected by the attacker. We want to maximize the value of that signal, subject to the constraint of not raising any alarms. Most PBAD algorithms have an anomaly score $S(k)$ quantifying the historical deviation of the system with respect to our expectations, and if $S(d) > \tau$ then an alarm is raised. Therefore, the goal of the adversary is to inject a sequence of false sensor readings $Y^a(k)$ to maximize the deviation caused to the system behavior (e.g., deviate the AV from its original position or making it crash), while maintaining $S(k)$ below the alarm threshold:

$$Y^{a*} = \arg\max_{Y^a} Y^a \tag{7}$$

$$\text{Subject to: } S(k) \leq \tau \tag{8}$$

For the CUSUM algorithm introduced above, the optimal attack is given by [26, 42]

$$Y^{a*}(k) = \hat{Y}(k) \pm (\tau + b - S(k)).$$

Notice how the residuals become $r(k) = Y^{a*}(k) - \hat{Y}(k) = \pm(\tau + b - S(k))$ and $S(k+1) = \tau$ for all k.

This stealthy attack allows us to consider the worst case scenario of our PBAD system, where an attacker is not detected while it persistently injects the maximum amount of false information in the system.. If our physical system can survive this type of attack, then we can say that our PBAD is secure. However, if a different PBAD cannot keep our system safe while sustaining this type of attack, then we can say that the second PBAD system is insecure.

As a consequence, if our defense is good enough to limit the impact of this powerful attack, then weaker attacks will be detected or will have less physical damages.

In Section 5 we evaluate our proposed defense and other alternatives proposed in the literature against stealthy attacks.

4 Implementation

We implemented our approach in two different AVs (aerial and ground). Despite both vehicles having different invariants, real-time needs, and specific environments, we show that our methodology can be applicable to AVs in general. Fig. 7 depicts both AVs; one is an Intel Aero Drone, and the other is an autonomous car we built following the BARC project [43].

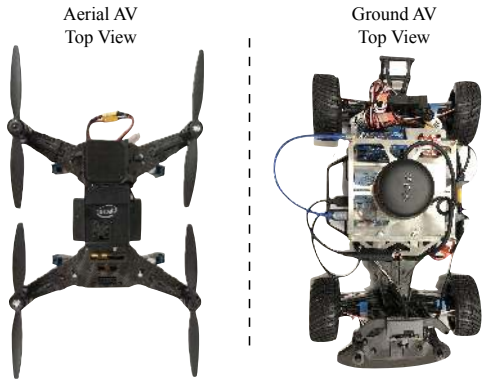


Fig. 7: Intel Aero Ready To Fly Drone and Autonomous Car built on top of a Traxxas Ford Fiesta ST Rally chassis.

4.1 Aerial AV

We implemented our first system in Dronecde's open-source PX4 autopilot controller due to its versatility, highly modulated architecture, and cross-platform hardware support. Implementing our code in PX4 allowed us to test our prototype not only on Intel's Aero Drone, but also on a high-fidelity simulation called jMAVSim.

We modified the autopilot firmware and created a module called `reference_monitor` (written in C++) that can be used in simulation and real hardware.On hardware, we compiled and flashed the firmware into a STM 32-bit ARM Cortex micro-controller clocked at 180MHz with 256+4KB of SRAM inside of the drone. On simulation, we use used PX4 as a flight controller for jMAVSim. We created both implementations from the latest stable source code (version 1.9.2).

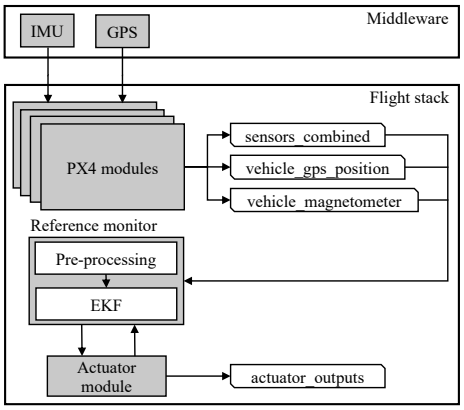


Fig. 8: We implement our anomaly detection tool right before the actuation command is sent to the rotors. In this way we hope our anomaly detection tool will become part of a future planned security reference monitor, deciding when to allow proper access to the rotors.

Fig. 8 depicts the overall architecture of the system with our implementation. Architecturally, the firmware consists of two layers: the flight stack and the middleware. The flight stack provides all the control and estimation modules required for navigating the AV while the middleware provides abstractions that facilitate interaction with hardware components. PX4 executes modules in parallel and it allows inter-process communication following a publish-and-subscribe architecture. We implemented our system in the flight stack layer since it is responsible for navigation.

We subscribed to three topics: `sensor_combined`, `vehicle_magnetometer`, and `vehicle_gps_position` that collectively publish the accelerometer, gyroscope, magnetometer, and GPS raw data. Once new raw data is available, it needs to be processed before it can be used by the estimator. Accelerometer, gyroscope, and magnetometer data is used to calculate the roll, pitch, and yaw angles and angular speed using the pre-processing algorithm described in Appendix A. GPS coordinates of latitude, longitude, and altitude are converted to flat-earth position coordinates with respect to the initial GPS location of the drone [44].

We modified the module in charge of mixing and translating commands such as take-off, land vehicle, and follow route. This module is called `pwm_out_sim` in the simulator and `tap_esc` for our drone. We inserted a function call right before it publishes the computed motor commands for the entire system. This function call then queries our estimator to determine whether an attack is occurring. It is here where discrepancies between control signals and pre-processed sensor information are discovered and the system is alerted. Because we are mostly worried about external attacks (transduction attacks or GPS-spoofing attacks), our adversary cannot bypass our system. If we had to worry about compromised modules (e.g., a malicious `Pwm_out_sim` or `tap_esc`), then our system would need to get the sensor data directly from each sensor, and more importantly, be the only module allowed to send actuation data to the rotors. While the pre-processing section of our implementation runs in parallel with the rest of the system, the function call to the estimator runs sequentially and therefore introduces a small amount of overhead to the entire system.

4.2 Ground AV

Our ground vehicle uses the Robotic Operating System (ROS), specifically, Kinetic Kame. ROS follows a similar architecture to PX4, where modules run in parallel in a publish-and-subscribe architecture. This allowed us to implement our system using the same methodology. Minor changes are related to the specific topics we subscribe to as well as the modules interacting with the reference monitor if an attack has occurred.

Our ROS controller allows for modules to be launched as their own processes while facilitating communication between modules using a centralized master node. ROS allows nodes to be written in different programming languages such as C++ and Python (our choice) to interact with each other via designated APIs. We created a program that executes a lane following algorithm. The vehicle uses the camera to capture an image of the lane, and then it calculates its offset with respect to the lane. After this offset is calculated, the vehicle adjusts the steering angle to maintain the vehicle in the center of the lane.

Our implementation subscribes to three topics: `vel_est`, `line_data`, and `ecu_line_follower/servo`. `vel_est` is used to estimate the velocity of the vehicle while `line_data` and `ecu_line_follower/servo` provide information regarding the position of the line and the servo commands respectively. The pre-processing stage for the ground vehi-

cle is more simple than the aerial vehicle. As with the aerial vehicle, once the values have been pre-processed, they are used in the algorithm to calculate the expected behavior of the system.

5 Evaluation

In this section we evaluate our implementation on PX4 running on jMAVSim and on the Intel Drone, and also our ROS implementation running on the autonomous car. We first show how our algorithm can detect attacks, and then we compare our proposal with other approaches proposed in the literature. In particular, we first compare the classification accuracy of our proposal when compared to others, and then we compare the performance of our proposal under stealthy attacks. We finally measure the overhead of our implementation on the Intel Drone and the BARC autonomous car.

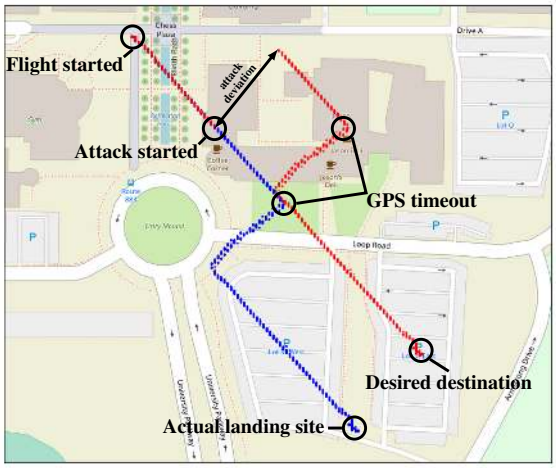


Fig. 9: The actual GPS data gathered from the sensor data (blue) is tampered with by the attacker before being sent to the autopilot. The autopilot then receives a corrupted set of GPS coordinates (red) and makes the "necessary" adjustments in order to return to the established path. The autopilot thinks that the drone reached the desired location, but it has actually deviated.

Our attacks were developed as additional software in each system that hijacked a sensor measurement and spoofed it. This included MAVLink impersonation to jMAVSim and software modules that published false sensor data (in PX4 for the Intel Drone and ROS for the autonomous car). Let us take a look at one example of our attack code. For the car, the line follower algorithm greatly depends on the image published by the camera on the "/cam/raw" topic since it is the main source of data for the decision-making process. Given the fact that there can be multiple nodes publishing the same topic and that there are no sanity checks in place, Fig. 11 shows how a malicious node can publish the same camera topic and

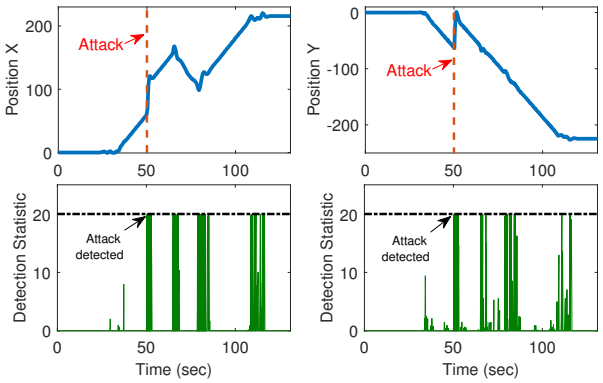


Fig. 10: Detection of the GPS attack in the longitude (X) and latitude (Y) data. The detection statistic of the CUSUM immediately increases and triggers an alarm after 0.2s.

replay a chosen image at a higher rate than that of the camera, overwriting any legitimate image with a malicious one and compromising the data that would be used by the controller in order to make the steering decisions.

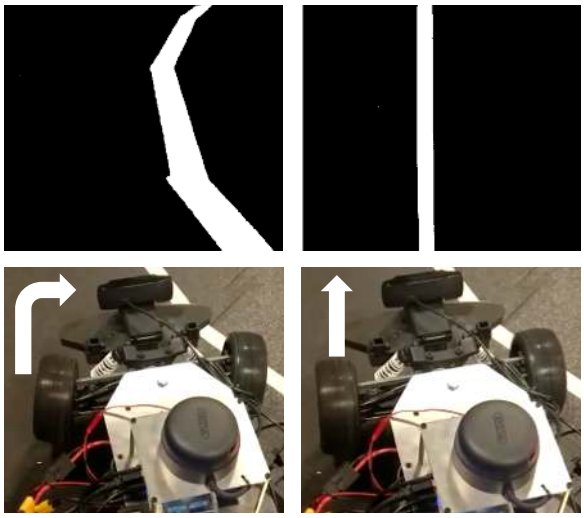
```
def attack():
    rospy.init_node("cam_pub")          # initialize the node
    rate = rospy.rate(30)               # set publishing rate
    ip = ImagePublisher()               # create publisher object
    try:
        while not rospy.is_shutdown():
            ip.get_rgb()                # publish corrupted image
            rate.sleep()                # sleep
    except KeyboardInterrupt:
        print("shutting down ros")
```

Fig. 11: Attack code sample

Fig. 9 shows an example attack on GPS spoofing for a drone, and Fig. 10 shows how our anomaly detection system encounters an inconsistency between desired actuation and direction. Similarly Fig. 12 shows an attack on the camera of our car (the attack resembles recent attacks that added stickers to a lane so an autonomous car would end up driving on the incoming traffic lane [45]). Fig. 13 shows the line deviation and the CUSUM detection metric. Before the attack, the detection metric indicates that the system is behaving correctly. A bias attack of 0.5m is launched after 3.6 s such that the steering angle tries to compensate the sudden change in the line distance, causing the vehicle to drift away from the line. The CUSUM algorithm is able to detect this attack after 0.1 s.

Videos showing our attacks can be found in the following link:

https://www.youtube.com/watch?v=Ljrbtfo0gvM&list=PLmicm3IoL28eLU5v1FH3ZOFSn5NlOuQLG



(a) Actual Camera Feed (b) Malicious Camera Feed

Fig. 12: Visual attack on the car: (a) shows the real image while (b) shows the injected image.

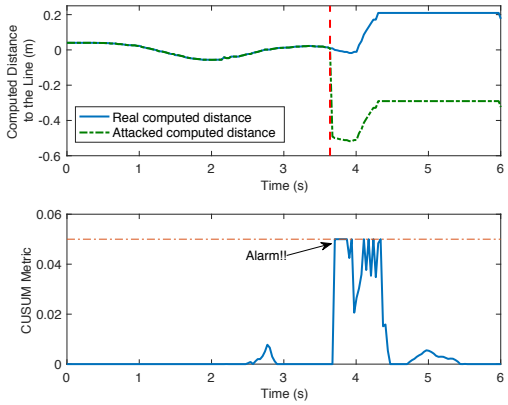


Fig. 13: Line deviation obtained from the video footage and anomaly detection metric. After 3.7 s a bias attack is launched causing the steering control to react leading the vehicle to drift away from the line and saturating the computed distance to the line to its maximum value, i.e., 0.2 m.

5.1 Comparison between NLC and LTW

The previous examples show that our system can detect attacks, but the question is now how do we improve on previous work? Because our proposal uses a Nonlinear Model for predicting the observations, and a CUSUM algorithm for anomaly detection, we will refer to our method as the **NLC algorithm**. To compare our NLC algorithm, we use Choi et al.'s [25] algorithm as a baseline. Because they used a Linear model for predicting observations and a Time-Window algorithm for anomaly detection, we will refer to their method as the **LTW algorithm**. In our experiments we use a time window for LTW of $tw = 3\ s$.

We now perform a series of experiments comparing NLC and LTW. First, we are going to show how the predictions from NLC (which uses EKF) are more accurate than the predictions of LTW (which uses a regular Kalman filter). Then we compare the detection accuracy of both algorithms in terms of the probability of detection, the probability of false alarms, and the time it takes to detect an attack. Finally we compare both NLC and LTW to sophisticated stealthy attackers, and show how NLC can minimize the negative impact to the vehicle caused by these stealthy attacks. .

5.1.1 Linear vs. Nonlinear Predictions

In the first experiment we compare how our nonlinear prediction (with the help of EKF) fares in comparison to previous models that use linear systems, and therefore, linear predictions with the help of the (linear) Kalman filter.

We first have our drone follow trajectory with three different desired positions $(20, 10), (10, -10), (25, -13)$ at a constant altitude 15 m. Using the (linear) Kalman filter and the (nonlinear) EKF, we obtained estimations of the positions x, y and the roll and pitch angles, as depicted in Fig. 14. Notice that both predictions are able to filter sensor noise, but due to the nonlinear dynamics of the quadcopter, the linear Kalman filter has larger prediction errors. On the other hand, the EKF is able to accurately predict the system states even when there are sudden changes in the target position of the drone.

5.1.2 Detection Accuracy

Now we conduct a series of experiments to compare the accuracy of both anomaly detectors, NLC and LTW, in terms of the false positive rates, true positive rates, and the time to detect an attack. The first two metrics are classical metrics in machine learning, but the second one is unique to time-series. In general we can increase the accuracy of the classifier if we keep collecting data to make a decision, but the longer we wait for a decision, the less useful an alert will be; therefore we need to balance all three metrics.

We start by focusing on the time to detect attacks. We select fixed detection thresholds for each detector, and then we launch bias attacks that are injected to the gyroscope of the drone reading of the pitch angle rate (angular velocity over the Y axis) and we measure the time it takes to detect the attacks for different intensities. NLC is able to detect this type of attack faster than LTW as depicted in Fig. 15 (left).

The reason LTW takes longer to detect attacks is two-fold: i) large prediction errors from the linear Kalman filter require large anomaly thresholds to avoid false alarms, and ii) using a time window that resets after a specific number of samples causes weaker attacks to take longer to detect or to not be detected at all. In contrast NLC uses a (nonlinear) EKF with better accuracy, and the CUSUM algorithm does not have time windows, so detecting an attack can be done faster.

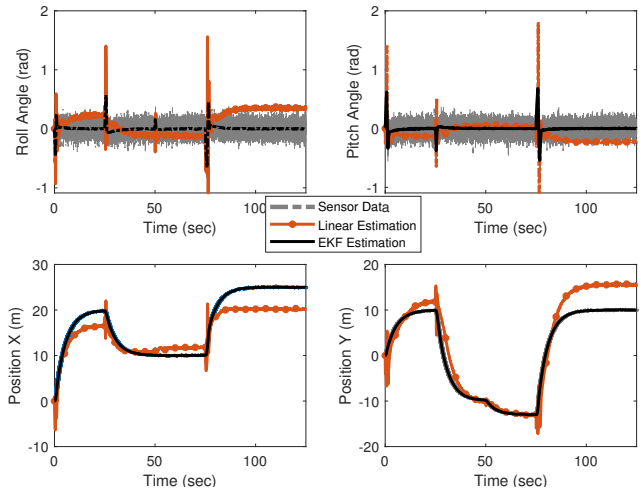


Fig. 14: Comparison between the linear and nonlinear prediction for some of the states of the quadcopter during a mission where both estimators take noisy sensor measurements. Notice that the EKF is able to generate an accurate estimation despite the noise. A linear predictor (similar to the one in [25]) has larger estimation errors.

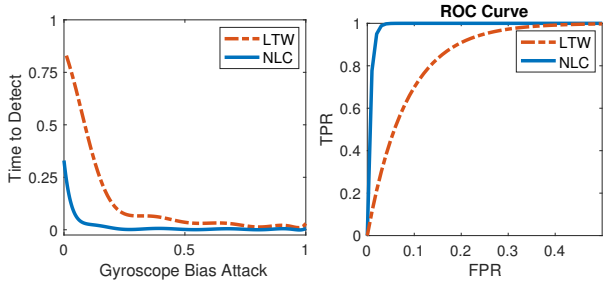


Fig. 15: Left: Time to detect an attack for different gyroscope bias attacks to the drone. Right: ROC curve comparison for NLC and LTW applied to the drone.

We now compute the ROC curve for NLC and LTW. Fig. 15 (right) illustrates the ROC curve for both anomaly detection strategies. Clearly, the NLC has a better ROC curve than LTW. In particular, the NLC is able to detect the attack with a probability close to 1 while having a false alarm rate (below 2%); on the other hand, when LTC detects almost all the attacks, the false alarm rate is around 40%.

When we turn our attention to the ground vehicle, we get similar ROC results, as illustrated in Fig. 16, showing again that NLC outperforms LTW in a variety of AVs.

5.1.3 Resiliency of NLC and LTW to Stealthy Attacks

We now describe how to launch stealthy attacks in gyroscopes and in GPS for LTW and NLCs.

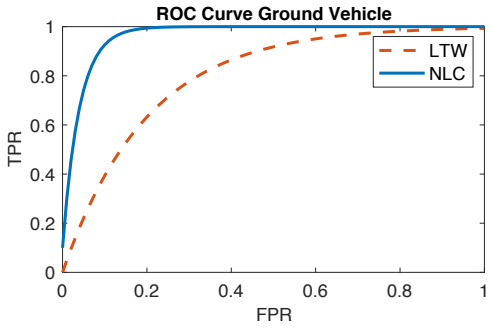


Fig. 16: ROC curve for both anomaly detection strategies implemented in the ground vehicle.

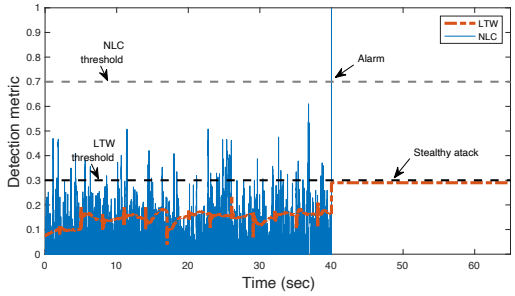


Fig. 17: A stealthy gyroscope attack is launched after 40 seconds. LTW does not detect the attack, but NLC detects the attack in less than 0.1 second.

Stealthy attack for LTW. The detection strategy introduced in [25] consists of accumulating the quadratic error $s\_err_i(k) = |Y_i(k) - \hat{Y}_i(k)|^2$, in an anomaly statistic $error\_sum_i(k+1) = error\_sum_i(k) + s\_err_i(k)$. Therefore, the detection statistic is given by $error_i(k) = err\_sum_i(k)/tw$, where $window > 0$ is the time window and tw is the time window count that increases at each iteration. When $tw > window$, then the detector is reset (i.e., $tw = 0$ and $err\_sum_i = 0$). The stealthy attack is then given by

$$Y_i^a(k) = \hat{Y}_i(k) + \sqrt{-err\_sum_i(k) + \tau_i tw}. \quad (9)$$

Replacing the attack in equation (9), we have that $s\_err_i(k) = -err\_sum_i(k) + \tau_i tw$ and $error_i(k) = \tau_i$, therefore the attack is never detected.

Stealthy Attack for NLC. Similar to the attack for LTW, we have that the stealthy attack for NLC is given by

$$Y_i^a(k) = \hat{Y}_i(k) - S_i(k) + b_i \pm \tau_i. \quad (10)$$

Replacing this attack in equations (5) and (6) shows that $S_i(k) = \tau_i$, and the alarm is never triggered.

Fig. 17 illustrates a stealthy attack for the LTW in the gyroscope after 40 s with $\tau_i = 0.3$. Note that the attack is designed such that the anomaly score (detection metric) never

reaches the threshold and no alarms are triggered. In contrast, this attack is quickly detected by NLC.

In our next set of experiments, we launch stealthy attacks for the angular speed associated to the roll angle and for the GPS reading associated to the X position. The target position of the drone is $(10, 10)$ at a constant altitude of 15 m. The drone reaches its desired position and after 25 seconds, the stealthy attack starts causing deviations in the X axis because the controller is trying to compensate for the false information. Fig. 18 depicts the sensor attack (top) and the real position (bottom) for both attacks and for both detectors. The solid circles indicate the final position of the drone at 50 s. Notice that the deviation from the desired position is larger with the LTW than with the NLC making our proposed NLC significantly more secure than LTW because it manages to keep the system closer to its desired trajectory under stealthy attacks.

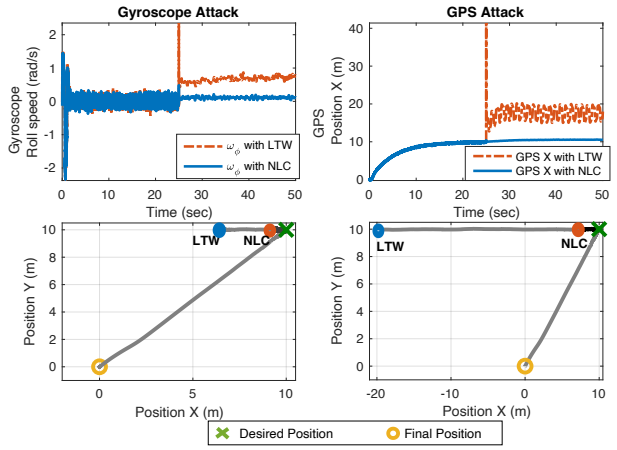


Fig. 18: Stealthy gyroscope and GPS attacks for both detectors, LTW and NLC. The bottom plots illustrate the target position of the drone and the final position (solid circle) due to the attack. Clearly, our NLC limits the impact of stealthy attack.

Now, we study the impact of a stealthy attack in the altitude reading. In this case, the duration of the attack is 20 s. Notice in Fig. 19 that the stealthy attack with LTW causes the drone to crash, damaging the drone and possibly injuring people. Therefore we can argue that previous LTW work is not secure against stealthy attacks because the attacker can catastrophically damage the system without detection. On the other hand, with the NLC the deviation caused by the adversary is small and the drone is able to recover and return to the desired altitude when the attack finishes. This shows the importance of considering stealthy attackers in future work on physical invariants for the cyber-security of drones and other autonomous vehicles.

Finally, we would like to use a systematic metric like the ROC curve to compare both NLC and LTW; however, ROC

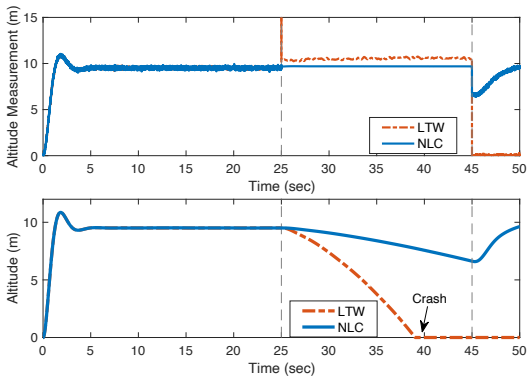


Fig. 19: Stealthy attack with a duration of 20 s in the altitude signal. With LTW the attack causes the drone to crash; however, with our NLC, the drone altitude is slowly affected and when the attack finishes the drone returns to its desired position.

curves assume a true positive rate, and stealthy attacks are by definition undetected, so we cannot use ROC curves to measure the performance of PBAD algorithms to stealthy attacks. To solve this problem we look at the new performance metric we previously introduced [26] to compare anomaly detection strategies against stealthy attacks. The Y axis of this new metric quantifies the maximum deviation caused by the stealthy attack during 35 s and the X axis corresponds to the expected time for false alarms (an adaptation of the true positive rate that includes the time component, which is important for classification of time series).

Fig. 20 shows the comparison of NLC and LTW. Clearly, due to the improved nonlinear model and better detection strategy, our proposed NLC forces an attacker who wants to remain stealthy, to launch very small attacks.

5.2 Effects of External Disturbances

Sudden disturbances like wind gusts have an undesired effect in the anomaly detection strategy that not only can affect the trajectory of the drone but can also raise false alarms. Significant wind forces impact air vehicles in two different ways: i) the drone is pushed from the desired position (translation), and ii) the drone rotates on any of its axis.

The PID controller on a drone is typically able to compensate for the effects of the wind when the wind velocity is around less than 5 m/s. Recall that the EKF in our detection module receives the pre-processed sensor signals and the control inputs sent to the propellers. Since the controller tries to compensate for the wind gust, but our model does not take into account the presence of non-zero disturbances, the estimation generated by the EKF will not be accurate and our detection algorithm will raise a false alarm.

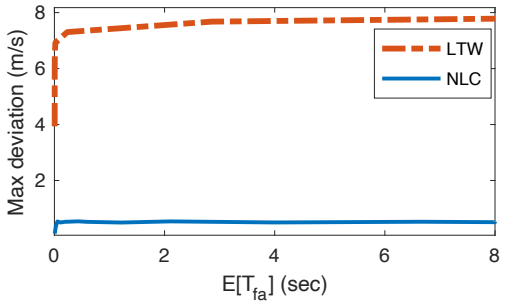


Fig. 20: Comparison between the NLC and LTW based on the performance metric proposed in [26] for the pitch gyroscope sensor. The maximum deviation consists on the maximum XYZ deviation after 35 s of the attack. Clearly, due to the improved nonlinear model and better detection strategy, our proposed NLC is able to limit the impact of stealthy attacks when compared to LTW.

During our experiments with the real drone there was not significant wind, so we could not check the results on real hardware, instead we look at high-fidelity simulations. In order to test and compare our anomaly detection in the presence of wind disturbances, we use the Dryden model, which is a mathematical model of continuous gusts accepted for use by the United States Department of Defense in certain aircraft design and simulation applications [46]. The Dryden model is characterized by power spectral densities of the gust's three linear velocity components described by (11). The parameters $\sigma_u, \sigma_v, \sigma_w$ are the turbulence intensities and L_u, L_v, L_w are the scale lengths. Particularly, this model can be considered as a linear filter that converts white noise into colored noise.

$$\Phi_{u_g}(\Omega) = \sigma_u^2 \frac{2L_u}{\pi} \frac{1}{1+(L_u\Omega)^2}$$
$$\Phi_{v_g}(\Omega) = \sigma_v^2 \frac{2L_v}{\pi} \frac{1+12(L_v\Omega)^2}{(1+4(L_v\Omega)^2)^2}$$
$$\Phi_{w_g}(\Omega) = \sigma_w^2 \frac{2L_w}{\pi} \frac{1+12(L_w\Omega)^2}{(1+4(L_w\Omega)^2)^2}, \quad (11)$$

Fig. 21 (left) illustrates the effect of a sudden increase in the wind speed that runs North-East between 50 s and 110 s. The wind change causes oscillations that cause the CUSUM detection metric associated to the roll angular velocity ω_ϕ to raise false alarms.

We can solve this problem by relying on wind sensors also known as anemometers (e.g., Ultrasonic Wind Sensors) that provide accurate measures of the wind speed and its direction. There is a wide variety of wind sensors that are suitable for UAVs, such as the FT205 from FT technologies or the TriSonica-mini from Anemoment. We can use these

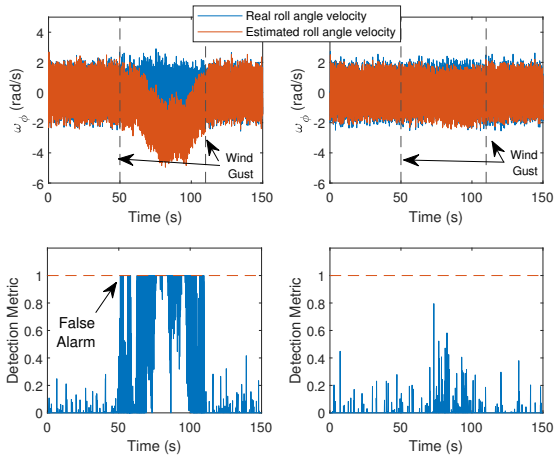


Fig. 21: Effects of wind in the NLC. Sudden changes in the weather conditions can cause false alarms and undesired oscillations of the drone. However by adding wind sensors (e.g., ultrasonic wind sensors), it is possible to improve the performance of the controller and avoid false alarms.

measurements to quantify the effects of wind and obtain better estimations that can decrease the false alarms.

To this end, we need to define a model of the drone dynamics with wind disturbance and modifying the dynamics in equation (2) used by the EKF, to include the disturbance elements. According to [47], the angular and linear velocities can be described as:

$$\dot{v}_x^W = \dot{v}_x + \frac{1}{m} d_x,$$
$$\dot{v}_y^W = \dot{v}_y + \frac{1}{m} d_y,$$
$$\dot{v}_z^W = \dot{v}_z + \frac{1}{m} d_z,$$

where d_x, d_y, d_z the wind disturbances that affect the drone position (translation). Fig. 21 (right) depicts how adding wind sensors may help to mitigate the effects of the wind in the drone and avoid false alarms.

5.3 Performance Overhead

In our last evaluation study we look at the performance overhead introduced by our reference monitor on both aerial and ground vehicles. Our results show that this increase does not impose adverse computational overhead to affect the real-time constrains of each AV.

5.3.1 Aerial Vehicle

The latest stable version of PX4, v1.9.2, compiled for the Intel Aero drone contains a total of 50 modules, drivers, and system commands. In terms of size, our additions consist of 6 files with a total of 920 LOC. The unmodified firmware has a size of 862.3KB, while the modified version with our additions is

874.7KB. This represents a 1.43% increase in the size of the binary firmware.

To measure execution performance, we first must define what overhead means when running multiple independent modules in a real-time OS. We cannot measure overhead from within a module since only the OS itself has a concept of system load. Also, the calculations done by these modules are continuous as they are constantly processing new data and do not have a point at which we can measure how long they took to finish a task. Instead, we can measure overhead by calculating the CPU utilization for all modules within slices of time. Fortunately, the scheduler for PX4 maintains a *system_load_s* structure containing data about all tasks. It uses the *hrt_absolute_time()* function to obtain an unsigned 64-bit integer containing the number of microseconds(us) since an arbitrarily selected epoch at boot. This gives over 500,000 years before the integer would overflow, allowing for a reliable measure of system time.

The system scheduler measures the time between when a task is resumed and suspended and adds this time to the task's *total_runtime*. Whenever the scheduler does not have a task to run, this time goes to the *idle* task. We cannot obtain overhead directly from this value, however, because this is a measure of how much CPU time each task has had over its entire lifetime. Longer-running tasks will naturally accumulate more CPU time. Therefore, we instead view the system at periodic snapshots, saving *total_runtime* for each task at each snapshot. Between two snapshots, we can compare the increase in *total_runtime* for each task which provides an accurate measure for how long each task ran in between those two snapshots. We then can use this to calculate the percentage of CPU time that each task used for that time slice. By collecting data from multiple time slices and averaging the results, we can get the average overhead for all tasks in the system.

Table 1 sorts the top 13 processes running on hardware by CPU utilization. These processes amount to about 95.22% of the CPU resources available. Looking at the top 13 processes we can observe that some modules perform system activities like the *idle* module which is designed to run when the system does not have a process to execute and the *hpwork* module which executes several high priority threads that do not own a stack frame. Other modules handle communication like the *mavlink_if0* and *mavlink_if1* modules that allow communication between the firmware and the ground station via the MAVLink protocol and the *logger* module that provides system and topic logging. Logic modules include: the *EKF2* module that implements the vehicles' own Extended-Kalman Filter for attitude and position calculations; the *mc_att_control* and *mc_pos_control* modules that provides attitude and rate control, as well as position and velocity error; and the *commander* module that manages internal states. Finally, driver modules that interact with physical devices include the *sensors* module that gathers gyroscope, accelerometer and magnetometer data, the *gps* module that handles the GPS signal, and the *tap_esc* module that mixes the actuator commands into PWM signals for the motors.

Module	Armed	Hovering	RC
Idle	30.1444%	29.4379%	30.6056%
mavlink_if1	16.0183%	15.6195%	15.8956%
EKF2	14.3242%	14.3779%	14.3006%
logger	6.8647%	7.1288%	6.8752%
mc_att_control	5.4349%	5.4007%	5.3425%
reference _monitor	**5.3572%**	**5.4332%**	**5.5093%**
tap_esc	4.4742%	4.4357%	4.4285%
sensors	4.2744%	4.4792%	4.5200%
hpwork	2.5077%	2.4462%	2.4750%
mavlink_if0	2.3323%	2.1384%	2.2667%
mc_pos_control	1.4911%	2.4727%	1.4693%
commander	1.4824%	1.4478%	1.4448%
gps	0.3662%	0.3323%	0.3077%

Table 1: CPU utilization of top 13 modules inside of IntelAero. The drone is tested under three different scenarios: armed, hovering, and Radio Controlled (RC).

On average, our *reference_monitor* module in hardware consumes 5.4332% of the CPU time available. Also, during our tests with the actual physical device, we did not observe any input delay or behavioral differences after installing the modified firmware.

5.3.2 Ground Vehicle

For our implementation of the reference monitor on the ground vehicle, we added a total of 231 LOC across three different files. This brings our implementation to 37.3KB of storage space. Since, our implementation is done in Python and no binary executable containing the controller is compiled, we did not calculate the percent increase with respect to the size of the controller. We added our module to the Robotic Operating System (ROS) controller running in the vehicle. ROS, like PX4, allows for modules to run parallel to each other. Therefore, our reference monitor also runs in parallel with the rest of the system. We measured the execution time of our module while the vehicle was executing its "line following" algorithm. We collected performance data with respect to the entire system while the vehicle was following a line utilizing a similar approach as the measurements done for the aerial vehicle. We collected CPU utilization for each module, including threads, and average it our with respect to the rest of the system. Our results indicate that on average, our reference monitor utilizes 2.2501% CPU resources on the overall system.

Table 2 states the top 13 processes in the ground vehicle related to the execution of the line following algorithm. System nodes include *rosout*, *rosmaster*, and *roslaunch* that handle logging information, initial set up communication between

Module	Line Following	CA
lidar_collision_avoidance	12.6886%	13.0694%
elp_cam_bridge	11.0179%	15.6009%
process_line	10.3861%	11.7353%
image_processing	6.0726%	7.8523%
reference_monitor	**2.5192%**	**1.9809%**
arduino_node	2.4150%	2.5133%
line_follower	1.0097%	1.0488%
low_level_controller	0.7948%	0.4503%
perot_demo	0.6990%	0.6589%
roslaunch	0.4541%	0.2678%
rplidarNode	0.3074%	0.3020%
rosmaster	0.2973%	0.1569%
rosout	0.0658%	0.0250%

Table 2: CPU utilization of top 13 modules inside of the ground vehicle. The vehicle is tested under 2 different scenarios: line following and Collision Avoidance (CA).

nodes, and the launching of several nodes simultaneously. The first node to interact with our system is the *elp_cam_bridge node* which receives raw camera information and makes it available to the system in pixels. The *image_processing* node receives this camera information, processes it, and publishes the image in terms of bytes. The *process_line* node takes this information and produces the position with respect to x and y as well as the angle of the current line. This information is fed to the *line_follower* node that produces the appropriate servo command. The node *perot_demo* then takes this information and outputs ECU commands. Finally, the *low_level_controller* publishes the corresponding PWM signal to the actuators. Our *reference_monitor* node runs in parallel with the rest of the system and publishes an attack flag when an anomaly has been detected. This attack flag alerts the system that an attack has been detected.

6 Conclusion and Future Work

In this paper we have presented SAVIOR, a general framework for protecting autonomous vehicles from signal injection attacks. The key elements of our proposal are the following: (1) use of well-known physical invariants, (2) the use of offline system identification, (3) the use of CUSUM algorithms, and (4) evaluating the effectiveness of the anomaly detection tool with stealthy attacks that attempt to maximize the damage to the system.

The main point of (1) is that if the physical models of the system under control are known, there is no need to use suboptimal generic linear models or to use neural networks or other black-box machine learning tools that do not explain the physics of the system. The main point of (2) is that we do not need to develop the nonlinear equations of the system from first principles, the parameters of these equations can be learned via system identification. The main point of (3) is that we have seen systematically how change detection algorithms such as CUSUM or the SPRT perform better than other ways to keep track of a historical anomaly [26, 41]. Finally, the main point of (4) is that we can always detect attacks that are random enough, but if an attacker attempts to bypass our system, then by looking at the worst case stealthy attacks, we can identify the lower bound of the performance of our system (i.e., identify how the physical system would behave if the attacker bypasses anomaly detection and injects false data).

In future work we plan to develop SAVIOR into a reference monitor that not only detects attacks, but can take action once an attack is detected, in order to protect the safety of the AV and the people around it.

Acknowledgements

We thank the anonymous reviewers for their insightful comments. This work was partially supported by National Science Foundation (NSF) Awards 1834215, 1834216, 1929410, 1931573 and the Air Force Office of Scientific Research under award number FA9550-17-1-0135.

References

[1] G. Seetharaman, A. Lakhotia, and E. P. Blasch, "Unmanned vehicles come of age: The darpa grand challenge," *Computer*, vol. 39, no. 12, pp. 26–29, 2006.

[2] F. Schroth, "Gartner predicts ~3 million drones to be shipped in 2017," https://dronelife.com/2017/02/10/gartner-predicts-3-million-drones-shipped-2017/, 2017.

[3] D. Jenkins and B. Vasigh, *The economic impact of unmanned aircraft systems integration in the United States*. Association for Unmanned Vehicle Systems International (AUVSI), 2013.

[4] E. Weise, "Mysterious gps glitch telling ships they're parked at airport may be anti-drone measure," https://www.usatoday.com/story/tech/news/2017/09/26/gps-spoofing-makes-ships-russian-waters-think-theyre-land/703476001/, 2017.

[5] A. Rawnsley, "Iran's alleged drone hack: Tough, but possible," https://www.wired.com/2011/12/iran-drone-hack-gps/, 2011.

[6] A. J. Kerns, D. P. Shepard, J. A. Bhatti, and T. E. Humphreys, "Unmanned aircraft capture and control via gps spoofing," *Journal of Field Robotics*, vol. 31, no. 4, pp. 617–636, 2014.

[7] T. E. Humphreys, B. M. Ledvina, M. L. Psiaki, B. W. O'Hanlon, and P. M. Kintner, "Assessing the spoofing threat: Development of a portable gps civilian spoofer," in *Radionavigation Laboratory Conference Proceedings*, 2008.

[8] N. O. Tippenhauer, C. Pöpper, K. B. Rasmussen, and S. Capkun, "On the requirements for successful gps spoofing attacks," in *Conference on Computer and Communications Security (CCS)*. ACM, 2011, pp. 75–86.

[9] J. Noh, Y. Kwon, Y. Son, H. Shin, D. Kim, J. Choi, and Y. Kim, "Tractor beam: Safe-hijacking of consumer drones with adaptive gps spoofing," *ACM Transactions on Privacy and Security (TOPS)*, vol. 22, no. 2, p. 12, 2019.

[10] K. Fu and W. Xu, "Risks of trusting the physics of sensors," *Communications of the ACM*, vol. 61, no. 2, pp. 20–23, 2018.

[11] I. Giechaskiel and K. B. Rasmussen, "Sok: Taxonomy and challenges of out-of-band signal injection attacks and defenses," *arXiv preprint arXiv:1901.06935*, 2019.

[12] T. Trippel, O. Weisse, W. Xu, P. Honeyman, and K. Fu, "Walnut: Waging doubt on the integrity of mems accelerometers with acoustic injection attacks," in *European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2017, pp. 3–18.

[13] D. F. Kune, J. Backes, S. S. Clark, D. Kramer, M. Reynolds, K. Fu, Y. Kim, and W. Xu, "Ghost talk: Mitigating emi signal injection attacks against analog sensors," in *Symposium on Security and Privacy (S&P)*. IEEE, 2013, pp. 145–159.

[14] Y. M. Son, H. C. Shin, D. K. Kim, Y. S. Park, J. H. Noh, K. B. Choi, J. W. Choi, and Y. D. Kim, "Rocking drones with intentional sound noise on gyroscopic sensors," in *USENIX Security Symposium (USENIX Security)*. USENIX Association, 2015.

[15] D. Davidson, H. Wu, R. Jellinek, T. Ristenpart, and V. Singh, "Controlling UAVs with sensor input spoofing attacks," in *Workshop on Offensive Technologies (WOOT)*. USENIX Association, 2016, pp. 221–231.

[16] Y. Cao, C. Xiao, B. Cyr, Y. Zhou, W. Park, S. Rampazzi, Q. A. Chen, K. Fu, and Z. M. Mao, "Adversarial Sensor Attack on LiDAR-based Perception in Autonomous Driving," in *Conference on Computer and Communications Security (CCS)*, 2019.

[17] J. Selvaraj, G. Y. Dayanıklı, N. P. Gaunkar, D. Ware, R. M. Gerdes, M. Mina *et al.*, "Electromagnetic induction attacks against embedded systems," in *Asia Conference on Computer and Communications Security (AsiaCCS)*. ACM, 2018, pp. 499–510.

[18] J. Giraldo, D. Urbina, A. Cardenas, J. Valente, M. Faisal, J. Ruths, N. O. Tippenhauer, H. Sandberg, and R. Candell, "A survey of physics-based attack detection in cyber-physical systems," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, p. 76, 2018.

[19] D. Hadžiosmanović, R. Sommer, E. Zambon, and P. H. Hartel, "Through the eye of the PLC: semantic security monitoring for industrial processes," in *Annual Computer Security Applications Conference (ACSAC)*. ACM, 2014, pp. 126–135.

[20] C. M. Ahmed, C. Murguia, and J. Ruths, "Model-based attack detection scheme for smart water distribution networks," in *Asia Conference on Computer and Communications Security (AsiaCCS)*. ACM, 2017, pp. 101–113.

[21] Y. Liu, P. Ning, and M. K. Reiter, "False data injection attacks against state estimation in electric power grids," in *Conference on Computer and Communications Security (CCS)*. ACM, 2009, pp. 21–32.

[22] S. Etigowni, D. J. Tian, G. Hernandez, S. Zonouz, and K. Butler, "Cpac: securing critical infrastructure with cyber-physical access control," in *Annual Computer Security Applications Conference (ACSAC)*. ACM, 2016, pp. 139–152.

[23] W. Aoudi, M. Iturbe, and M. Almgren, "Truth will out: Departure-based process-level detection of stealthy attacks on control systems," in *Conference on Computer and Communications Security (CCS)*. ACM, 2018, pp. 817–831.

[24] A. A. Cardenas, S. Amin, Z.-S. Lin, Y.-L. Huang, C.-Y. Huang, and S. Sastry, "Attacks against process control systems: risk assessment, detection, and response," in *Asia Conference on Computer and Communications Security (AsiaCCS)*, 2011, pp. 355–366.

[25] H. Choi, W.-C. Lee, Y. Aafer, F. Fei, Z. Tu, X. Zhang, D. Xu, and X. Xinyan, "Detecting attacks against robotic vehicles: A control invariant approach," in *Conference on Computer and Communications Security (CCS)*. ACM, 2018, pp. 801–816.

[26] D. I. Urbina, J. A. Giraldo, A. A. Cardenas, N. O. Tippenhauer, J. Valente, M. Faisal, J. Ruths, R. Candell, and H. Sandberg, "Limiting the impact of stealthy attacks on industrial control systems," in *Conference on Computer and Communications Security (CCS)*. ACM, 2016, pp. 1092–1105.

[27] Y. Tu, Z. Lin, I. Lee, and X. Hei, "Injected and delivered: fabricating implicit control over actuation systems by spoofing inertial sensors," in *USENIX Security Symposium (USENIX Security)*. USENIX Association, 2018, pp. 1545–1562.

[28] C. Yan, W. Xu, and J. Liu, "Can you trust autonomous vehicles: Contactless attacks against sensors of self-driving vehicle," *DEF CON*, vol. 24, 2016.

[29] J. Petit, B. Stottelaar, M. Feiri, and F. Kargl, "Remote attacks on automated vehicles sensors: Experiments on camera and lidar," *Black Hat Europe 11*, 2015.

[30] H. Shin, D. Kim, Y. Kwon, and Y. Kim, "Illusion and dazzle: Adversarial optical channel exploits against lidars for automotive applications," in *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, 2017, pp. 445–467.

[31] K. C. Zeng, S. Liu, Y. Shu, D. Wang, H. Li, Y. Dou, G. Wang, and Y. Yang, "All your GPS are belong to us: Towards stealthy manipulation of road navigation systems," in *USENIX Security Symposium (USENIX Security)*. USENIX Association, 2018, pp. 1527–1544.

[32] J. Giraldo, D. Urbina, A. A. Cardenas, and N. O. Tippenhauer, "Hide and seek: An architecture for improving attack-visibility in industrial control systems," in *International Conference on Applied Cryptography and Network Security (ACNS)*. Springer, 2019, pp. 175–195.

[33] T. D. Gillespie, *Fundamentals of Vehicle Dynamics*. Society of Automotive Engineers, Inc., 1997.

[34] A. Chovancová, T. Fico, L. Chovanec, and P. Hubinsk, "Mathematical modelling and parameter identification of quadrotor (a survey)," *Procedia Engineering*, vol. 96, pp. 172–181, 2014.

[35] T. Luukkonen, "Modelling and control of quadcopter," *Independent research project in applied mathematics, Espoo*, vol. 22, 2011.

[36] J. Kong, M. Pfeiffer, G. Schildbach, and F. Borrelli, "Kinematic and dynamic vehicle models for autonomous driving control design," in *Intelligent Vehicles Symposium (IV)*. IEEE, 2015, pp. 1094–1099.

[37] I. Griva, S. G. Nash, and A. Sofer, *Linear and nonlinear optimization*. Siam, 2009, vol. 108.

[38] J. J. Moré, "The levenberg-marquardt algorithm: implementation and theory," in *Numerical analysis*. Springer, 1978, pp. 105–116.

[39] T. F. Coleman and Y. Li, "An interior trust region approach for nonlinear minimization subject to bounds," *SIAM Journal on optimization*, vol. 6, no. 2, pp. 418–445, 1996.

[40] L. Xie, D. Popa, and F. L. Lewis, *Optimal and robust estimation: with an introduction to stochastic control theory*. CRC press, 2007.

[41] A. A. Cárdenas, S. Radosavac, and J. S. Baras, "Evaluation of detection algorithms for mac layer misbehavior: Theory and experiments," *IEEE/ACM Transactions on Networking (ToN)*, vol. 17, no. 2, pp. 605–617, 2009.

[42] C. Murguia and J. Ruths, "Cusum and chi-squared attack detection of compromised sensors," in *Conference on Control Applications (CCA)*. IEEE, 2016, pp. 474–480.

[43] J. Gonzales, F. Zhang, K. Li, and F. Borrelli, "Autonomous drifting with onboard sensors," in *International Symposium on Advanced Vehicle Control (AVEC)*, 2016, p. 133.

[44] B. L. Stevens, F. L. Lewis, and E. N. Johnson, *Aircraft control and simulation: dynamics, controls design, and autonomous systems*. John Wiley & Sons, 2015.

[45] Tencent Keen Security Lab, "Experimental security research of tesla autopilot," White Paper, 2019.

[46] "Flying qualities in piloted aircraft," *Department of Defense Handbook. MIL-HDBK-1797B*, 2012.

[47] C. Wang, B. Song, P. Huang, and C. Tang, "Trajectory tracking control for quadrotor robot subject to payload variation and wind gust disturbance," *Journal of Intelligent & Robotic Systems*, vol. 83, no. 2, pp. 315–333, 2016.

[48] S. Romaniuk and Z. Gosiewski, "Kalman filter realization for orientation and position estimation on dedicated processor," *acta mechanica et automatica*, vol. 8, no. 2, pp. 88–94, 2014.

[49] M. J. Caruso, "Applications of magnetic sensors for low cost compass systems," in *Position Location and Navigation Symposium (PLANS)*. IEEE, 2000, pp. 177–184.

Appendices

A Sensor Pre-Processing

IMUs used in vehicles are composed of a 3-axis accelerometer, 3-axis gyroscope, and 3-axis magnetometer that can be combined to calculate the vehicle attitude (roll ϕ, pitch θ, yaw ψ angles) and attitude rates $(\dot{\phi},\dot{\theta},\dot{\psi})$. Also, most AVs have GPS receivers to collect information about the spacial position of the drone (x,y,z). Before using this data, there are several challenges that arise when using IMU information: i) the IMU does not provide direct information about the attitude of the drone, ii) the accelerometer is very noisy, iii) gyroscopes have an intrinsic bias that causes a drift in the angles calculation, and iv) the GPS captures latitude, longitude, and altitude information, but we need to compute the x, y, z position in meters with respect to an initial location. In order to overcome these issues, it is necessary to design a *pre-processing stage* that takes all sensor readings and returns new and usable readings of the system states.

First, we can define a_x,a_y,a_z as the 3-axis accelerometer measurements; $\omega_x,\omega_y,\omega_z$ as the angular velocity measured by the 3-axis gyroscope; m_x,m_y,m_z as the magnetometer readings; and G_{lat},G_{lon},G_{alt} as the GPS position. All of them form the vector of raw sensor readings $Y=[a_X,a_y,a_z,\omega_x,\omega_y,\omega_z,m_x,m_y,m_z,G_{lat},G_{lon},G_{alt}]^\top$. With Y, the pre-processing stage computes a new output $\widetilde{Y}=[\phi,\theta,\psi,\dot{\phi},\dot{\theta},\dot{\psi},x,y,z]$ with the information necessary to generate predictions of the system states. The pre-processing stage uses a data-fusion algorithm that combines the gyroscope readings of angular velocities with the accelerometer or magnetometer measurements to calculate the intrinsic bias of the gyroscope and then generate accurate roll, pitch, and yaw angle readings. The algorithm is based on a simple linear Kalman filter that exploits some geometric properties of the accelerometer and magnetometer.

On the other hand, the pre-processing takes the GPS readings that correspond to the geodetic latitude, longitude, and altitude and converts them to flat Earth position (x,y,z) that can be used to determine the position of the drone in meters with respect to its initial location. We choose a simple approach that is precise for changes up to hundreds of meters, which considers the ellipsoid planet model known as WGS84. Details about the conversion algorithm can be found in [44].

Bias Correction

In order to correct the bias of the gyroscope, we use a data-fusion procedure that combines the accelerometer/magnetometer with the gyroscope readings to obtain accurate angle measurements [48]. This methodology exploits the fact that the accelerometer and magnetometer are affected by the gravitational field of the Earth such that any inclination of the accelerometer (pitch or roll) will be reflected on each of its measurements. Similarly, the magnetometer acts as a compass and is affected by the direction and inclination of the drone. We then perform two main steps: 1) compute a noisy angle approximation using the accelerometer (or magnetometer for the yaw angle), and 2) using the angular velocity measured by the gyroscope and the angle approximation obtained in step 1, estimate the gyroscope bias and correct the gyroscope measurement in order to compute an accurate angle.

The first step uses geometrical properties of the accelerometer as follows:

$$\phi_{a,t}=-\tan^{-1}\left(\frac{a_{y,t}}{\sqrt{a_{x,t}^2+a_{z,t}^2}}\right),\ \theta_{a,t}=\tan^{-1}\left(\frac{a_{x,t}}{\sqrt{a_{y,t}^2+a_{z,t}^2}}\right) \tag{12}$$

where $\theta_{a,t},\phi_{a,t}$ are roll and pitch computed from the accelerometer readings.

For the second step, we will describe the procedure introduced in [48] to obtain only the roll angle ϕ, but the same steps can be applied for θ and ψ. We need to describe the dynamic equation that describes the evolution over time of ϕ with respect to the angular velocity measured by the gyroscope ω_x as follows:

$$\begin{aligned}\phi_{t+1}&=\phi_t+d_t(\omega_{x,t}-\omega_{x,t}^b)\\ \omega_{x,t}^b&=\omega_{x,t-1}^b,\end{aligned} \tag{13}$$

where $\omega_{x,t}^b$ is the gyroscope bias and d_t is the sampling period. With the dynamic representation in (13), we can use a Kalman Filter to estimate both unknown variables (i.e., unknown because they are not directly measured), the angle ϕ_t, and the bias $\omega_{x,t}$. Kalman filter is a mean squared error minimizer that is used to estimate unknown variables from available sensor readings. Its form is as follows:

$$\mathbf{x}_{t+1}=\mathrm{A}\mathbf{x}_t+\mathrm{B}\mathrm{u}_t+\mathrm{K}_t(\mathbf{z}_t-\mathrm{H}\mathbf{x}_t)$$

where $H=[1\ 0]$, $\mathbf{z}_t=\phi_{a,t}$, $\mathbf{x}_t=[\phi_t,\omega_{x,t}^b]^\top$, $u_t=\omega_{x,t}$,

$$A=\begin{bmatrix}1 & -d_t\\ 0 & 1\end{bmatrix},B=\begin{bmatrix}d_t\\ 0\end{bmatrix},$$

and the Kalman gain K_t is updated recursively according to Appendix B.

The same procedure can be applied to estimate the pitch angle θ. Then, with ϕ, θ, we can compute the yaw angle from the magnetometer readings $\psi_{mag,t}$ according to Caruso et al. [49]

$$
\begin{aligned}
H_x &= m_{x,t}\cos(\theta) + m_{y,t}\sin(\theta)\cos(\phi) + m_{z,t}\cos(\phi)\sin(\theta) \\
H_y &= m_{y,t}\cos(\phi) - m_{z,t}\sin(\phi) \\
\psi_{mag,t} &= \tan^{-1}\left(\frac{-H_y}{H_x}\right),
\end{aligned} \tag{14}
$$

and then apply the same Kalman filter procedure described above with $u_t = \omega_{z,t}$ and $z_t = \psi_{mag,t}$.

B Extended Kalman Filter Implementation

General Description. The Kalman filter algorithm is described in Fig. 22. At each instant k, the algorithm receives u_k, which is the vector of control commands, $\hat{x}_k$, which is a vector that contain the predicted states obtained in the previous iteration, and the sensor readings y_k.

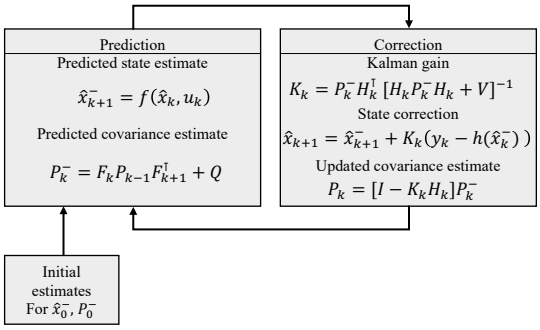


Fig. 22: General scheme of the Kalman filter algorithm.

The algorithm can be divided into two main routines: prediction, and correction. The first routine takes the last estimation $\hat{x}_k$ and the current input u_k and generates a prediction $\hat{x}_{k+1}^-$. However, this prediction has to be further corrected using the sensor data. Similarly, the covariance matrix of the estimation error P_k^- (i.e., the error between the real states x_k and the estimated states $\hat{x}_k$) is predicted using the process covariance matrix Q and the state transition matrix F_k, which will be defined later. The second routine takes the previous predictions $\hat{x}_k^-, P_k^-$, the observation matrix H_k, and the covariance of the sensor noise V, and computes the Kalman gain K_k. Therefore, the state prediction is corrected using the sensor readings and the covariance matrix is updated. The output of the algorithm is then $\hat{x}_{k+1}$ and P_k, which will feed the next iteration of the algorithm. There are several variations of the Kalman filter algorithm for linear and nonlinear systems – the main difference lies in the derivation of the transition matrix F_k and the observation matrix H_k.

Suppose there is a physical process with a set of states or variables $x_k \in \mathbb{R}^n$ that evolve over time, where $k = 1, 2, \ldots$ represent the k^{th} sampling instant (i.e., the k^{th} iteration of the algorithm) with a sampling period Δ_t. For example, x_k may represent position and velocity of a vehicle or temperature, pressure, and water level in a tank. The control input $u_k \in \mathbb{R}^m$ corresponds to the commands sent by the controller in order to achieve a specific goal based on the sensor measurements $y_k \in \mathbb{R}^p$. For instance, open a valve when the level of water is low, or increase the acceleration in a car to reach a desired velocity. The behavior of the process is approximately defined by a function $f(x_k, u_k)$, which depends on the current states and the control commands. In general, $f(x_k, u_k)$ can be defined using the laws of physics, or mechanical or electrical equations; however for complex systems, the function $f(x_k, u_k)$ is only an approximation due to uncertainties and assumptions (e.g., in certain conditions, friction of a wheel can be neglected or approximated).

In general, the main goal of the Kalman filter is to minimize the error between the real set of states x_k and the estimation $\hat{x}_k$. Thus, we can define the estimation error as $e_k = x_k - \hat{x}_k$. Due to the different sources of noise (e.g., sensor noise or external disturbances), e_k is also noisy, and that amount of noise can be quantified in terms of a covariance matrix P_k.

The Kalman filter algorithm is summarized in Fig. 22 and it can be divided into two main routines: prediction, and correction. The first routine takes the last estimation $\hat{x}_k$ and the current input u_k and generates a prediction $\hat{x}_{k+1}^-$. However, this prediction must be further corrected using the sensor data. Similarly, the covariance matrix of the estimation error P_k^- is predicted using the process covariance matrix Q and the state transition matrix F_k, which will be defined later. The second routine takes the previous predictions $\hat{x}_k^-, P_k^-$ and computes the Kalman gain K_k. Therefore, the state prediction is corrected using the sensor readings and the covariance matrix is updated. The output of the algorithm is then $\hat{x}_{k+1}$ and P_k, which will feed the next iteration of the algorithm.

For the extended Kalman Filter, the transition and observation matrices at each iteration k are defined in terms of the Jacobians (i.e., partial derivatives of a vector-valued function with respect to each of its variables)

$$
F_k = \left.\frac{\partial f}{\partial x}\right|_{\hat{x}_k, u_k}, \quad H_k = \left.\frac{\partial h}{\partial x}\right|_{\hat{x}_k^-}.
$$

Notice that the transition matrix F_k corresponds to the Jacobian of f evaluated in $\hat{x}_k, u_k$, while the observation matrix is computed by the Jacobian of h evaluated in $\hat{x}_k^-$. In general, EKF is a suboptimal algorithm due to the fact that the prediction of the covariance matrix P_k is only an approximation of the real one. This is because there are not analytical expressions to compute covariance matrices for nonlinear dynamic systems, and it is necessary to use Jacobians to compute that approximation.