2024

Functional Monitoring for Run-Time Assurance of a Real-Time Cyber Physical System

Matthew W. Gelber
Virginia Commonwealth University

VIRGINIA COMMONWEALTH UNIVERSITY

DISSERTATION

Functional Monitoring for Run-Time Assurance of a Real-Time Cyber Physical System

---

Author:
Matthew GELBER

Supervisor:
Dr. Robert KLENKE
Dr. Carl ELKS
Dr. Smitha GAUTHAM
Dr. Milos MANIC
Dr. Irfan AHMED

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University

Department of Electrical and Computer Engineering

Richmond, Virginia

February 5, 2024

VIRGINIA COMMONWEALTH UNIVERSITY

Abstract

Functional Monitoring for Run-Time Assurance of a Real-Time Cyber Physical System

by Matthew GELBER

As cyber-physical systems (CPS) become more integrated into everyday life, the security of these systems must also be considered during their development due to their ever-increasing importance. With the growth of physical components in the system, more autonomous control requirements, and increased dependence on proper functionality, verifying system safety and correct operation becomes increasingly difficult. CPS have become more complex through the combination of additional hardware and the resulting interconnected software in many layers, each requiring unique security solutions. One example of such a safety-critical CPS embedded system is the Flight Control System (FCS) of an Unmanned Aerial System (UAS). An FCS consists of many complex sensors which provide aircraft state information to a central processor to execute the autopilot flight control firmware. Developers of the FCS for these aircraft are dependent on a diverse supply chain for the sensors and processors used in these systems, and they cannot always ensure the trusted delivery of their verified firmware updates to the end user. In addition, the complex sensors necessary in an FCS may wear down and fail over time. These factors lead to system vulnerabilities from various types of cyber-attacks and physical faults of the sensors on a UAS. An architecture of a real-time functional monitor and associated detection techniques for run-time assurance has been developed to detect such cyber-attacks and sensor data faults in a UAS FCS. The results are demonstrated using an FPGA-based Hardware-in-the-Loop Simulation testbed for simulating attacks and the attack detection algorithms to provide the user with information regarding these sensor attacks and faults.

Acknowledgements

This project would not have been possible without the guidance and support of my advisor, Dr. Robert H. Klenke. Dr. Klenke has encouraged my continued education by providing me with the ability to pursue this degree in his research lab. I am grateful for his insights and explanations during the design of this project, and for helping to keep my goals in focus.

Next, I'd like to thank my committee members: Dr. Carl Elks, Dr. Smitha Gautham, Dr. Milos Manic, and Dr. Irfan Ahmed for their advice and feedback throughout this process. Also, I'd like to thank Peter Vaughan Truslow for his knowledge on anything and everything throughout our time together in the UAV lab. To my many peers at VCU for keeping me sane throughout this experience, including Lauren Linkous and Jonathan Lundquist, thank you.

A huge thank you to my parents for the opportunities and continuous encouragement they have given me that contributed to my pursuit of this degree, and to my brother for always being there for me. I would like to thank my friends for their continued support and confidence in me throughout this process.

Lastly, I give my most heartfelt appreciation to my beautiful wife, who has given her endless support throughout my graduate studies, without which this degree would not have been possible.

Contents

List of Figures

List of Tables

List of Abbreviations

CPS	**C**yber **P**hysical **S**ystems
FCS	**F**light **C**ontrol **S**ystem
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
GCS	**G**round **C**ontrol **S**tation
GPS	**G**lobal **P**ositioning **S**ystem
HECAD	**H**ierarchical **E**mbedded **C**yber **A**ttack **D**etector
HILS	**H**ardware-**I**n-The-**L**oop **S**imulation
ML	**M**achine **L**earning
RTA	**R**un-**T**ime **A**ssurance
RV	**R**un-Time **V**erification
STPA	**S**ystems-**T**heoretic **P**rocess **A**nalysis
UAS	**U**nmanned **A**erial **S**ystem
UAV	**U**nmanned **A**erial **V**ehicle
V&V	**V**erification & **V**alidation

Chapter 1

Introduction

Complex cyber-physical systems (CPS) are computer systems that observe specific aspects of the physical environment to determine proper controls of a larger mechanism or system. One such component of the cyber-physical system architecture is an embedded system. CPS are all around us in smart home systems (from home assistants to robot vacuums), medical devices, and autonomous ground and air vehicles, to name a few. CPS operate in the physical world, using computing control systems in the cyber world, and are comprised of both cyber and physical components that work together to ensure proper operation in the physical world. CPS use sensors to measure the physical characteristics of a system and/or environment, computing systems to perform the data collection and analysis, and control algorithms to provide actuation commands used to control or change the physical attributes of the system. Cyber-physical systems are also typically safety-critical systems meaning that the correct and safe operation is vital, otherwise any malfunction of a CPS can result in system failure, system damage, and possibly harm to the people in, near, or around the system. One example of such a safety-critical CPS is the Flight Control System (FCS) of an Unmanned Aerial System (UAS).

Many applications of UAS have recently gained increased attention, such as package delivery [1, 2, 3], Urban Air Mobility [4, 5, 6], structural inspection[7, 8], and facility inspection [9]. All the UAS applications mentioned have safety-critical roles and require the use of autonomous control systems.

CPS have become more complex through the combination of additional hardware and more interconnected software. The increase in software complexity is a result of the expansion of use-cases for autonomous control CPS, which requires new capabilities [10]. For example, software can be a pre-programmed control algorithm, like PID loops, or a neural network Machine Learning (ML) based control algorithm. ML based control algorithms offer improved performance but also increase complexity and difficulty of verification of safety during run-time. These ML systems are commonly considered a black-box component. Any black-box component requires monitoring to ensure accuracy and compliance to specifications to detect run-time violations and unexpected behaviors [11].

These complex CPS help minimize the impact of natural disturbances on the system, but generally do not address man-made disturbances such as cyber-attacks and physical attacks. Both of these attacks can result anywhere between system (physical) malfunction and failure to complete mission (total system) failure [12]. A UAS FCS uses information from various complex sensors to make actuation commands for a specific task. Most FCS do not have any built-in data validation or data verification, and thus must trust the sensors' values. This reliance leads to physical attacks on a sensor being an exploitable method for disrupting a system through data manipulation [13].

Because of the aforementioned characteristics, the UAS FCS is susceptible to cyber-attacks. Network, firmware, and sensor attacks are just a few categories of such cyber-attacks [14, 15]. Network attacks include denial of service, man-in-the-middle, false data injection, and replay attack. Firmware attacks include code injection, false data injection, and direct firmware modification of the controller. Sensor attacks include supply chain modification of the sensor's firmware, addition of a back door, false data injection, sensor spoofing, walk-off, and expansion of typical noise margin. An example FCS architecture showing the system level overview, as well as the potential locations and types of vulnerabilities to cyber-attacks, is shown in Figure 1.1.

An FCS typically has three main components: the sensors, the main processor, and the communication system. The sensors measure information about the aircraft's state and external environment such as position, velocities, accelerations, angular

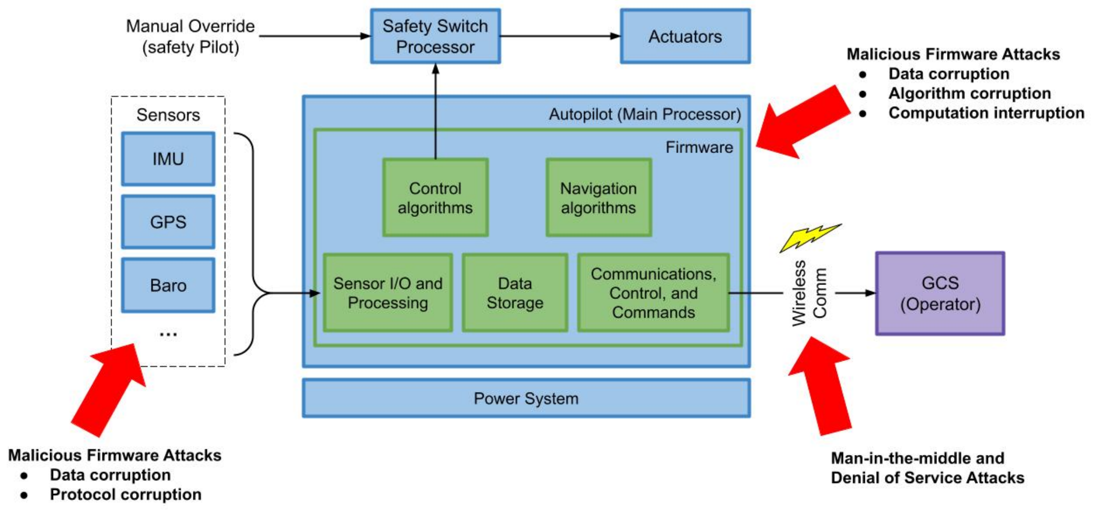


FIGURE 1.1: High Level Abstraction of a UAS FCS Architecture

rates, ambient temperature, and air pressure. The micro-controller, or main processor, contains control algorithms for the aircraft. This software consists of low-level drivers for retrieving information from the sensors and providing that information to the autopilot controller. The autopilot processes the sensor data, taking into account any predetermined control algorithm tuning parameters and other aircraft limits for use by the navigational algorithms. The navigational algorithms provide proper actuation outputs to guide the vehicle in the desired direction to perform a specific mission, if one is provided. The last component of an FCS is the data communication between the FCS and a Ground Control Station (GCS). The aircraft state, status, tuning parameters, and commands are transmitted over a wireless network between an operator at a GCS and the UAS.

Many of the common components of an FCS, such as the firmware, sensors, actuators, and communication, contain vulnerabilities that can be introduced during the manufacturing process, in the distribution chain, or at run-time through physical access or from other vulnerable components. These vulnerabilities lead to the attempted exploitation of such by way of human's malicious intent to disrupt a system. Malicious intent is based on manipulating hardware or software, or intentionally interfering with the supply chain by introducing unsecured or corrupted firmware at the manufacturing, distribution, assembly, or programming stage of any component. The communication between the main processor and its sensors and actuators is also open to vulnerabilities as is the navigational, control, and lower-level driver firmware of the main processor. Corrupted firmware, for example, can interfere with the communication protocol, such as between a sensor and the main processor running the control algorithms, or the sensor data itself could be manipulated through false data injection [16].

One common method used for ensuring proper CPS system behavior is through Run-Time Assurance, or RTA. RTA is based on a simplex architecture by combining Run-Time Verification (RV) with the Run-Time Monitor [17]. Run-Time Verification is the combination of the three main verification techniques (theorem proving, model checking, and testing) to be used during system run-time to look for evidence of the satisfaction or violation of a system property [18]. Run-Time Monitors use the detection information from the verification to determine if a simpler, back-up control algorithm should be used.

RTA is becoming increasingly more prevalent in the domain of autonomous systems as the most useful approach for enforcing safety of cyber physical systems [19]. Architectures of RTA monitors vary widely in their structure, which allows for greater usability and flexibility of such monitoring architectures. A CPS with run-time implemented monitoring typically consists of sensors, actuators, preset mission requirements, and a control algorithm which can be complex code or driven by machine learning. RTA can be used to ensure that the control algorithm (1) provides the correct actuation outputs based on sensor inputs, (2) ensure sensor values are accurate based on predetermined thresholds or compared with duplicated sensors, (3) the actuators perform the desired response based on the control algorithm used, and (4) that the entirety of system components work together to fulfill mission requirements.

1.1 Contributions

The research discussed herein focuses on the development of a functional monitoring architecture in support of Run-Time Assurance running in real-time on a CPS based process, specifically the Flight Control System of a UAS. To illustrate the high-level design of a functional monitor, Figure 1.2, adapted from [16], is displayed below. In this diagram, this simplified functional monitor contains the sensor data, communications processing, and the output actuator commands. These values are observed by the data validation block, which can represent any monitoring architecture or framework.

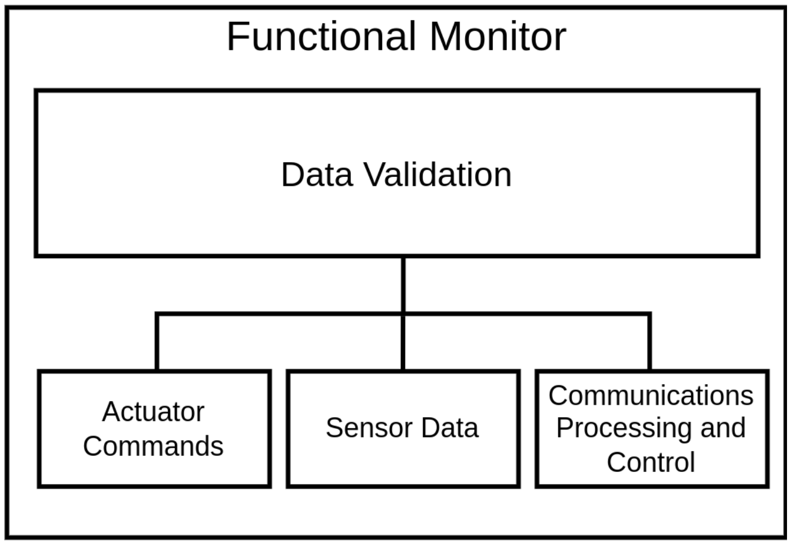


FIGURE 1.2: High-Level Functional Monitor Design

Functional monitoring is the process of continually observing a specific set of system inputs and any other accessible internal values to ensure that the expected outputs are being performed. If the actual outputs differ from expected ones, the typical functional monitoring architecture would provide information to the user about the anomaly. Depending on how robust and thorough the monitors are, they could even provide more detailed information on where the errors occurred. The functional monitor architecture developed in this research is intended to be placed between the sensors and the autopilot block of the high level abstraction of Figure 1.1. One of the biggest challenges in developing a functional monitor based upon the run-time assurance architecture is determining the system operational boundaries, which typically involve specific domain knowledge. Based on the review of existing Run-Time Assurance architectures (as discussed next in Chapter 2), there is no known method for analytically determining the boundaries of such a system.

The goals of this research are to determine those system boundaries and detection thresholds of a UAS through data collection, data analysis, and empirical modeling to classify both how the system bounds can be determined and what data is required to define the bounds. While existing physics-based aircraft dynamic models exist, no detailed aerodynamics model exists for a specific UAS flying with the VCU-developed Aries FCS. There are many instances where empirically determining boundaries for a Run-Time Assurance-based monitor in the absence of a simple but accurate dynamics model is helpful. As in this case, when a developer of a complex CPS does not have a detailed system model but still needs to characterize the limits of an autonomous system, an empirical approach must be used. More specifically, this research focuses on creating a functional monitoring architecture based on empirical models derived from observed system behavior. This method will be used to determine the system boundaries for use in a run-time implemented functional monitor in support of RTA.

1.2 Organization

This dissertation is organized as follows. Chapter 2 presents a literature review of run-time assurance monitors, the run-time verification framework, and existing methodologies for using run-time assurance with cyber-physical systems. A cyber-security testbed developed for the testing and verification of the work described herein in Chapter 3. The simulated walk-off attack methodologies implemented on the cyber-security testbed and used for testing the detection algorithms is discussed in Chapter 4. The main work of this dissertation is the development of the functional monitoring architecture as a subcomponent of Run-Time Assurance in Chapter 5, along with descriptions of its individual components are presented. The results from the implementation and testing of the monitoring architecture are presented in Chapter 6. Lastly, the conclusion and future work are discussed in Chapter 7.

Chapter 2

Background and Related Work

In safety critical cyber-physical systems, the cost and time required to verify safety has increased in recent years driven by the increase in system complexity. Since the construction of a CPS can vary widely, the resulting complexity of the safety assurance process is increased. Existing methods of ensuring CPS safety of both the software and hardware utilizes Verification and Validation (V&V). The V&V approach of safety assurance requires comprehensive testing of as many system states of software and hardware as possible at design time to determine the safety of a system. This process is not feasible with the ever-increasing complexity of both hardware and software and the quantity of components within a CPS. At the same time, the addition of these components to a complex CPS can significantly improve system performance when they are fully functional and accurate.

In addition, comprehensive testing for V&V to obtain a high-level of safety assurance is difficult. Typical model-based designs using traditional V&V approaches for safety focus on designing specifically for correctness of verification. Any design time models and testing simulations must consider the run-time characteristics and environments the safety specifications are designed to handle in order to accelerate the Verification and Validation process of such systems. These characteristics have driven the development of run-time architectures for online, real-time system analysis.

2.1 Simplex Overview

Most examples of Run-Time Assurance, including most of the existing work reviewed, are built on a form of the Simplex Architecture, also referred to as the switching strategies architecture [20, 21, 22, 23]. The primary focus of Simplex is to utilize simplicity to manage system complexity. Simplex works by monitoring the function of advanced autonomous control systems to predict violations of safety properties while maintaining lower development costs and increased system reliability. These predictions are based on predetermined boundaries in sufficient time to switch to a safer control algorithm.

The Simplex Architecture consists of a complex controller, safety controller, and decision module, as shown in Figure 2.1 (adapted from [24]). All three of these components receive sensor data about the external environment for use in triggering control actions (e.g., actuator commands), which are then sent to the plant, (e.g., the Unmanned Aerial System). The complex controller refers to the main software controlling the system, e.g., the UAS FCS (i.e., the autopilot). Some examples of Simplex for use in performing Run-Time Assurance of an FCS are shown in [19, 25, 26]. The safety controller is a simpler, more reliable, control algorithm that is tested at design time to be used to return a system to and maintain a safe state. The decision module is a component that decides which controller to use based on specified system operating requirements.

Current research involving the Simplex Architecture identifies two types of approaches, the application level and the system level [24]. The typical method of a Simplex-based architecture is the application level. During operation of Simplex at the application level, only the function of the application itself is monitored. Thus, safe handling of any bugs in the main software of the micro-controller or operating system is not guaranteed.

During operation of Simplex at the system level, software bugs in the main processor can be handled correctly if the proper hardware/software setup is used. System level Simplex monitoring is similar to the application level in that a safety controller and

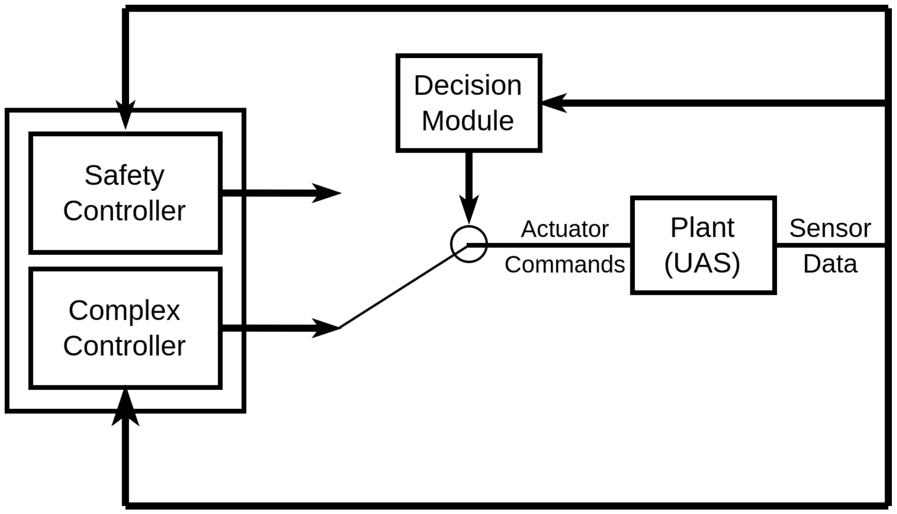


FIGURE 2.1: Simplex Architecture Logical View

decision module are used, with the unique distinction being in the location of the decision module component. In system level Simplex, dedicated hardware is used for the decision module, versus in the typical application level Simplex design, the decision module is part of the main system hardware.

The use of a dedicated hardware decision module in the system level method is based on the need for it to be isolated from any existing software bugs present in the main system. The goals of the system level approach are to not only ensure system safety from data errors but also protect from software bugs or other errors of the operating system and main control processing unit.

Implementing the decision module in dedicated hardware may increase system cost and power consumption, but it may also help maintain, or increase, system performance over the alternative approach. In addition, it does provide for more effective monitoring of system temporal properties. When implementing the Simplex Architecture, the entire design process can be conducted in a similar fashion to hardware/software co-design techniques. One such method under development which utilizes both the system and application level methodologies is the System Level Simplex Architecture discussed in [24] which is based on the application level architecture using components of the Architecture, Analysis, and Design Language (AADL) Simplex approach [27].

2.2 Run-Time Assurance

Run-Time Assurance (RTA) is an important addition to CPS to ensure full system safety. RTA is a control-based framework where a complex, high-performance control algorithm runs as normal but is supervised by a monitor. The monitor that looks for specific system behavior and switches the control algorithm to a simpler algorithm with known limits if the system behavior goes outside some predetermined bounds [28].

The inputs and outputs of any, or all system components, are observed through RTA to formally verify specified behavior or system state during operation. Run-time architectures require determining the boundaries of system behavior and its characteristics at run-time. The result is the relocation of system analysis to these newly developed run-time architectures. With the complexity of perception and control of complex CPS, RTA is designed to maintain safe system operation of required, but unverifiable, functional components when comprehensive V&V processes cannot be used. These RTA-based architectures are used to accurately detect system problems and then initiate a switch to a safer fallback control method to maintain safe operation.

The application of RTA to the FCS of an Unmanned Aerial System that is capable of autonomous flight is useful in analyzing: (1) system modes of operation that are only achievable in flight, (2) decisions by the autonomous control algorithm, (3) decisions based on ever-changing mission requirements, and (4) unexpected interactions between the vehicle and environment.

Run-Time Assurance frameworks are used to ensure safe operation of complex CPS that use advanced control algorithms and increasing numbers of complex sensors when these system components cannot be fully verified. The system contains a baseline control algorithm that can be reverted to by the advanced controller if deemed necessary during run-time by the RTA framework to maintain system safety and stability to minimize system loss. This baseline control algorithm is simple enough that it can be verified offline using comprehensive testing of V&V techniques. Run-Time Assurance is the basis for online safety certification of complex flight control systems, which was originally developed by Lockheed Martin and Barron Associates [25].

Utilizing RTA provides for online verification methods to ensure system safety through an analysis of the outputs of an unverified controller. RTA removes the need to redesign the control architecture every time any system subcomponent is added or changed. Another benefit is that unlike verification complexity, which increases as the main controller grows in complexity, safety verification with RTA does not require modifications, thus not increasing its complexity [19].

The authors of [29] develop an RTA model based on the Simplex Architecture. Their model utilizes a black and white-box approach for ensuring correct system behavior and optimal performance of an AI-based system. Their RTA model is designed for any CPS with a controller that uses AI in the form of machine learning. The black-box component is used to detect when a system is on a failure trajectory and switch to a known safer, and likely less effective, backup control algorithm. The white-box in this implementation is for predicting if the AI-based controller is uncertain of the proper action to take, thus potentially leading to an unsafe system state. When this condition is detected, the white-box switches to a backup controller, possibly saving the system from reaching the unsafe state altogether.

It is discussed in the work of [29] how this approach only works if no decision by the controller can lead to complete system failure, and if the safer controller is needed, the algorithm must be previously fully verified. This RTA method was tested on traffic light control of a small traffic grid, extended to a water treatment testbed [30], and to a hardware-in-the-loop smart city environment to determine the scalability and flexibility of this RTA approach. Throughout the testing, the authors of [29] aimed to determine the effects of communication delays, sensor malfunctions, and incomplete information on the performance of the RTA Architecture.

2.3 Run-Time Verification with Respect to Run-Time Assurance

When discussing the higher-level topic of Run-Time Assurance methodologies, an understanding of Run-Time Verification and how it fits into the RTA picture is required [17]. Run-Time Verification (RV) is a component of RTA where program verification is used to validate individual segments of a program's execution or functionality against a specification. The formal methodology of RV replaces offline program validation and model-checking, which have become less viable in larger applications and have a limited scope of verifiable properties. Run-Time Verification is tailored to analyzing the system implementation during online runtime and can be paired with a form of temporal logics to ensure specification compliance.

The purpose of RV applications is to verify that the system execution at run-time exhibits the same safety properties as could be verified by offline methods, such as V&V. This online verification is different from the online operations, depending on the architecture used for the RV application. The RV monitors can perform the same error detection as is usually conducted offline, except the monitor can have an additional component that serves as a decision maker. This decision maker can switch a system's control algorithm to a safer algorithm to guide the system back towards a safe state that is within the bounds of the safety property [31].

Run-Time Verification is typically used for continuous monitoring and analysis of a system for recovery potential based on specified boundary conditions. It is based on the original idea of program checking used to determine the accuracy of a program at run-time. Program checking has resulted in the development of run-time monitoring being usable for monitoring the system in real-time in instances where design-time verification is not possible. The main goal of the resulting Run-Time Verification is to detect run-time problems by providing a dynamic method for system analysis [32].

Run-Time Verification monitors are component based, meaning that simpler tests are conducted to check if a specific trace complies with a specific safety property resulting in the minimization, or removal all together, of offline comprehensive testing. The elimination of pre-deployment testing can be helpful in situations where comprehensive testing is not possible, either because of system complexity or lack of domain knowledge. Conducting comprehensive verification of such systems is difficult due to the innumerable combinations of system states and functional traces.

When utilizing methods such as system trace, a success or failure flag is typically reported based on whether the trace satisfies a safety property specification or not. Run-Time Verification has roots within program profiling where a normal and abnormal execution are run to analyze similarities and differences between the two results. The analysis of the results from this testing assume that the cause of any differences are from abnormal conditions and can be viewed as a precursor for runtime execution analysis [33].

The high-level architecture commonly used for RV is composed of a software monitor to observe the program's functionality, which is then compared to the system's run-time elements to verify safety requirements or safety specifications. The safety requirements are usually written using a form of temporal logic, state machines, or another plain English method of property specification. The inclusion of temporal specifications in RV requires model checking algorithms for use in certifying a system model from temporal logic-based properties [34]. In the typical temporal logic approach to system verification, algorithms are designed to check if the system generated state-based event trace complies with its corresponding safety property. This method relies on testing if the system as a whole falls within a set of safety properties. Although the name of RV alludes to its only use being online at run-time, RV can also be used offline during implementation, development, or any pre-deployment stage to monitor recorded simulations of the system to detect any errors.

Monitoring and verification methods are used in ongoing research for specifically designed RV with identified uses of debugging, testing, behavior analysis and recovery, safety property monitoring, and an overall understanding of system usage and operation. The application of these tools, such as run-time monitoring, is made possible by the concept that any system can be broken down into a series of events or system states, all of which have some operation characteristics that allow for transitioning between these states. It is these transitions that pave the way for processing with formalized specification [35], development of behavioral models, and algorithm and statistical analyses [36].

Run-Time Verification is commonly used when integrating untrustworthy hardware components, which contain their own dedicated, proprietary firmware, with an existing system. The theoretical RV approaches discussed within [37, 38, 39, 40, 41] are based upon architecting frameworks for specifying run-time properties by integrating existing RV monitors that use a form of temporal logic for semantic monitoring of safety properties [42, 43]. The Property Specification Language model checking is another method used to create a version of temporal logic to include time-based and regular expression-based formal verification techniques [44]. This extension on formal verification is achieved by expanding the use of monitors within finite execution traces and improving the property classification with respect to run-time monitoring.

The analysis of existing verification methodologies shows that most algorithms contain a monitor to check for variations of the system state then update a flag when a new observation fails a particular specification model. Current research of Run-Time Verification encompasses three main categories which work in conjunction with each other to ensure overall system safety and performance:

- Algorithms for ensuring adherence to property specifications of a system trace
- Instrumenting the system to collect trace data
- How this information supports detection, mitigation, and feedback used by another part of the system for switching states, provided to an observer through an alarm or flag, or a combination of the two

When analyzing the system trace and generating the algorithm that fits the required property, it is of vital importance to ensure that the correct observations are made, and sufficient data is collected. Too much data or incorrect trace analysis can produce increased overhead as a result of the addition of a monitoring architecture. The observations of system trace show the sequence of events including some history of performance and are typically combined into specifications such as regular expressions, temporal logic, and state transition analysis.

Continuing with the idea of ensuring accurate observations during system trace analysis, it is important to maintain adherence between the proper specification language and the desired safety property. The specification language alone provides a key insight into the type of traces being observed based on whether it is focusing solely on the system state changes, the values of any variables within each state and how they change, or some combination of the two as this integration would require additional properties. Due to the focus on safety properties, methods of analysis are centered around the transitions between two and only two states, or potentially a single state, leading to more application specific algorithms for determining property coherence.

The system recovery mechanisms and feedback are significant in preserving system functionality. The information gathered from the trace observations and safety properties are used to create a method with which to inspect performance to initiate a switch of control algorithms if the feedback received does not match the desired properties. An inaccuracy in these decisions can result in a false positive or false negative, leading to an unwarranted controller switch, reducing the performance, or not switching when necessary, causing a system failure.

Another area of active research in run-time monitoring is based on the importance of the feedback from the decisions in ensuring the data being analyzed from the run-time observations of the system is accurate. It is critical that this data is properly compared with prerecorded trace data to determine to what effect a property violation will have on the performance and how much the recovery controller can correct the problem. The feedback and switching methods vary slightly from the previous categories of RV in that, instead of specification languages for monitoring safety properties, more specific algorithms for safety property analysis are created and used to account for the ability of concurrent executions.

When attempting to conduct verification on analog and mixed signal systems, random behavior can be exhibited, making the verification difficult. Research conducted in [45] attempts to use statistical properties to determine a confidence measurement and error margins for system accuracy during RV. Challenges identified in verifying such systems include overall complexity, how specific the property is, how many resources the verification computation requires, and how random the behavior is as a result of using analog devices. These challenges create an RV implementation that is computationally intensive in both time and resources [45].

To maintain a high level of safety, a separate but purpose-built monitor is used to ensure safety goals are met and that the behavior is as expected. The basic architecture of the monitor is to analyze system variables and behaviors to compare against specifications for accurate run-time operation. The authors of [46] analyzed the usefulness of designing a monitor specifically to evaluate what conditions produce false positives or false negatives. They concluded that for the monitor in question to prevent false responses, the conditions must be previously noted and incorporated into the safety specification, which is not as feasible in a complex system with continuously changing states.

Run-time monitors can be used in any industry including automotive applications. In [47], another monitoring framework for use in offline and online processes was developed for fault detections based on formal specifications. This framework analyzes the relationship between system behavior and formal properties to draw connections between the run-time characteristics and possible faults. The faults identified are ones that cause a variation in system characteristics and performance from typical operation caused by hardware degradation, externally induced bit flipping, or both intentional and unintentional supply chain errors.

A difficulty in run-time detection of safety-critical system issues identified in [48] is from the increased complexity as a result of systems being more distributed and reactive. The objective of distinguishing between the source of the fault versus a result of the fault is a difficult task. The authors aimed to develop a systematic approach for fault detection of both source and result in distributed systems during run-time through the use of automatically created monitors from temporal logic specifications.

The authors of [48] note that simply using temporal logic-based properties does not account for every event sequence encountered during run-time. Their approach is based on reactive applications which are defined as systems with hard software deadlines where the environment dictates system needs. For example, in a flight control application, the sensors produce values based on the environment which are then used by the main processor to respond dynamically. The approach theorized must also take into account that safety properties, typically created from hazard and risk assessments from the system designers, are regarded as lookup tables between the cause and result and do not account for every real-world scenario. Minimization of overhead was also considered to ensure the system's normal execution is not affected by the monitors and the system would be mostly unaware of the addition of any software unless mitigation is deemed necessary.

In contrast to [48], detecting an error as well as diagnosing the origin are both simply considered faults in the work of [49] where no distinction is made between what caused a fault and the result of that fault. In the work of [50], identifying faults is conducted for use in system recovery and repair of the components. This method is described as a comprehensive behavioral model encompassing normal and faulty behavior and the interaction between these two states. Development of a comprehensive model is a difficult task since system behavior is continuously changing, which results in some fault identification being left out. Utilizing the system models, run-time monitors are created from deduction of system operation between normal operation and error filled operation to gain an understanding of its behavior. Another example of which includes the discrete event fault detection [51, 52].

A few categories and implementations of Run-Time Verification were researched and are discussed below. These categorizations include feedback and recovery controller switching in Section 2.3.1, the effects of timing and induced system overhead in Section 2.3.2, implementations of distributed monitors in Section 2.3.3, and challenges to Run-Time Verification and Run-Time Assurance in Section 2.3.4.

2.3.1 Feedback and Recovery Switching

The feedback information obtained from an RV monitor and the control algorithm for switching to a recovery mechanism is arguably the most important part of the run-time monitoring architecture, thus how a safety violation is handled is an important area of research. The main question is how to develop a recovery mechanism that accurately switches controllers to return a system to a safe state from an unsafe state as needed with minimal design-time verification being conducted.

When designing the switching component of a monitor, the goal is to maximize how much the advanced control logic is used without losing any performance or safety. The purpose of the switching component is to monitor online, during run-time, the system operation and performance of the control algorithm and flag for a switch to a baseline controller if necessary based on predetermined boundary conditions. One method used in modeling the switching system is through analyzing the various system states, such as present, target, safe, and unsafe [53, 54].

Existing research in the area of feedback and recovery focuses on at what point during operation to switch to a backup control method. While it is beneficial to determine if a system has crossed some predetermined threshold and is no longer operating within specified bounds, there is limited focus on how those boundaries are created. One such example is in [55] where the monitoring method is designed to ensure safe operation of a controller that includes black-box components based on predetermined safety bounds. These safety bounds were determined by the authors as the subset of states that are within the expected operation of the system. The recovery controller, utilizing the bounds as a safety envelope, selects the necessary controller based on the state of the system which was tested and can ensure the system remains in a safe state.

To ensure safety of autonomous systems that include complex black-box like components for the control algorithm and potentially the perception layer as well, new monitoring techniques must be developed as existing methods were not developed with this level of complexity in mind. Proper functionality of any black-box like component is crucial to system performance, therefore a monitoring architecture for system inputs and outputs to detect incorrect behavior must be developed. The most comprehensive monitoring architecture would have complete system access including input/output, internal states, all data signals, and system resources to compare data against trusted values, sources, and components, thus designating any component as trusted becomes an even more difficult task.

2.3.2 Timing and Delays

When monitoring architectures are added to a CPS, delays are introduced in the normal operation of the system. The timing impacts on both hardware and software interaction must be taken into account. There are various methods that can be used to help minimize such timing issues which are system dependent and include multiple processes or cores, process synchronization, and hardware timing can be considered. Any additional system configurations can also potentially result in extra inter-system communication overhead, which also needs to be taken into account.

With the addition of Run-Time Verification into a system, the extra time required for the monitor to check system variables and perform its analysis at run-time can result in increased overhead costs. Some existing research analyzes the trade-off between accuracy and monitoring overhead reduction with the hypothesis that in some cases lowering the detection accuracy of execution events to allow the monitor to perform its analysis may be worth the reduced system performance [56].

When attempting to detect timing violations, there is a tradeoff between detection speed and resource utilization. As noted by [57], to provide early detections, a high amount of computation resources are required, which may be infeasible for a specific implementation. The authors discussed a theoretical approach for efficient monitoring algorithms for timing requirements using Real-Time Logic to detect violations in a minimal amount of time. The algorithm used was implemented in the Java Run-Time Timing-Constraint Monitor (JRTM) for use with any Java application to monitor system timing requirements.

The goal of the JRTM is to collect timing information from the system during run-time for verification with predetermined timing constraints and if a violation is detected, a flag is raised. The benefits to this system include separation of the monitor from the main system being monitored, to reduce any interference and introduction of timing violations simply by this addition of the monitor, and minimization of overall overhead by limiting the amount of shared resources between the monitor and the main system.

One commonly noted challenge of conducting run-time monitoring is the potential overhead created by the addition of such a system. Depending on the implemented method of RV, the overhead is either negligible, i.e., as detached from the main processing as possible, or significant enough to cause a slow down in the system being monitored, leading to missed software deadlines and stale data used for critical decisions. Some common sources of monitoring overhead include how many locations of the main process are being monitored, how many monitors are processing similar data, and the execution resources utilized in monitoring.

Research on optimizing monitors to reduce overhead is theorized in [58] through analyzing the effects of assertion, temporal, and sequencing properties in a software health management approach. This work contains conceptual architectures but no implementable method, thus lacking experimental results towards their goal of determining if monitor optimization results in improved error detections.

Based on existing research on various styles of temporal logics and the RV frameworks that use those variants, the researchers of [31] attempted to develop a more generalized rule-based temporal logic methodology, called EAGLE, utilizing recursive equations for describing monitoring logics with the intention of creating a logic that encompasses the ideas of formal logics including interval, future, past-time, and statistics-based, along with finite traces, state machines, and regular expressions.

The idea behind interval logic is that a predefined logical formula can be evaluated within the current run-time interval such that the formula is satisfied by an interval sequence formula creating the assumption that it will hold true over the entire interval. Interval logic is a high-level framework for describing temporal relationships instead of individual state-based properties where a violation can be identified at any given time. Typical temporal logics do not provide information about an entire computation, only specific behaviors. Details of the entire computation are important when timing requirements are being analyzed, as those are typically represented in terms of interval durations [59, 60].

Past-time temporal logics methods differ from typical present-time temporal logics in that the present property satisfaction is based on past data from starting time to the present [61, 62]. Future-time temporal logics create inferences on what the expected states are based on the present information [62, 63, 64]. Finite traces are developed based on the idea that a path of a finite state graph is guaranteed to satisfy the safety properties of the formula used to generate the graph, as long as the finite execution of a system ends in an accept state and all properties were satisfied along the path trace [65, 66]. Lastly, regular expressions are useful in describing performance patterns succinctly and have many characteristics that allow for extensive patterns and are scalable [67].

The researchers of the EAGLE framework determined that the logic required was expensive and resulted in undecidable satisfiability, therefore resulting in the creation of a modified version where satisfiability checking is decidable which would allow for proper verification during run-time. The subsequent implementation to fulfill the requirement of satisfiable checking is the RULER framework [68], based upon EAGLE. The goal of this improved implementation is to develop a lower-level rule-based system using simpler logic for monitoring than EAGLE, since RULER's logic was designed to encompass the benefits of pairing RTA with any form of temporal logic.

A major component of CPS that makes it useful in real-world daily applications is its responsive nature where it reacts to external stimuli in the environment to create desired outputs or results within a timely manner [69]. Typical CPS are not designed to be free of faults, resulting in lack of functionality. This results in the need for a method to ensure correctness when faults are present. Work conducted by [70] analyzes the variability of the physical environment when timing constraints are not met, leading to unexpected outputs and undesired system run-time behavior based upon prior work of a generalized methodology for formally specifying run-time requirements in time-sensitive systems [28, 71].

2.3.3 Distributed Monitors

Run-Time Verification can also be applied to distributed CPS such as in the work of [70] and the evaluation of which are analyzed in [72]. It has a benefit over standard validation techniques on distributed systems since RV is not as time and resource intensive. A difficulty of implementing RV on a distributed system is that typical distributed CPS do not have a central clock for time syncing among all subsystems.

The researchers of [28, 71] expanded their work to include an analysis of distributed systems, which come with their own challenges of collecting information from multiple processes and the lack of clock synchronization across distributed platforms. The architecture records system performance as an event trace with the design requirements and expected system safety properties being monitored online using real-time logics against the event trace. If a property violation is detected by the monitor, all other monitors are notified, and the process institutes an adequate recovery option to maintain system safety. The monitoring framework is built specifically for distributed systems where each processor contains one monitor where a processor provides events to its dedicated monitor which can attempt to perform time synchronization with a specific algorithm and send the information about a property violation to the other monitors.

One example of the lack of a central clock causing difficulties is when attempting to use the timestamps from sensor data as a point of comparison, which can result in an inaccurate timeline of events. As a result of time-based sensor data comparisons not being the best option for data comparison, the proposed RV method uses the Satisfiability Modulo Theory approach [72]. The author's goal was to determine if the formula is satisfiable by creating written based properties, i.e., the safety property being monitored, by treating states as booleans and values as numerical while utilizing temporal operators such as until and eventually. Analyzing these written-based monitoring properties allows RV to provide information only when the safety property is fully satisfied. This is beneficial in that less information is transferred among multiple monitors.

An example method of distributed run-time monitoring is the SOTER framework containing RTA modules for each component, with as many as needed running in parallel [73]. The modules are designed with a high-performance controller for the data in question, a low-performance controller of safety-verified programming, and the desired safety specification. SOTER is another example that uses the Simplex Architecture as the basis for its design. The goal of this system is to minimize the limitations between system complexity and safety requirements employing a combination of design-time verification as well as environment verification at run-time utilizing the same switching methodology to a safer controller as necessary.

Another example of distributed run-time monitoring is the multilevel runtime monitoring of [74, 75]. The run-time monitors are implemented using Simulink to detect and mitigate system failures and attacks that are missed during the offline verification process. The authors claim that a single monitor is insufficient for detecting system failures, leading to the development of the multilevel architecture. This multilevel framework contains multiple unique monitors strategically placed in a CPS to provide a more comprehensive detection result.

2.3.4 Disadvantages and Challenges

With the many uses of RV for monitoring and maintaining safe system states, it isn't without its drawbacks. One such disadvantage to using RV as described in current research is in handling the additional temporal overhead created with the implementation of the three category RV approach [32].

The three categories of the Run-Time Verification approach to safety and performance are through system trace, system observation, and feedback and recovery analysis as previously discussed. The process of implementing correct trace observation methods, implementing temporal logic-based safety properties, and analyzing the data for recovery and feedback to the user and system can result in a less safe system. This decreased safety is attributed to these components reducing the performance of the normal system as compared to before the monitor was incorporated, depending on the specific method of implementation.

The real-time nature of the subcomponent of CPS in question within this research, embedded systems, results in complex causes of errors and anomalies [76]. The efficiency of a run-time monitor is important so as not to disturb the standard operation and introduce delays into a system. The time to conduct the switch is crucial to the safety and reliability of the system's performance if the monitor has correctly identified a property failure and initiated a recovery mechanism. If the additional temporal overhead is unavoidable, this must be taken into account when designing the RV system and its respective safety properties.

One known difficulty of RV is that offline testing tends to be more predictable, organized, and can be restarted whenever needed. Online monitoring is constantly changing and cannot be reset since the environment creates situations which cannot be simulated and the addition of RV can introduce behavioral changes. The efficiency of RV comes into question when these unexpected behaviors are added into the system as temporal properties are typically based on infinite system paths, thus the monitoring algorithms must attempt to account for known and unknown behavior of the finite data collected.

Some other difficulties that come with monitoring during run-time are a result of the variations in system control algorithms, specifically machine learning or adaptive algorithms, and codebase size. Systems with large codebases result in a higher likelihood of errors, which has led to a simpler controller used for providing the bare minimum requirements to maintain system safety [17].

Run-Time monitors require efficiency within the areas of safety property checking, reducing overhead of the monitoring algorithms, handling uncertainties within the system, what feedback information to provide, and how to develop the recovery controller. In order for these aspects to even be considered a possibility, selecting which safety properties to monitor is fundamental. Typically, overall system accuracy is analyzed by RV, which is also looked at during offline verification as well and often cannot provide any additional information when done online. This leads to the requirement for the development of distinct properties that must be satisfied during development and which are checked at run-time.

The high-level assurance and verification framework consists of two main categories of safety properties, the assurance properties that are satisfied through Run-Time Assurance and then the monitoring properties that are monitored by the RV model and recovery components of the framework. The assurance properties are mostly focused on system safety, accuracy, and performance and are considered more as a metric of future system state than components that are directly monitored. Monitoring properties differ from assurance properties in that they are based on data being directly monitored from a past system state.

The monitor provides information, including alarms or flags, about the system to a switching controller which is part of the recovery component with the goal of avoiding a safety property violation. In other words, to detect a failure of an assurance property, a corresponding monitoring property must fail first to indicate that an assurance property violation is close to occurring and should be avoided if possible. Most existing research maintains the separation of these two categories of properties with no suggestions on combining, or even translating between, monitoring and assurance properties.

2.4 Boundary Determination

RTA architectures utilize a set of boundaries to verify performance and correctness of a system to ensure maximum operation potential by maintaining safety constraints and proper backup controller operation. In an autonomous control system utilizing RTA for safety verification, the boundaries must first be determined by analyzing the relationships between input and output states and with the known environment.

These relationships can be used to develop a bounding algorithm through mathematical techniques to create a system model describing the environment and system boundaries. Once the model is created, its constraints can be verified offline using existing formal verification methods. The boundary analysis method has two major considerations, the first being the scope of input and output state relationship analysis and their significance in providing useful information for monitoring an adaptive system, and the second that all potential failure modes must be accounted for. An ongoing focus of current research is the development of bounding algorithms that result in a provable boundary and development of a complete safety property domain of the boundaries.

The development of boundaries through mathematical models to ensure compliance with all potential system states is computationally intensive and difficult even for offline calculation, depending on the dataset involved. Once the boundary models are generated, the monitor component of RV can then be created with the appropriate conditions to switch from the main, complex, control algorithm to the safer control algorithm. Ideally, the models and conditions to switch can be verified for correctness and reused for other parts of the system. One drawback to any mathematical methodology is that software overhead to analyze the data and initiate the switch is typically not accounted for in the mathematical models.

2.5 Design Time Formal Verification

Over the years, development of formal methods for analysis and validation of real-time systems has allowed for the construction of system specification properties through mathematical formalisms. While formal proofs for property specification are limited in scope and do not always provide a complete picture of offline system behavior, they can indicate some level of trust in particular components of a system. The lack of completeness of formal proofs is due mostly to the advances in system hardware and software, resulting in increased software complexity, as well as ensuring all aspects of the system behavior are tested is time-consuming and expensive financially and in computational resources.

Formal verification is not a commonly used method since correctness verification conducted in an offline system does not guarantee correctness during run-time. The divergence in correctness is categorized as design verification and implementation verification, where adherence to formalisms in the design does not guarantee adherence to the implementation. In an attempt to bridge this gap, testing predetermined inputs on the implementation can help provide some system confidence, but is still unable to assure full confidence of all possible input combinations. As a result, offline verification of any system is a difficult task, all while not providing confidence in the entire system, leading to the rise in research into online Run-Time Verification techniques to monitor continuous aspects of a system while it is performing its designated task.

The ensuing run-time assurance framework developed by the authors of [77] is the Monitoring and Checking (MaC) framework utilizing Java bytecode. The MaC framework is used to ensure adherence to formal specifications of a system during run-time to create specifications of low-level events from high-level events and the development of code for low-level event detection.

The methodology starts with a filter having inputs of low-level data and timestamps, which are used later, then passes the data to an event generator to convert the data into high-level events. The first two stages of the filter and event generator are components of the monitor. The generated events are then forwarded to the checking component, where events are compared with the requirements for consistency. The goals of the MaC architecture ensure correctness of the system during run-time in terms of the formal requirements through monitoring specific states and checking against formal specifications [78].

Initially, the MaC framework was designed around systems with fixed safety properties, referred to as hard real-time systems. The authors of [79] aim to expand this architecture to soft real-time systems using probabilistic models to verify system correctness. One such example of soft real-time systems, or more specifically a flexible system that allows for data to miss specific deadlines, that is under research in [79] is a wireless sensor network. In the wireless sensor network, data is transmitted and received at varying frequencies, sometimes even being repeated if no acknowledgements occur depending on the network protocol used. The constraints on the system behavior fall into the probabilistic category instead of concrete deadlines requiring to be met. One notable aspect of MaC is that extra implementation is required for it to be inserted into the system, potentially introducing errors or timing issues during run-time that were not present during offline testing before any such Run-Time Verification method, such as MaC, is added.

In contrast to most RV applications designed around ensuring correctness of computation with quantifiable property specifications, the extension of MaC, RT-MaC, adds verification for timeliness and reliability through quantitative time-based temporal logic specifiers and probabilistic models [80]. The addition of time-based components to system verification allows for time constraints, such as maximum time spent, when comparing system functionality with safety properties.

The difficulty in designing such an RV method is that the systems can diverge from its requirements based on several external factors, such as incorrect data packet format or signal loss, resulting in the combination of run-time system checking as well as the added assurance with probabilistic models that can help increase system confidence. Run-Time Verification typically is only able to track a single function trace for each model. This is where the probabilistic model checking is beneficial in providing improvements to a system with recurring events, such as any system that has a task scheduler [79].

Some current RV research utilizes models based on probability for predicting safety specification satisfaction of system behavior. Probability based Run-Time Verification methods using statistical models is one way the RV framework can be used for monitoring system performance [45, 46, 79, 81]. Another method is models with hidden states for detection of unobservable states, i.e., when a state cannot be determined from a series of unique inputs [82]. While many methods for run-time monitoring exist and have been applied to various systems, the uncertainty of complicated systems, e.g., UAVs, continues to be an area of safety concern for system developers. By incorporating model-based detections and methods to ensure a system can continue in the presence of a violation, such as the switching algorithms, is one area to review for uncertainty of a system.

Run-Time Verification for use is in continuous system performance observation of individual components to detect violations of safety properties with the goal of increased accuracy is considered by the authors of [81]. The approach specifies the properties with probability-based metrics of property satisfaction style monitors of the system for real-time monitoring and provide feedback for detection of safety property violation. One difficulty identified is that most of the systems in question have infinite possibilities of the control system output, therefore detecting violations must be done carefully on specific aspects of the system.

At present, most existing RV architectures require the addition of code to emit specific values that are of interest to the monitor. Typically, this extra code modifies the run-time aspects of the system, creating different run-time behaviors than in normal operation and potentially causing timing or general system errors. The requirement of efficiency in system trace monitoring during run-time has resulted in the development of various Java-based RV tools, such as Java PathExplorer, Java Monitoring and Checking, and Monitoring Oriented Programming [63, 78, 83, 84], to implement formal specifications such as temporal logics that have history for use in predictability of states and regular expressions [35, 42, 61, 77, 85].

Using the performance trace of a program to compare against preset properties to determine if any errors have occurred during operation is not a novel concept. Prior work on trace analysis has been conducted by the NASA Ames Research Center as part of the PathExplorer project. PathExplorer, along with many other Run-Time Verification approaches, are paired with a form of temporal logic for specification generation in systems with infinite trace possibilities, such as an operating system or control system for an autonomous application [61, 86, 87].

Some other publicly available tools that utilize forms of temporal logic for program and system accuracy are that of the Temporal Rover Project [85, 88] and Java PathFinder [63, 89]. In these two examples, temporal logic is used to express properties as program annotations and then replace those properties with executable code and other tools such as the Monitoring and Checking Toolset previously discussed.

The Temporal Rover Project focuses on the development of two efficient algorithms for verifying safety formulas using past-time logic through a rewriting-based approach followed by code synthesis from the safety formula. The goals of these algorithms are to analyze execution traces during run-time for error detection such as deadlocks, data race conditions, noncompliance with formal specifications, and errors in multi-threaded execution. Other unstudied aspects, in addition to the lack of scalability, are creating fault tolerance in the system through safety specifications and initiating recovery actions for a system being monitored with the safety algorithms. The algorithms developed are designed with a specific use case in mind, analyzing only a single execution trace and are not scalable. The non-scalability results in only being able to prove individual past-time logical safety properties are correct, since proving correctness of the entire system is not possible.

The Monitoring Oriented Programming (MOP) variation of run-time monitoring developed by [84] focuses on ensuring system reliability as a fundamental baseline in system implementation. The framework utilizes various methods for automatically generating monitors from preset properties and checks those against normal system operation to analyze the behavior for property violation and validation during run-time. Two instances of the MOP framework are introduced for verification and run-time monitoring; JavaMOP, for Java-based programs implemented in software, and BusMOP, for PCI bus analysis implemented in hardware.

JavaMOP Monitoring Oriented Programming as designed by [84] is based upon a high-level logic component consisting of logic plugins and a logic plugin manager. The logic plugins are used for generating the monitoring pseudocode based on the logic being used which is sent to the logic plugin manager having both a transmitter and receiver component. The manager receives a request from the specific MOP implementation which indicates the target programming language and issues a request to a specific logic plugin. The data received from the plugin is transmitted to the MOP Framework in use which requested it for use in the implementation. This specifically designed MOP Framework is split into a language translator which converts the pseudocode from the plugin manager to usable, implementable, code which is then used in conjunction with formulas extracted from the initial safety property. Lastly, each MOP architecture contains a user interface for interactions with other MOP Frameworks.

The benefit of MOP is that it is a generic run-time monitoring architecture for combining logical specification properties with the implementation suited for comparisons during runtime [90]. The toolset can then generate pseudo-coded monitors of the specifications to be used by another level of the framework to create programming language specific code. The purpose of creating pseudocode first is to ensure that MOP remains as generic as possible and can be used on a system in various programming languages. Monitoring Oriented Programming differs from other runtime monitoring implementations such as Aspect Oriented Programming, AOP, in that AOP is based more on monitoring specific system traces, whereas MOP is designed for a more automatic and generic approach through the use of logical plugins to allow for translation between various logical formalisms and various programming languages.

The MOP and other existing RV methods typically presume that all possible system states are accessible to the monitor, which is not always feasible with the software reliability and development expense of allowing access to all system variables.

BusMOP Another location of system faults comes from hardware degradation or manufacturing defects, which can lead to software unpredictability and unknown system state transitions and need to be considered when analyzing the accuracy of model observations. While RV architectures have been developed for various industries and applications, the monitoring components that slowly degrade over time is one such application that is not commonly reviewed in the literature.

Along with the embedded component of cyber-physical systems becoming more complex, developers are shifting focus by using more Commercial-Off-The-Shelf (COTS) components due to their continual development by the manufacturer which minimizes the development time and cost of custom components. One problem of COTS having continual development and updates from third parties is that they are considered black-box components. A drawback to black-box components is that correct functionality of a specific part, which should be considered at design time to meet formal requirements, is not guaranteed and difficult to verify its accuracy. This can be as a result of many factors including manufacturer reluctance to provide component specifications, details of non-standard implementation for a more application specific design, or malicious modification of drivers by a third party.

An attempt at creating a run-time monitor for COTS components was conducted in [91] based upon prior work of the MOP framework. The component's specifications are compared with its observable characteristics during runtime and if any violations are detected, a recovery mechanism is initiated to return a safe system state. The challenges with run-time monitor development of COTS components are that they contain unknown hardware and software aspects, and the location of any potential fault is difficult to determine.

The developed method, BusMOP, attempts to analyze the interaction between the hardware and software of the black-box component as well as the communication with the rest of the system. Typical run-time monitors introduce overhead through analyzing the current system variables and comparing the current system state with specifications. To eliminate overhead from the monitor, BusMOP performs bus sniffing, so no execution time is added to the main system and the comparisons are conducted on dedicated FPGA hardware to ensure no unnecessary system delays are created. One of the major benefits to this approach is not only the addition of negligible overhead, but also the main system requires no modifications and is unaware of the monitor unless a recovery mechanism is requested by BusMOP. The run-time monitor was created in the FPGA hardware to ensure the speed of processing matches the speed of modern COTS components due to their extremely optimized nature. The usefulness of this monitor variation is not illustrated with any data, and only states theoretically and algorithmically that the method developed will detect variations in COTS components with degradation.

Degradation of any system hardware component can result in uncertainty of that component leading to incorrect values, thus incorrect system state, which can become complete failure. The work conducted in [82] looks to expand the typical RV techniques utilizing model-based estimation to monitor safety specifications of cyber-physical systems with hardware uncertainty and hardware failures. This work aims to utilize the hidden state approach to minimize a common difficulty in creating a RV technique of the high cost of being able to access all system states. Temporal logic is again used in this implementation, as is commonly used in many RV architectures, to create the formal specifications as well as a probabilistic approach where the monitor computes the probability that the safety property is satisfied. The research concluded that the validations were successful on a limited system model and were not perfect in detecting the simulated plant model-based degradation.

2.6 Hierarchical Monitoring Architectures

Each run-time monitor implementation may have some similarities among certain components, each one is mostly unique in at least one of the following areas including the focus and goals for verification, type of system being verified, design requirements, and how the monitoring components are architected. A hierarchical monitoring architecture developed in [92] maintains some similarities with other run-time monitors reviewed in that the main focus is to develop the monitor that switches to a safe recovery controller, or state, if a violation of a predetermined safety property occurs. Safety property is defined as comparing monitored function outputs to sensors and physical components to create a holistic system view using handwritten formal specifications based on aviation regulations. Formal temporal logics for monitoring are used in the R2U2, the Responsive, Realizable, and Unobtrusive Unit, [93], RTLola [94], and falsification [95] of temporal logic frameworks. The hierarchical approach is split into five levels from the lowest layer of abstraction to the highest and shown below in Figure 2.2: item-level, system-level, aircraft-level, mission-level, and operation-level.

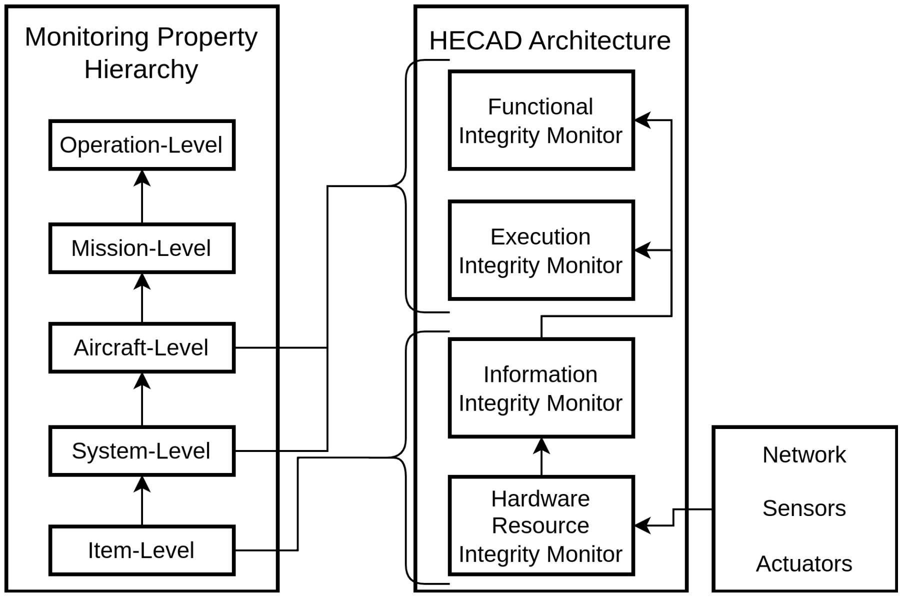

Monitoring Property Hierarchy
Operation-Level
Mission-Level
Aircraft-Level
System-Level
Item-Level
HECAD Architecture
Functional Integrity Monitor
Execution Integrity Monitor
Information Integrity Monitor
Hardware Resource Integrity Monitor
Network
Sensors
Actuators

FIGURE 2.2: Mapping of the Monitoring Property Hierarchy to the HECAD Architecture

2.6.1 Embedded Cyber Attack Architecture

The existing research into hierarchical monitoring from [92] closely resembles the ideas of HECAD, Hierarchical Embedded Cyber Attack Detector [16, 96, 97, 98]. HECAD, as developed, is a hierarchical architecture to secure a CPS from malicious attacks at every level of the system without impacting system functionality and performance. HECAD is a purpose designed run-time monitor for the FCS of an Unmanned Aerial System, UAS. HECAD is designed to monitor every component and all communication within a system in real-time with the capability to isolate a potentially compromised subsystem to prohibit any incorrect data from being processed by the main processor. Cyber-attack detections are achieved in HECAD through passively monitoring specific on-board sensors and components to not introduce additional vulnerabilities to the CPS through the inclusion of HECAD in the system.

The HECAD architecture is divided into four main subsystems: hardware resource integrity monitor (HRIM), information integrity monitor (I2M), functional integrity monitor (FIM), and execution integrity monitor (EIM). The HRIM functions as a bus protocol monitor between the autopilot and on-board sensors. Parsing and verifying the integrity of the data received from the HRIM is done by the I2M consisting of sensor sniffers. The FIM monitors the functional integrity of the system as a whole to verify that it behaves according to its specifications. Lastly, firmware vulnerabilities are taken into consideration with the EIM. For purposes of the research developed herein, which is based on the ideas of the FIM, it is assumed that the HRIM and I2M are fully implemented to provide their respective contribution to the higher-level monitor, specifically the functional integrity monitors.

Some similarities and differences between the monitoring hierarchy described in [92] and HECAD [16] were analyzed. The item-level closely resembles the HRIM and I2M of HECAD in ensuring sensors are operating properly providing complete and accurately formed data. The system-level and aircraft-level resemble the FIM and EIM although the scope of the system-level and aircraft-level both encompass the properties that the FIM and EIM are intended to monitor as well. The mission-level focuses on the feasibility of the mission requirements and the operation-level ensures operation constraints can be maintained which are both not found within the HECAD architecture. The hierarchies, their flow of data, and their respective mapping of comparable components is displayed in Figure 2.2.

A benefit to the functional monitor being a component of HECAD is that no software modifications are required on the CPS being monitored, in this research the FCS. The monitor proposed by [26], while complex and allows monitoring of all parts of the system, i.e. the sensor inputs, actuator outputs, and outer- and inner-loop control algorithms, requires software modifications to attach the monitor to all levels of system abstraction for data collection. While connecting the monitor to every component provides access to more data and many layers of data abstraction and can result in a more detailed determination of where and when exactly the system state left its predetermined operating bounds, degradation of performance and additional points of failure are likely introduced. Having an embedded monitor that is isolated from the main processing of the CPS ensures that system performance is unchanged, and no additional sources of error are introduced.

2.6.2 Importance of Accurate Sensors

Cyber-Physical Systems are a result of the combination of cyber based control and computing systems with the physical world sensors and actuators. The control systems require sensor and actuation data to be received and sent, respectively, for proper control, which introduces additional sources of attacks [75].

The term cyber-attack is typically used to identify any such malicious attack where the attacker does not need physical access to any equipment of the CPS and can use methods such as GPS satellite spoofing or Denial-of-Service and replay attacks on wireless based communication. This differs from the physical attacks where an attacker has physical access to a system to degrade the system through various means such as uploading malicious firmware at any point between sensor manufacturing and actual installation or programming into the CPS, altering actuation surfaces, or even removal of sensors altogether.

For the UAS application, autopilot systems require physical sensors for understanding of the environmental conditions and aircraft state. Information from various sensors is used to make actuation commands for a specific task. Most UAS do not have any built-in data validation or data verification and trust that the sensors record the correct values. This leads to physical attacks being an exploitable method for disrupting a system through data manipulation.

One of the most complex sensors used in a UAS FCS is the Global Positioning System (GPS) receiver. The GPS receiver uses RF signals received from orbiting GPS satellites, along with very complex calculations, to determine the position and altitude of the UAS for navigation purposes. In order to disrupt the GPS receiver's functionality, an attacker can produce alternate RF signals to confuse, or "spoof" the GPS receiver. These real-world scenarios of GPS spoofing are based on hijacking satellite signals by providing a stronger, more local, signal to send incorrect GPS data to the receiver to trick the system into producing an incorrect GPS location [99, 100, 101, 102]. If used on a UAV, this can potentially result in the UAV altering its course in an attempt to go back to a target location, and if implemented correctly, could result in the aircraft moving into the airspace of the person spoofing the signal for retrieval of the system. In the real-world, GPS signal spoofing is difficult due to the complexity of the GPS system and regulations controlling RF emissions. In addition to GPS signal spoofing, the complex algorithms used in the GPS receiver can be modified to provide incorrect or corrupted data. The GPS in this research is virtual so these types of spoofing and data manipulation can be simulated in order to mimic real-world GPS spoofing and other GPS attacks.

Another sensor whose data is vital for the proper interpretation of the movement of an autonomous UAS is the Inertial Measurement Unit, commonly referred to as the IMU. An IMU typically consists of a 3-axis accelerometer, a 3-axis gyroscope and a 3-axis magnetometer. Acceleration and rotational rate data is provided by the IMU to the autopilot controller for use in determining the aircraft's attitude (roll, pitch, and yaw) and movement over time. By manipulating these values, the calculation of the aircraft's orientation can be modified, leading the control software to believe that the aircraft is angled in a particular attitude requiring a correction, even if the aircraft is not physically in that attitude.

Chapter 3

Cyber-Security Testbed

The FCS cyber-security testbed described in [103] is an enabling technology that has been developed for implementation of the research presented herein. The construction and use of the testbed is not the main contribution of this dissertation but is presented in support of the development of the functional monitor. The testbed utilizes a real-time flight control system operating a fixed-wing aircraft via a hardware-in-the-loop simulation (HILS).

The testbed makes use of two Digilent Zybo Z7-20 Field Programmable Gate Array (FPGA) SoCs boards, one which performs the hardware-in-the-loop connection between the aircraft simulator and the flight control system, and the other which implements an embedded cyber-attack detector based on the HECAD architecture. These boards feature a Zynq-7000 System on a Chip (SoC) with Programmable Logic (PL) and a dual-core ARM Cortex A9 Processing System (PS) with the ability to run embedded Linux (Petalinux+Ubuntu). The PS of both ZYNQ SoCs runs a variation of Ubuntu 20.04 built for an armhf image for the embedded petalinux platform. To develop the PL, the Xilinx Vivado toolset was used to create the complete block design architectures for the testbed. Interaction between the FCS implemented in the PS and the sensor interfaces implemented in the PL was achieved using custom AXI cores, Xilinx AXI IP cores, and GPIO in the block designs. The custom AXI cores were used to ease the workload of the ARM processor in the PS, allowing for faster data access, higher accuracy, and eliminating the need for software interrupts.

The main processing component of the testbed is a modified version of the

VCU-developed Aries Flight Control System (FCS) [104, 105]. This modified version is unique in that it has seven new connections to attach to the PMOD interfaces of the Digilent Zybo boards. The connections for the sensors between the HILS and Aries are direct pass-through to the HECAD implementation to allow the HECAD board to "monitor" the data between the sensors (whether the data be from physical sensors in flight or the constructed sensor data from the flight simulator) and the main processor. The modified Aries FCS also does not include the physical sensor interfaces, as seen in the center of the hardware setup of Figure 3.1. These sensor interfaces are replaced with sensor buses connected to the HILS implementation to allow the sensor data to be derived from the aircraft simulator as described below.

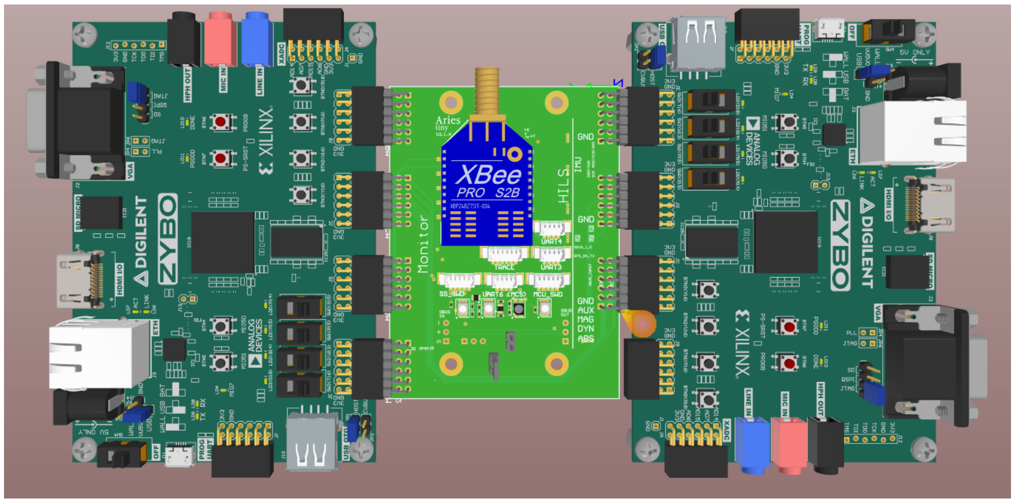


FIGURE 3.1: Testbed Hardware Setup

3.1 Hardware-in-the-Loop Simulation (HILS)

One board is dedicated to the hardware-in-the-loop simulation (HILS) interface [106], as shown on the right of the hardware setup figure. The HILS interface connects to the open source FlightGear aircraft simulation through the simulator's built-in Ethernet communication. The aircraft state data is received into HILS through this Ethernet interface through the Processing System (PS). A virtual sensor for each sensor typically used in the VCU-developed Aries FCS was implemented in the Programmable

Logic (PL) to imitate the appropriate sensor communication. This virtual sensor implementation means that no software modifications need to be made to the FCS, resulting in the FCS operating exactly as though it were in flight. The in-flight data is processed by the HILS PS and is converted to raw sensor data in the byte form and is stored in the same register locations in the PL, exactly as a real sensor would store the data. The method of implementation of the virtual sensors makes this setup a true hardware-in-the-loop simulation.

The Aries FCS has complete access to the data registers within the PL of the HILS in a read only direction through the PMOD 12-pin connections on the Zybo board. The Aries FCS retrieves the raw virtual sensor data utilizing the same communication protocol of each sensor typically used in real-world flight as is already part of the low-level drivers of the FCS. For example, the real airspeed sensor uses I^2C for communications to the FCS, so the virtual sensor implements the I^2C protocol in order for the FCS to not notice any difference in the source of data.

The code within the HILS PS has read and write access to the full set of data registers of the FPGA hardware. Each virtual sensor implemented along with its respective communication protocol is described in Table 3.1. The sensors include airspeed, barometer, battery current, GPS, magnetometer, and inertial measurement unit (IMU) sensors. The communication protocols implemented include I^2C, UART, and SPI for the virtual sensors and SBUS for communications with the aircraft's servo actuators.

Once the FCS calculates the appropriate actuator outputs, the HILS board receives those actuator commands from the FCS via the SBUS, and sends them back to the simulator through the Ethernet interface to "fly" the aircraft. During the simulation, the FCS also communicates with the Ground Control Station (GCS), as is typically done during normal flight operation, to allow an operator to control the aircraft's parameters and monitor the aircraft's flight.

UDP sockets are used to implement the communications between the flight simulator (i.e., FlightGear Flight Simulator) and the HILS interface. The PS receives

Virtual Sensors	Communication Protocol
Airspeed	I^2C
Barometer	I^2C
Battery Current	I^2C
GPS	UART
Magnetometer	I^2C
IMU	SPI

Actuation	Communication Protocol
User input	SBUS
Autopilot output	SBUS

TABLE 3.1: Virtual Sensors, Actuation, and their Corresponding Communication Protocols

a data packet containing information about the aircraft's state and updates the corresponding virtual sensor data. Another packet containing the current control surfaces information is also received by the PS. During the normal autopilot operation of the FCS, the sensor data is read from the FPGA and the controls packet is updated with new control signal values from the FCS and sent back to the flight simulator. The flight simulator is run with the specific UDP protocol, ports, and IP address of the FPGA, so the data packets can be sent and received correctly [107].

All aircraft dynamics and control data from FlightGear is parsed and converted to the correct format, with any necessary computations and unit conversions, and then stored into the specific data registers and offsets for each of the virtual sensors. The FCS runs in full autonomous autopilot mode, with the ability for the user to fly the simulator in other flight modes as well. The user input for control of the aircraft in any mode other than autopilot through a joystick, or an RC transmitter signal, is implemented in the PS through the SBUS protocol.

3.2 Hierarchical Embedded Cyber Attack Detector (HECAD)

In addition to the HILS board and the Aries FCS, the second Zynq SoC-based FPGA board in the testbed implements HECAD and is connected to the monitor side of the FCS as shown on the left of Figure 3.1. The same method of how the HILS board stores data in the PL was implemented in reverse on the HECAD board. To do this, HECAD implements sensor sniffers, described below, that retrieve the sensor data from the HILS PL. The data is then stored in the HECAD PL and converted back into human-readable information for further processing.

It is important to note that HECAD does not have the ability to modify any sensors' values, it is a stand-alone embedded monitor and does not affect the normal operation of the FCS. HECAD monitors the data buses of the Aries FCS, displays any user specified data, and performs attack detections based on predetermined detection algorithms. The process for determining the detection algorithms is part of the research described within this dissertation.

The points of interest of potential cyber-attacks and the overall high-level system architecture of the FCS cyber-security testbed developed in [103] is illustrated in Figure 3.2.

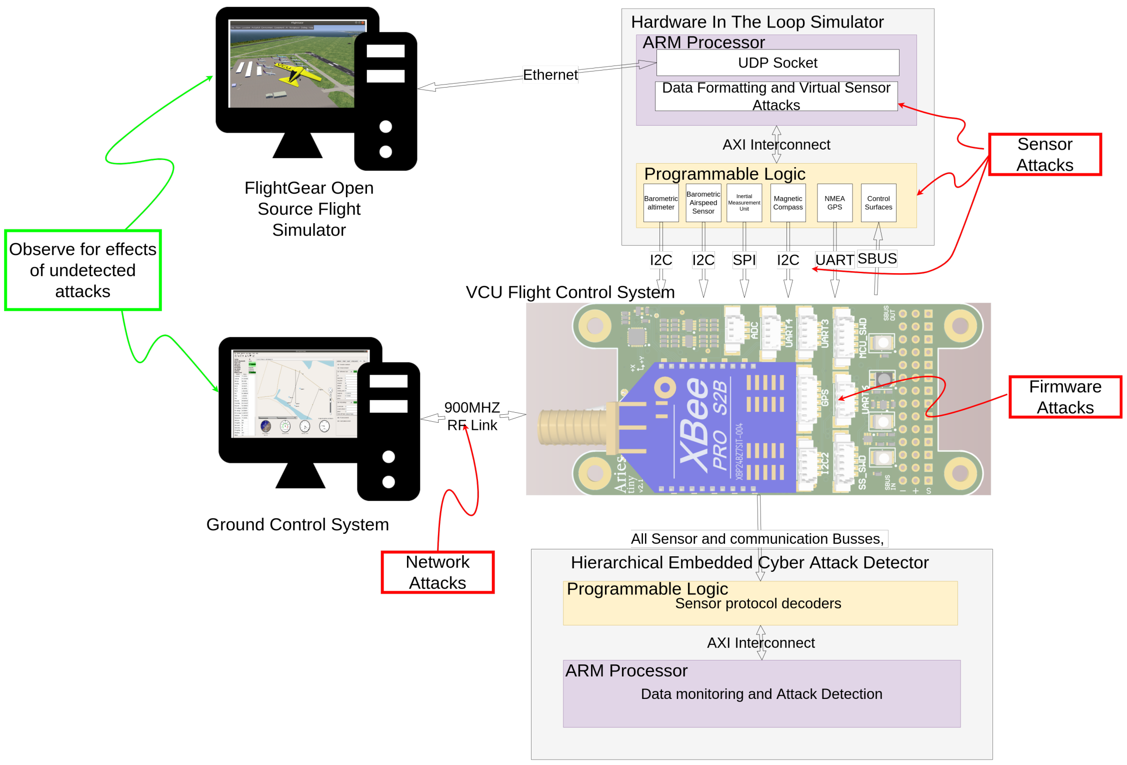

Hardware In The Loop Simulator
ARM Processor
UDP Socket
Data Formatting and Virtual Sensor Attacks
Ethernet
AXI Interconnect
Programmable Logic
Barometric altimeter
Barometric Airspeed Sensor
Inertial Measurement Unit
Magnetic Compass
NMEA GPS
Control Surfaces
Sensor Attacks
FlightGear Open Source Flight Simulator
Observe for effects of undetected attacks
I2C
I2C
SPI
I2C
UART
SBUS
VCU Flight Control System
Firmware Attacks
900MHZ RF Link
Ground Control System
Network Attacks
All Sensor and communication Busses,
Hierarchical Embedded Cyber Attack Detector
Programmable Logic
Sensor protocol decoders
AXI Interconnect
ARM Processor
Data monitoring and Attack Detection

FIGURE 3.2: FCS Cyber-Security Testbed Architecture

Similar to the virtual sensors that are implemented in the PL on HILS, HECAD incorporates the same sensors listed in Table 3.1 as sensor sniffers. The sensor sniffer updates its internal registers any time that register's data in the virtual sensor appears on the bus, be it from the master writing the corresponding register in the sensor sniffer, or the master reading that register's contents. This is unlike the virtual sensors where data is only written to by the master and read from if the master attempts a data read. The exact behavior varies per sensor, but generally the master first sends the address of the first register to be written or read. The sniffer captures that address in its register pointer and then any subsequent bytes in the transaction, regardless of direction, are saved into the internal registers of the sniffer. After each byte, the pointer is incremented, so that transactions of arbitrary length are supported. The sensor sniffers also implement the AXI Bus protocol, so that the contents of any sensor sniffer can be read whenever needed by the PS, as though they were a variable in RAM.

With access to the previously mentioned data, the aircraft state information, and the predefined detection algorithms, HECAD can parse and analyze specified data to locate any anomalies in the sensor values. HECAD is also interfaced with the GCS to provide the user with any desired information relating to the data anomalies, including specific values or plain text messages. In addition to information sent to the GCS, HECAD has full logging capabilities. The logging system includes sensor, aircraft state, debug values, values within the detection algorithms, and information related directly to the error detections. The logged data is saved to a comma-separated value file for analysis and graphing.

To display how lightweight the HILS and HECAD hardware implementations are, the post-implementation design report from Vivado was run. The FPGA utilization of the HILS implementation is shown below in Table 3.2 and the results for HECAD shown in Table 3.3. This data indicates that, except for I/O pins, the HILS and HECAD implementations use less than one quarter of the Zybo's FPGA resources.

Resource	Utilization	Available	Utilization %
LUT	8637	53200	16.23%
LUTRAM	95	17400	0.55%
FF	8966	106400	8.43%
IO	40	125	32.00%
BUFG	2	32	6.25%
MMCM	1	4	25.00%

TABLE 3.2: HILS FPGA Utilization Data

Resource	Utilization	Available	Utilization %
LUT	4790	53200	16.23%
LUTRAM	128	17400	0.55%
FF	6443	106400	8.43%
IO	42	125	32.00%
BUFG	2	32	6.25%
MMCM	1	4	25.00%

TABLE 3.3: HECAD FPGA Utilization Data

Chapter 4

Experiment Development and Study-Space Design

The simulated attack methodologies discussed within this chapter were implemented on the cyber-attack testbed described in Chapter 3. These simulated attacks are the experiments used to test the effectiveness of the Functional Monitoring Architecture developed within this dissertation. The processes for the development of the various attacks are discussed below. The goals of these attacks are to manipulate the aircraft's flight while avoiding detection by the existing autopilot algorithm. A method of attacks utilizing the walk-off process is just one method used for creating the simulated attack scenarios used to challenge the functional monitor.

In order to begin developing and implementing detections, simulated cyber-attacks were developed to test the effectiveness of the detection methodologies for each implemented sensor of the FCS setup. A Systems-Theoretic Process Analysis (STPA) [108, 109, 110] was conducted to determine the importance of the proper sensor values. The results from the STPA were used to influence the construction of various simulated walk-off attacks. A generic set of losses and hazards of the aircraft based on the system states during these simulated attacks were considered and are shown below. The losses identified are displayed in Table 4.1. The hazards along with their corresponding losses are shown in Table 4.2. The list of losses and hazards is not comprehensive, just a few of the more significant ones have been identified.

The losses and hazards identified above are just a few that correspond to the overall

Loss #	Loss Description
L-1	Property Damage
L-2	Loss of life or injury
L-3	Loss of mission objectives
L-4	Loss of aircraft

TABLE 4.1: STPA: Losses

Hazard #	Hazard Description	Link to losses
H-1	Instability of aircraft	L-1, L-2
H-2	Incorrect trajectory	L-3, L-4
H-3	Aerodynamic stall	L-1, L-2, L-3, L-4
H-4	Aircraft above the never exceed speed	L-1, L-2, L-3, L-4
H-5	Motor burnout	L-1, L-2, L-3, L-4

TABLE 4.2: STPA: Hazards

simulated attack results. A more detailed STPA was conducted on each of the simulated airspeed, altitude, GPS, and IMU attacks to show exactly how this type of walk-off attack can affect the aircraft's operation in flight.

Loss Scenarios related to Airspeed Sensor Data:

- Unsafe Control Action 1 — Data provided by the airspeed sensor is not consistent with change in GPS position over time when throttle output must be decided.
- Loss Scenario 1 — Data provided by the airspeed sensor is not consistent with change in GPS position over time when throttle output must be decided because of either a cyber-attack or sensor fault on the airspeed sensor and is no longer functioning as expected. The autopilot will still provide the throttle output to maintain the target airspeed, which can cause either the aircraft to lose airspeed, stall, and crash or increase throttle to max and exceed a maximum safe airspeed.

Loss Scenarios related to Barometric Pressure Sensor Data:

- Unsafe Control Action 2 — Data provided by the barometer is not consistent with reported GPS altitude over time when throttle and elevator outputs must be decided.

- Loss Scenario 2 — Data provided by the barometer is not consistent with reported GPS altitude over time when throttle and elevator outputs must be decided because of either a cyber-attack or sensor fault on the barometer and is no longer functioning as expected. The autopilot requires accurate altitude data to stay within any predefined altitude ceiling or to prevent a crash if the autopilot believes it needs to descend. Either situation could lead to loss of the aircraft, either through hitting the ground or flying to an unsafe altitude.

Loss Scenarios related to GPS Sensor Data:

- Unsafe Control Action 3 — Data provided by the GPS indicates a diversion from the waypoint path.

- Loss Scenario 3 — Data provided by the GPS indicates a diversion from the waypoint path because of either a cyber-attack, sensor spoofing, or general sensor fault on the GPS. The autopilot requires accurate GPS data to maintain a set flight path to complete the mission objective. The inaccuracy of GPS data can cause the autopilot to issue actuation commands to guide the aircraft back on course, resulting in the aircraft moving further away from the intended flight zone. This can lead to loss of aircraft and loss of the mission objectives.

Loss Scenarios related to the IMU Data:

- Unsafe Control Action 4 — Data provided by the IMU indicates a component of the aircraft's orientation has diverted from the ideal value for a particular stage in flight.

- Loss Scenario 4 — Data provided by the IMU indicates a component of the aircraft's orientation has diverted from the ideal value for a particular stage in flight because of either a cyber-attack or sensor fault on the IMU. The inaccuracy of the IMU in relation to an incorrect roll value can cause unintended turning of the aircraft. This change in direction can result in loss of mission objective, as the autopilot needs to correct for this change to attempt to return to the desired flight path.

- Loss Scenario 5 — Data provided by the IMU indicates a component of the aircraft's orientation has diverted from the ideal value for a particular stage in flight because of either a cyber-attack or sensor fault on the IMU. The inaccuracy of the IMU in relation to an incorrect pitch value would result in a similar loss to Loss Scenario 2 related to the altitude data.

- Loss Scenario 6 — Data provided by the IMU indicates a component of the aircraft's orientation has diverted from the ideal value for a particular stage in flight because of either a cyber-attack or sensor fault on the IMU. The inaccuracy of the IMU in relation to an incorrect yaw value would result in a similar loss to Loss Scenario 3 related to the GPS.

- Loss Scenario 7 — Data provided by the IMU indicates a component of the aircraft's orientation has diverted from the ideal value for a particular stage in flight because of either a cyber-attack or sensor fault on the IMU. The inaccuracy of the IMU in relation to any of the three axes could result in any number of unknown issues. The losses could be loss of aircraft, mission objectives, loss of life, and damage to property or people in the surrounding area.

The attacks developed modify the data being collected by the sensors in order to achieve a specific goal. The data modification during a simulated attack will be referred to as walk-off or data offset in this research. These simulated walk-off attacks are used as examples of the effects of intentionally placed malicious firmware in a sensor and how it can modify the reported values of sensors.

A walk-off attack involves the modification of the sensor data systematically to achieve a specific attack objective while attempting to avoid detection. A few other types of cyber and physical attacks exist including, interception, spoofing, falsification, repudiation, and man-in-the-middle. Interception is a type of attack used to breach the confidentiality of a system. One way in which an interception can be used is in relation to communication between systems. A spoofing attack is one where a malicious actor can create an untrusted, but similar, website for a firmware download, for example. The firmware for a system component, such as a sensor, can be a modified version of the real firmware with added vulnerabilities specific to the attacker's intent. Falsification is an attack method in which incorrect data is achieved by either data deletion, modification, or even being entirely unrelated to the original data. Falsification is often paired with repudiation in which a system lacks the ability to properly log where an error is coming from, resulting in that system unable to contradict which part of the system the issue originated [111, 112]. Lastly, a man-in-the-middle style attack is conducted by sniffing the data communication, interrupting the data transmission, and re-transmitting outdated data.

An example of a walk-off attack might be applying a cumulative offset to a particular sensor data value. To implement a simulated walk-off attack, the virtual sensor data, from the aircraft's state in FlightGear, is passed through a "simulated_attack" function where the function parameter and return type are a data structure containing the necessary data for each respective sensor.

An example pseudo-code implementation of the "simulated_attack" function for the ms5611 barometric pressure sensor is shown below in Algorithm 1. The *ms5611_data* data type is a structure containing two items of type double: (1) the air pressure and (2) the ambient temperature in Celsius. Normally, this data would be generated from readings of the physical environment by the sensor, but in this case, the data, such as the barometric pressure, in pascals, is calculated from the information provided by FlightGear. The input to the simulated attack function is the data for the specific sensor, and the output is the data of the same type after modification from the function. First the data is copied to a local variable with the same structure and then any variables containing walk-off/offset values and any static variable for accumulators are created.

The main section of this function is executed once the simulated data attack is initiated from the HILS board with a specific combination of switch positions and a button press for activation by the user. The I/O of the FPGA was utilized in this method to allow the user to implement any number of attacks at once by using the 4 switches as a 4-bit binary value and separate buttons for activating or deactivating the attack previously selected by the switches. If the attack is initiated, the accumulator variable

Algorithm 1: This pseudo code function shows the general idea of how a sensor data attack is conducted on the ms5611 barometric pressure sensor

```
function simulated_attack (ms5611_data _data);
Input  : The structure containing the data for a specific sensor
Output: The same data type with any modifications

ms5611_data data_post_attack = _data;

//variables used in conducting a sensor data modification attack
double pressure_offset = −3 ∗ 3.65;
//-3 ft/sec * conversion factor to convert ft/sec to pascals/sec
static double pressure_offset_accumulator = 0.0;

if attack is active then
    pressure_offset_accumulator += pressure_offset * $\frac{1}{loop\ rate\ in\ Hz}$;
    data_post_attack.press_pa -= pressure_offset_accumulator;
else
    pressure_offset_accumulator = 0.0;
    //reset accumulator to clear attack
end
return data_post_attack;
```

is used to aggregate the offset value, which is typically assigned as a per second value, multiplied by 1 over the loop rate of the function (200 Hz in the current HILS setup). This accumulator value is then added or subtracted to the pressure value based on the sign of the offset amount in this example. If no attack is initiated, the accumulator is held at 0.0 and no modification is made to the sensor data. The data structure for that sensor is returned from this function, and the virtual sensor registers in the PL are then set to this "modified" data. The FCS processes the potentially modified data from the registers resulting in the use of simulated sensor data that has been spoofed or attacked in some way. The following sections describe the walk-off attacks that have been implemented in the testbed.

4.1 Altitude Walk-Off

The altitude walk-off attack is completed by slowly changing the barometric pressure that is reported by the virtual barometric pressure sensor to the FCS. This change can be either increasing or decreasing the reported barometric pressure, depending on the desired effect. The FCS autopilot sees this pressure change as the aircraft changing altitude and will adjust the throttle accordingly to counteract this “false” altitude increase or decrease to maintain the target altitude. The indicated barometric altitude from the FCS, the FCS altitude, and absolute altitude (from the GPS) during an altitude walk-off attack are shown in Figure 4.1. As indicated in the figure, the barometric and FCS reported altitudes, the blue dotted line and grey dashed line respectively, remain at the set point of 200 feet. The orange long dashed line indicates when the altitude walk-off attack was started in software.

At the point where the barometric pressure increase walk-off attack occurs, at a rate of 3 ft/sec, the aircraft attempts to correct this change by descending, thus increasing the barometric pressure. The rate of increase of actual pressure and decrease in reported pressure are equalized, thus resulting in the FCS interpreting that it is remaining at its set altitude. The green solid line shows the absolute altitude of the aircraft from the GPS which indicates the aircraft is actually descending during the attack until it eventually crashes. This behavior is also seen in the simulator during system run-time. Over the run-time of this example altitude walk-off attack, the aircraft descends 200 feet in approximately 65 seconds, which matches the walk-off rate of 3 ft/sec.

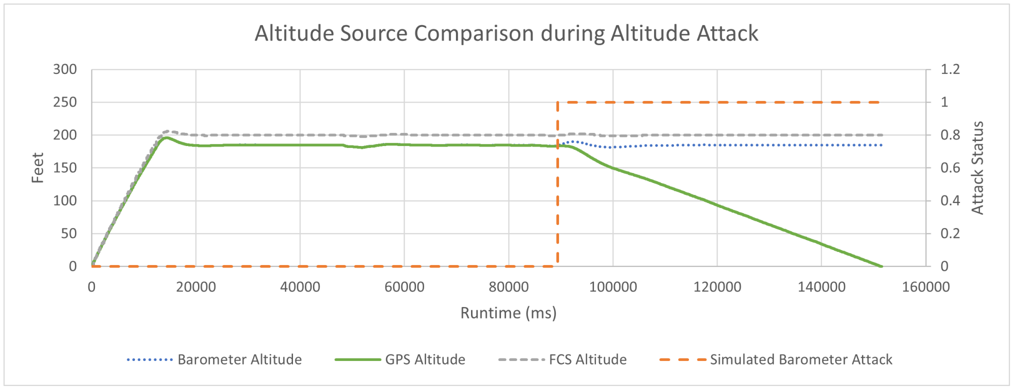


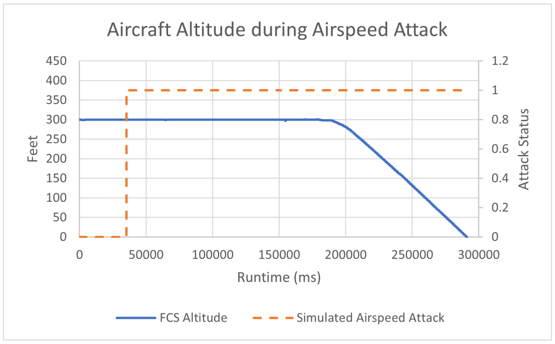
FIGURE 4.1: Comparison of Altitude Sources During Simulated Altitude Attack

4.2 Airspeed Walk-Off

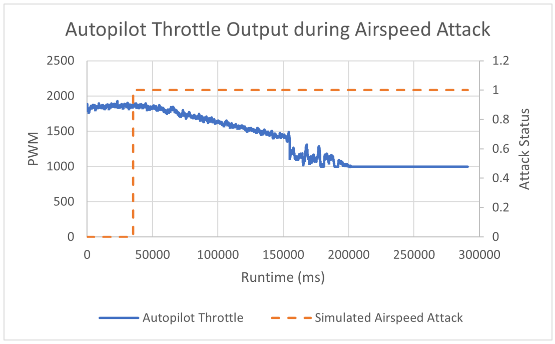
Similar to the altitude walk-off attack method, the airspeed walk-off attack is completed by slowly increasing or decreasing the ram air pressure, as measured in the actual aircraft by a pitot tube, used in the airspeed calculation. In this example scenario, the airspeed sensor value in the software was decreased at a rate of 2 pascals/sec. The slow rate of walk-off at the target airspeed of 60 knots calculates to about 0.1 knots/sec. As the cumulative walk-off starts to build up, the autopilot throttle begins to decrease very shortly after the simulated attack is initiated to return the aircraft to the set airspeed.

In the graphs of Figure 4.2 below, the orange dashed lines indicate when the airspeed walk-off occurred. Over the run-time, this results in the autopilot significantly dropping the throttle at approximately 15,000 ms of runtime where the aircraft begins to lose some altitude, as shown in the blue solid line of Figure 4.2a. The throttle, shown in Figure 4.2b, is reduced completely at the 20,000 ms point where the altitude begins to decrease until the aircraft crashes into the ground. In this situation, an actual aircraft might encounter an aerodynamic stall, also resulting in a crash, but the flight dynamics of the simulated aircraft in FlightGear do not allow for the simulation of an aerodynamic stall.

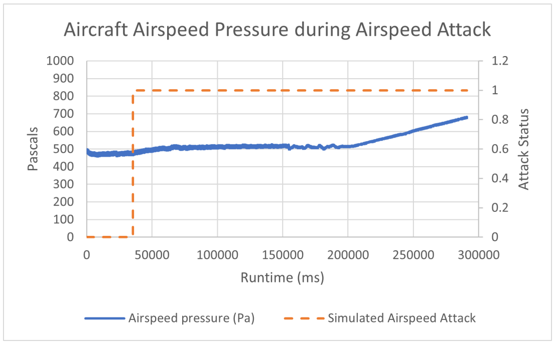
The effect of this attack on the ram air pressure data is shown in Figures 4.2c and 4.2d, respectively. The blue solid line in Figure 4.2c shows the airspeed pressure value in pascals from the virtual sensor. This is converted from pascals to knots, as shown by the solid blue line in Figure 4.2d.



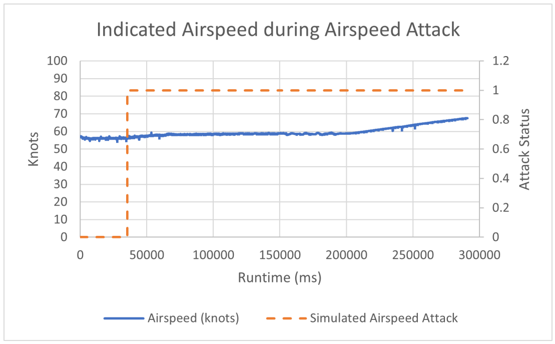
(A) Aircraft Altitude



(B) Autopilot Throttle PWM Output



(C) Ram Air Pressure for Airspeed Measurement



(D) Indicated Airspeed From Ram Air Pressure

FIGURE 4.2: Airspeed, Altitude, and Throttle Graphs During Simulated Airspeed Attack

4.3 GPS Walk-Off

The GPS sensor provides vital information needed to allow the aircraft to fly autonomously in autopilot mode. This sensor provides information about the aircraft's position and movement in 3D space, including latitude, longitude, altitude, heading, and ground speed. By modifying specific components of the GPS data, a simulated walk-off attack can be conducted to slowly alter the aircraft's flight path.

The GPS walk-off was initially completed by incrementally applying an offset to both the latitude and longitude values that were sent to the FCS. To determine the feasibility of this approach, the walk-off was tested on the longitude axis. This was conducted by applying an $1.0 * 10^{-5}$ deg/sec (approximately 3 ft/sec) offset to the GPS longitude value. As the offset is applied to the longitude value, the autopilot makes the necessary corrections, typically through aileron and throttle adjustments, to guide the aircraft back on course.

Figure 4.3 compares the preset waypoint flight path (indicated with the black dashed line), the autopilot's indicated latitude versus longitude location (illustrated by the blue solid line), and the actual aircraft location as a result of the walk-off of the GPS data (shown by the grey dotted line). The indicated location and actual location both start where the grey dot is displayed on the left-hand side. The aircraft flew in a counterclockwise direction, indicated by the blue and grey arrows. These arrows also show the end point at the end of run-time, when data collection was stopped. The longitude and latitude axes are displayed in both degrees and feet for readability.

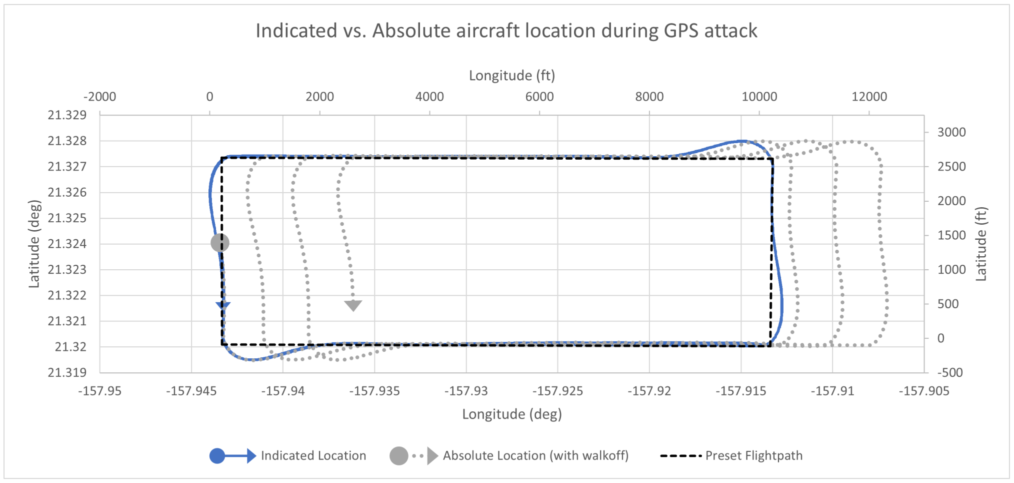
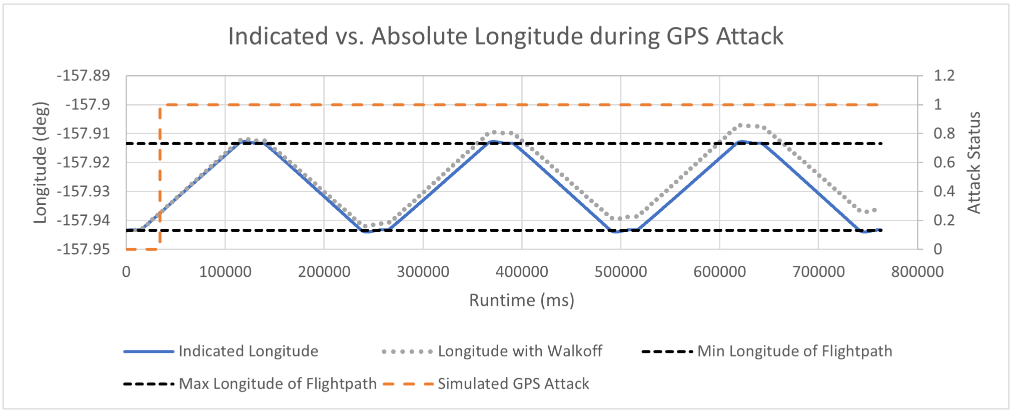


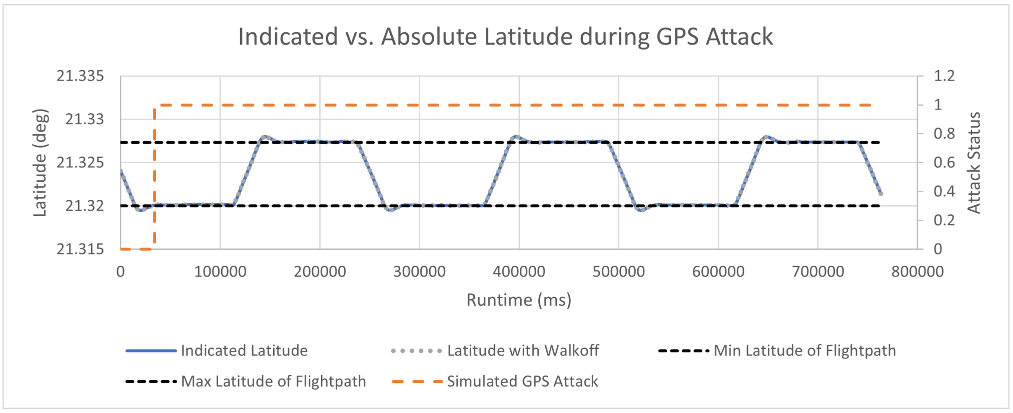
FIGURE 4.3: Flight Path with Longitude Walk-Off

Based on data obtained from Figures 4.4a and 4.4b, the attack run-time of the GPS attack was approximately 730 seconds. Over the course of this run-time with an offset of 3ft/sec, the longitude should have a walk-off of about 2,200 feet. Using the longitude axis in feet of Figure 4.3, this walk-off distance can be seen between the starting point shown by the grey circle and the endpoint by the grey arrow. The latitude and longitude data were displayed separately in Figures 4.4a and 4.4b to show the effect of this walk-off on the longitude axis and to illustrate that no latitude walk-off was conducted in this test.

This first method tested was determined to not be appropriate for conducting a GPS sensor walk-off, since it created mismatches of the vehicle's ground speed and heading measurement between the Kalman filter and GPS. This mismatch was an issue since



(A) Longitude Comparison Showing Successful Walk-Off



(B) Latitude Comparison Showing no Walk-Off Performed

FIGURE 4.4: Latitude and Longitude States During Simulated GPS Attack

the normal FCS functionality already provided warnings of the error, and the goal of the walk-off attacks is to be undetectable in normal operation by the FCS. To counter this effect, the attack was redesigned to account for these issues by targeting a heading walk-off and calculating the new corresponding latitude and longitude to maintain ground speed. The goal of this improved walk-off algorithm is to cause the aircraft to ultimately navigate to a desired target coordinate.

To spoof the GPS data in this simulated walk-off attack, the first step is to calculate the target heading of the target coordinates based upon the current heading of the aircraft. This heading difference is computed by first converting the latitude and longitude values of the current coordinates of the aircraft and target coordinates into radians, calculating the required quantities of X (Equation 4.1) and Y (Equation 4.2) to be inputs into Equation 4.3 to obtain the heading angle between the two points, and converting back to degrees. The difference between the target heading just calculated and the current heading is taken to achieve the heading difference, which is used to determine the proper direction the aircraft needs to turn. The heading difference is also used to calculate the velocity of walking off the heading, latitude, and longitude using a proportional controller. Lowering or raising the proportional gain can either slow down or speed up the walk-off velocity as the idea is to induce a fast enough walk-off to initiate a response from the autopilot to adjust the course but slow enough to not be detected by the Kalman Filter and reported as a GPS data error. The attack velocity is then scaled by the time between loops, dt, to obtain the desired heading offset that needs to be applied to the current heading to achieve the new heading, $Heading_{attack}$.

$$X = sin(longitude_{target} - longitude_{aircraft}) * cos(latitude_{aircraft}) \tag{4.1}$$

$$\begin{aligned} Y = &\Big(cos(latitude_{aircraft}) * sin(latitude_{target})\Big) \\ &- \Big(sin(latitude_{aircraft}) * cos(latitude_{target}) * (longitude_{target} - longitude_{aircraft})\Big) \end{aligned} \tag{4.2}$$

$$Target\ Heading_{radians} = atan2(X, Y) \tag{4.3}$$

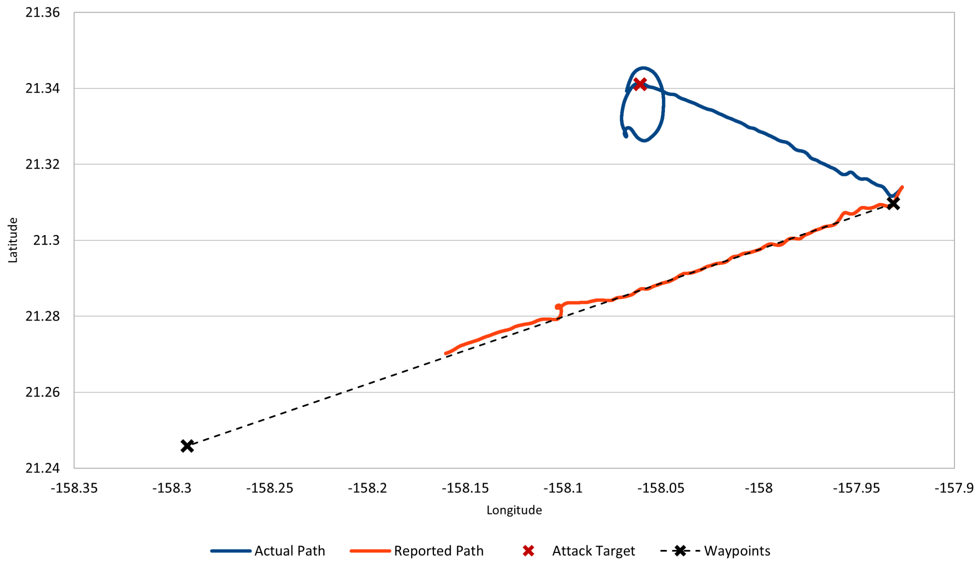
The latitude and longitude values must also be altered along with the heading incrementally to avoid the flight controller from ignoring or filtering out these values if a sudden change occurs. The new heading, with the attack offset applied from the previous step, is utilized along with the reported ground speed from the GPS of the aircraft to calculate a new latitude and longitude velocity. This new latitude and longitude velocity is used to calculate the spoofed latitude and longitude values. For the purposes of the following equations, the ground speed is converted into feet per second and all angles are in radians. The latitude velocity, according to Equation 4.4, first calculates the y-component of $Heading_{attack}$ and multiplies it with the ground speed and the resulting value is divided by the conversion from feet of latitude per second to degrees per second (i.e., 1 degree is equivalent to 364,000 feet). A similar method is applied to the longitude velocity of Equation 4.5 by calculating the x-component of $Heading_{attack}$ and multiplying with ground speed, then converting from feet of longitude per second to degrees of longitude per second. This conversion on the longitude axis is dependent on the cosine component of the current latitude.

$$V_{lat} = \frac{Ground\ speed_{fps} * cos(Heading_{attack})}{364000} \tag{4.4}$$

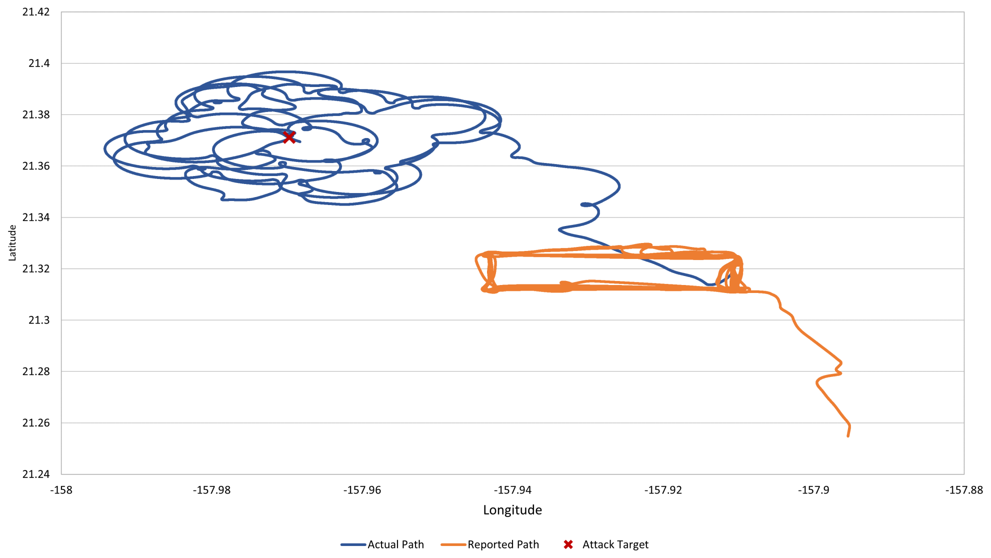
$$V_{lon} = \frac{Ground\ speed_{fps} * sin(Heading_{attack})}{364000 * cos(Latitude_{current})} \tag{4.5}$$

The resulting latitude and longitude velocities, V_{lat} and V_{lon}, are multiplied with dt, the time between two consecutive observations, to obtain the desired coordinate degree walk-off amount. The calculated walk-off amount is then applied to the previous latitude and longitude values, accordingly. The new values are then reported to the flight control system as it would normally receive data from GPS messages, without any knowledge the data has been manipulated causing the aircraft to slowly drifting off course. As indicated by the graphs in Figure 4.5, the reported flight path of the aircraft from the autopilot is indicated by the orange line, which closely follows the desired waypoint path. The blue line indicates the actual flight path when the simulated GPS walk-off attack is applied, showing that even early on in run-time, the aircraft drifts off its pre-programmed course and begins to head to the target coordinate designated by the attack, indicated by the red X.

In Figure 4.5a, once the aircraft reaches the desired attach coordinate, it begins to circle the coordinate. This circling is a result of the interaction of the attack with the autopilot control algorithm of the fixed-wing aircraft, as stopping fixed-wing aircraft in mid-air at any coordinate is impossible. The same test with a square waypoint sequence was conducted and the flight path is shown in Figure 4.5b. The shorter paths with more turns create an interesting response from the simulated attack, with the aircraft's heading being adjusted by the autopilot to follow the preset waypoint path and walk-off attempting to keep up and readjust at the same time. The algorithm must continually walk-off the GPS data to guide back to the target coordinates, resulting in a larger radius and uneven shaped encircling of the target.



(A) GPS Walk-Off Attack Run for 15 Minutes



(B) GPS Walk-off Attack Run for 70 Minutes

FIGURE 4.5: Aircraft Flight Path During GPS Attacks

Some limitations of the GPS attack algorithm are dependent on the attack velocity as well as the pre-programmed waypoint path shape and size. The attack performs as intended for long distance waypoints, where the aircraft makes few turns. However, when the attack is activated while the aircraft is flying a short-distance waypoint pattern, the attack will struggle to compensate for the constant turns, resulting in the aircraft taking a noticeably longer amount of time to reach the target. Attempts to mitigate this by increasing the attack velocity only during turns failed, as the attack cannot alter the course of the aircraft fast enough to compensate for the turns. Testing of values higher than 3.0° of change per second resulted in the aircraft crashing, so a smaller velocity was applied instead. This is believed to be a result of the difference between the yaw reported by the IMU and the heading reported by the GPS being too far apart, resulting in the Kalman filter not knowing what the true heading of the aircraft is.

4.4 IMU Walk-Off

The IMU sensor in use provides acceleration values in the units of G force in the x, y, and z directions and roll, pitch, and yaw rates in radians per second. In order to simulate an attack on the IMU sensor, the current accelerations and rotation rates must be rotated to create proper new accelerations and rotation rates after a specific amount of roll, pitch, or yaw walk-off is requested. This method allows for applying a walk-off to only one axis or all three simultaneously. To apply the rotations, the standard three-dimensional rotation matrix is used [113].

The simulated attack algorithm for the IMU takes as inputs the current gyroscope rotation rates, accelerometer accelerations, and the desired roll (ϕ), pitch (θ), and yaw (ψ) the attack is attempting to target. Using the rotation matrices, a new set of 3-axis rotation rates and acceleration values are computed by applying the desired roll, pitch, and yaw to the values reported from the flight simulator. These modified values are then stored back into the registers for the IMU data, and are reported to the FCS in the normal fashion that IMU would send its data.

The first IMU attack performed was a walk-off of the roll of the aircraft. While the autopilot is conducting a waypoint sequence, the data from the IMU sensor is rotated with a roll offset of 6 degrees. This value is small enough to cause a visible change in the data, yet maintain stable flight and avoid having the aircraft lose control and crash. Figure 4.6 below shows the result of the roll angle reported by the autopilot in blue, the actual roll angle from the flight simulator in grey, and the dashed orange line indicated when the attack was initiated.

The waypoint sequence under test was set up in a square formation. The spikes in the roll data occur during turns at the corners of the square. Before the attack is turned on, the actual roll angle and autopilot's reported roll angle are consistent, but after the attack is initiated, the actual and reported angle quickly start varying from each other. This difference can be seen in the long flat sections of the graph when the goal was to fly straight and level as well as in the turns where the angles differ by a small amount.

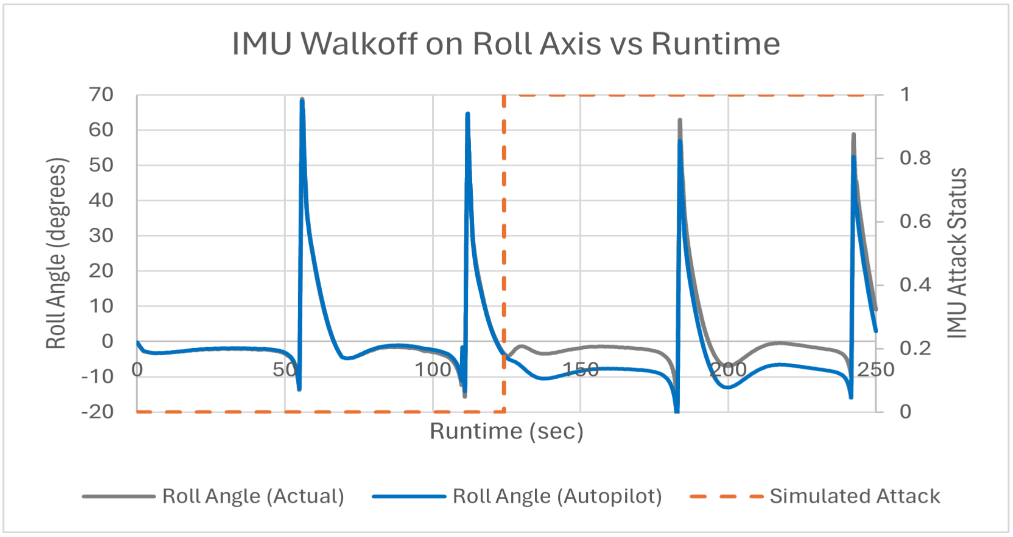


FIGURE 4.6: IMU Walk-Off Attack on Roll Axis

The same waypoint sequence method of attack and was run when conducting a pitch angle walk-off attack. As indicated in Figure 4.7, before the attack was initiated, the autopilot reported pitch angle and actual pitch angles vary slightly as a result of the autopilots constant corrections in pitch with minor airspeed and elevator adjustments. Similar to the roll walk-off, when the pitch walk-off is started, the actual and reported pitch angles diverge, creating the difference shown in Figure 4.7.

While the yaw walk-off was conducted in the same method as roll and pitch, a difference response to the attack was observed. As seen in Figure 4.8, before the attack

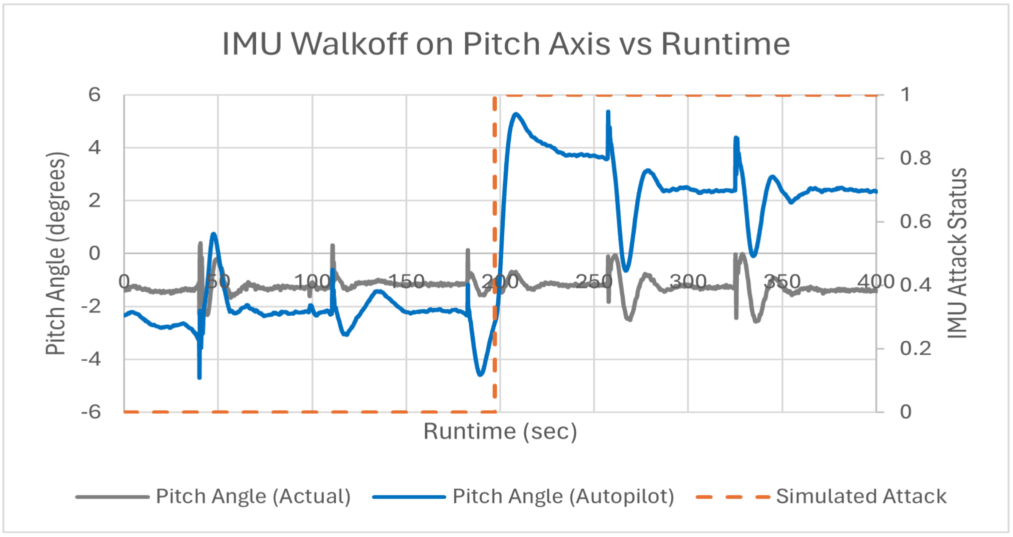


FIGURE 4.7: IMU Walk-Off Attack on Pitch Axis

was started, the actual yaw angle and autopilot reported yaw angle differ slightly as a result of data measurements for this value coming from the IMU and GPS which are then combined by the Kalman filter to create a proper yaw measurement. When the attack is initiated on the yaw value, the data from the IMU begins to differ significantly enough from the data reported by the GPS, which results in the Kalman filter putting a heavier emphasis on the GPS data. Eventually, the Kalman filter changes its weights for yaw determination such that it fully relies on the GPS data to calculate the yaw value.

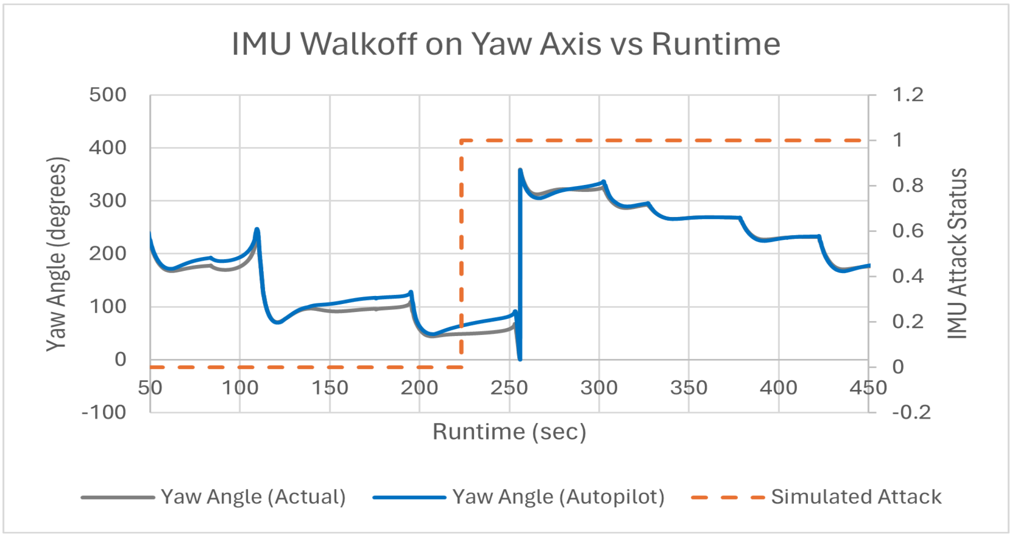


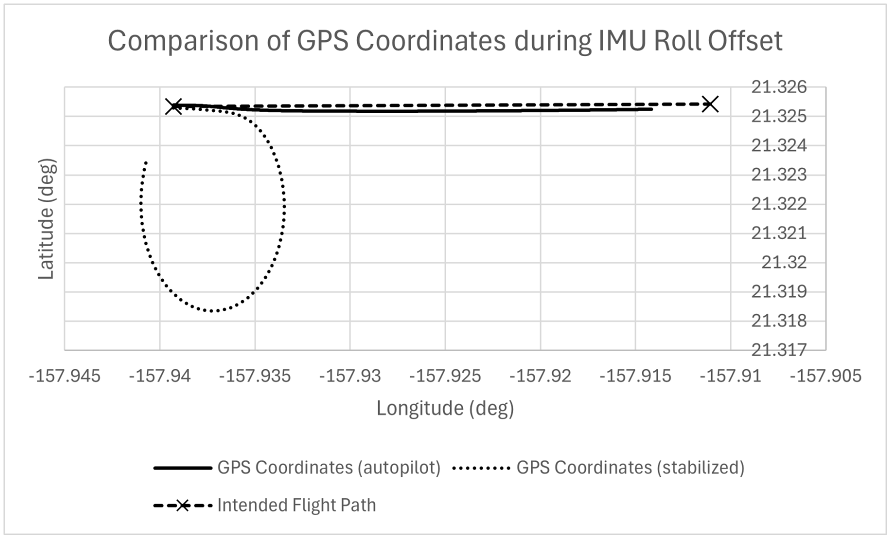
FIGURE 4.8: IMU Walk-Off Attack on Yaw Axis

To illustrate the effects that IMU roll walk-offs have on the FCS, data from the aircraft's flight in both autonomous autopilot and stabilized flight modes was collected and is shown below in the graphs of Figure 4.9. For the purposes of these observations, the IMU roll walk-off was conducted in the same method as previously described with an offset value of 10.0°. In stabilized flight mode, the FCS was set to maintain level flight, i.e., 0.0° roll and pitch angle, without reference to the aircraft's flight path with respect to the GPS coordinates. In contrast, the primary goal of the FCS in autonomous autopilot mode is to follow the waypoint navigation sequence, even if that means changing the orientation of the aircraft to a non-level orientation.

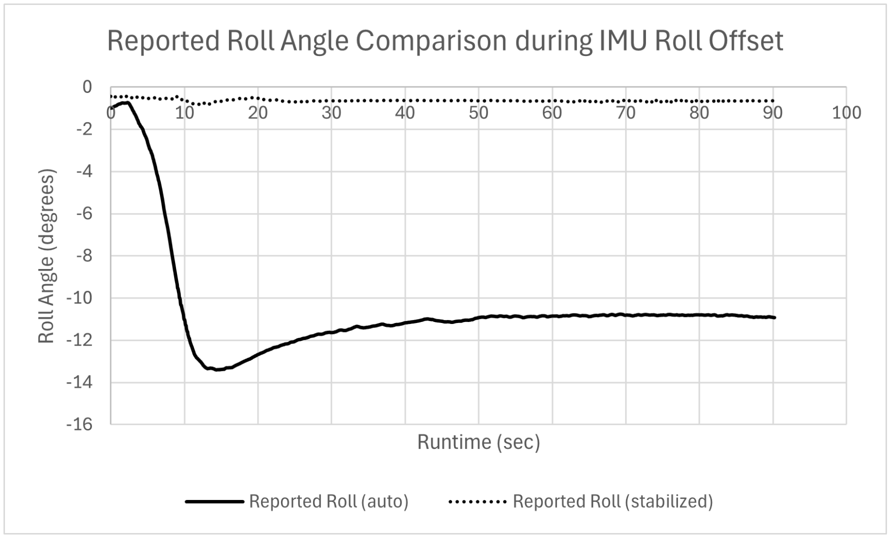
Figure 4.9a on the left-hand side shows a comparison of the latitude and longitude GPS coordinates reported by the FCS, and Figure 4.9b on the right-hand side displays the reported roll value. In both graphs, the data collected during autonomous autopilot flight is indicated by the solid line, data collected during stabilized flight indicated by the dotted line, and for the GPS comparison graph the intended flight path of the aircraft shown by the dashed line.

When the IMU roll walk-off attack was applied to the FCS while it was in stabilized flight mode, the aircraft begins to turn to the right (a negative roll angle), eventually flying in circles if left long enough, since a positive 10.0° roll offset is applied to the IMU data being provided to the FCS. The resulting turn induced on the aircraft is due to the stabilization controller receiving the IMU data indicating the aircraft is rolling to the left, thus commanding the aircraft's control surfaces to roll the aircraft in the opposite direction to maintain level flight. Since the aircraft did not actually have a positive roll, and only the manipulated sensor data indicated that it did, the result is that the aircraft physically rolls in the direction commanded by the FCS to return the reported roll of the aircraft to level. As seen in the graph on the right, the corresponding roll value reported by the FCS remains approximately 0.0°, as this is the target angle for the stabilization controller. The result is that the aircraft turns to the right as a result of an IMU roll walk-off to the left during stabilized mode.

The effects of the IMU roll walk-off attack are exhibited differently during



(A) Comparison of GPS Coordinates



(B) Comparison of Roll Values

FIGURE 4.9: A Comparison of IMU Roll Walk-Off Response in Both Autonomous Autopilot and Stabilized Flight Modes

autonomous autopilot flight. In autonomous autopilot mode, the flight path of the aircraft even with the roll offset maintains the intended course, as indicated by the dashed line in Figure 4.9a on the left. This reaction does not reflect the reported roll value from the FCS, shown in Figure 4.9b by the solid line on the right, which is the opposite of the offset applied (i.e., the offset of +10.0° is applied, but the reported roll angle is -10.0°).

When the attack is initiated, the roll from the IMU sensor begins to change, but the aircraft has not physically changed orientation. As a result, the autopilot continually makes minor corrections to the proper axis. The FCS assumes the output actuations induced the intended effect, in this case rolling the aircraft the opposite direction, to return it to "level". Leveling the aircraft causes the reported roll value to change and stabilize on the negative of the attacked offset. In order to force the aircraft off course during an autopilot flight, the offset value must be greater than the maximum roll angles programmed into the FCS which would result in the autopilot controller being unable to compensate for the offset to bring the aircraft back on course.

In summary, when applying a small roll offset to the IMU data during autonomous autopilot mode, the aircraft maintains level flight and continues following its intended course by reporting a roll offset of the negative value of the offset. However, during stabilized flight, the reported roll angle is 0.0° but the aircraft physically begins rolling, changing the flight path. These two different responses to the same simulated attack make for a more complicated model to detect this type of walk-off attack.

Chapter 5

Functional Monitor Architecture and Implementation

FCS Architecture

The methods for simulated attacks introduced in Chapter 4 show the effectiveness of data manipulation in altering the flight of a UAS and the resulting difficulties of detecting such variations. This chapter describes the process for development of the Functional Monitor Architecture and the work conducted for creating the methods to perform detections on the FCS.

An important aspect to keep in mind when developing detection algorithms is understanding how the information from the sensors, input commands, and output actuations are used and interpreted within the Flight Control System. To better understand the interconnections of the operations of the FCS, the high-level functional block diagram of the FCS was constructed as shown below in Figure 5.1.

The two main components of the FCS functional block diagram are the Peripheral Drivers and the Kalman Filter. To provide additional information related to the outputs of the peripheral drivers (i.e., the sensors), a zoomed in view of the peripherals, their respective outputs, and the data flow is shown in Figure 5.2. As shown in the figure, all the sensor data, except the altitude from the GPS, is fed directly into the Kalman Filter. The purpose of consulting this diagram of the flow of data between the sensors and the Kalman filter is to ensure that any values being compared for detecting sensor anomalies are not all affected by the Kalman Filter. For example, to compare the vertical velocity

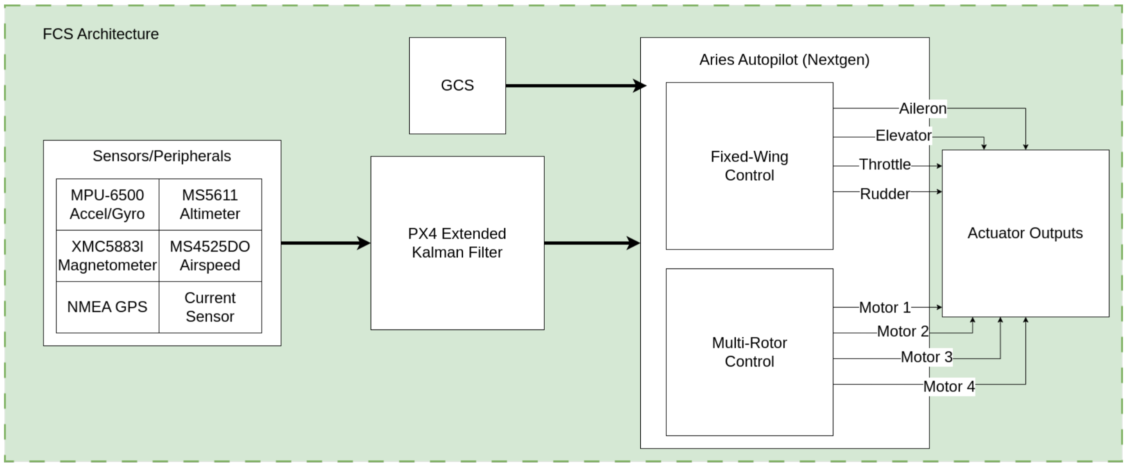


FIGURE 5.1: FCS High-Level Block Diagram

data of the aircraft, the derivative of the altitude sensor can be used in comparison with the vertical component of the velocity from the Kalman Filter. This is possible since only one of these observations is a computed estimate from the Kalman Filter.

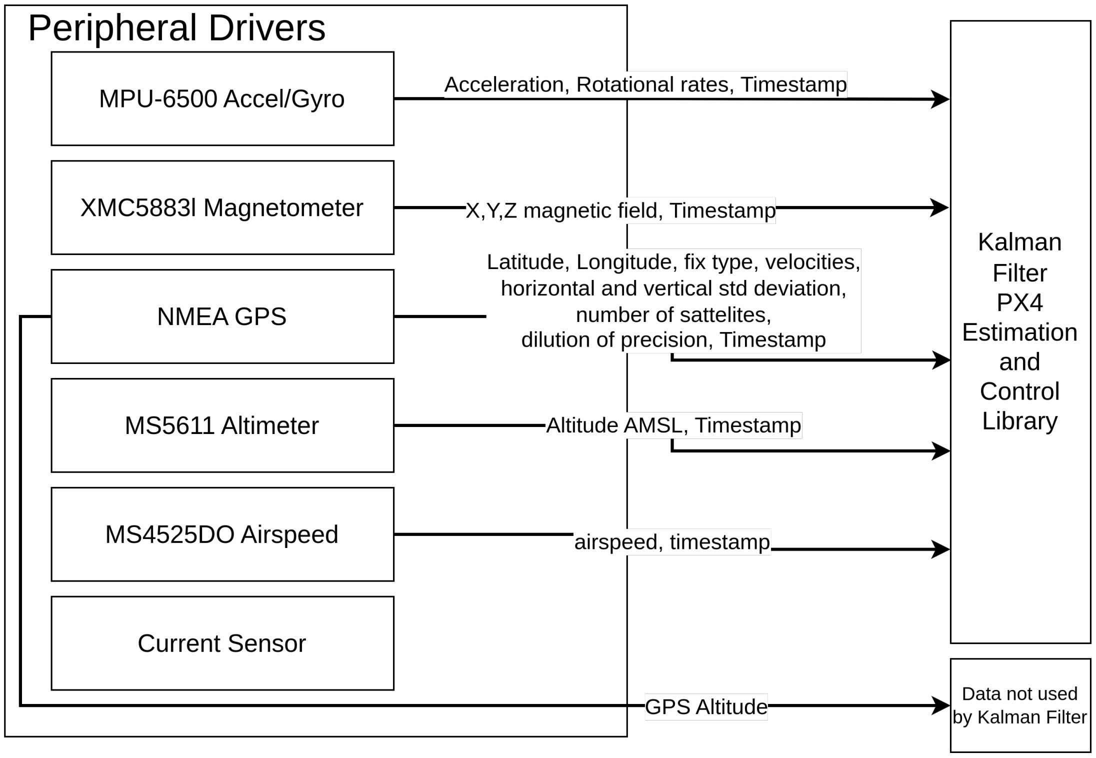

Peripheral Drivers
MPU-6500 Accel/Gyro
XMC5883l Magnetometer
NMEA GPS
MS5611 Altimeter
MS4525DO Airspeed
Current Sensor
Acceleration, Rotational rates, Timestamp
X,Y,Z magnetic field, Timestamp
Latitude, Longitude, fix type, velocities, horizontal and vertical std deviation, number of sattelites, dilution of precision, Timestamp
Altitude AMSL, Timestamp
airspeed, timestamp
GPS Altitude
Kalman Filter PX4 Estimation and Control Library
Data not used by Kalman Filter

FIGURE 5.2: Zoomed In View of the FCS Peripheral Drivers

Next, a zoomed in view of the Kalman Filter is shown in Figure 5.3. The Kalman Filter takes the data from the sensors to compute its outputs of the aircraft's position, velocity, acceleration, orientation, and airspeed, and the current wind speed. These values are part of the aircraft state estimate. The results are fed to the Aries FCS API, which is visible on the GCS and accessible by the Aries autopilot control algorithm.

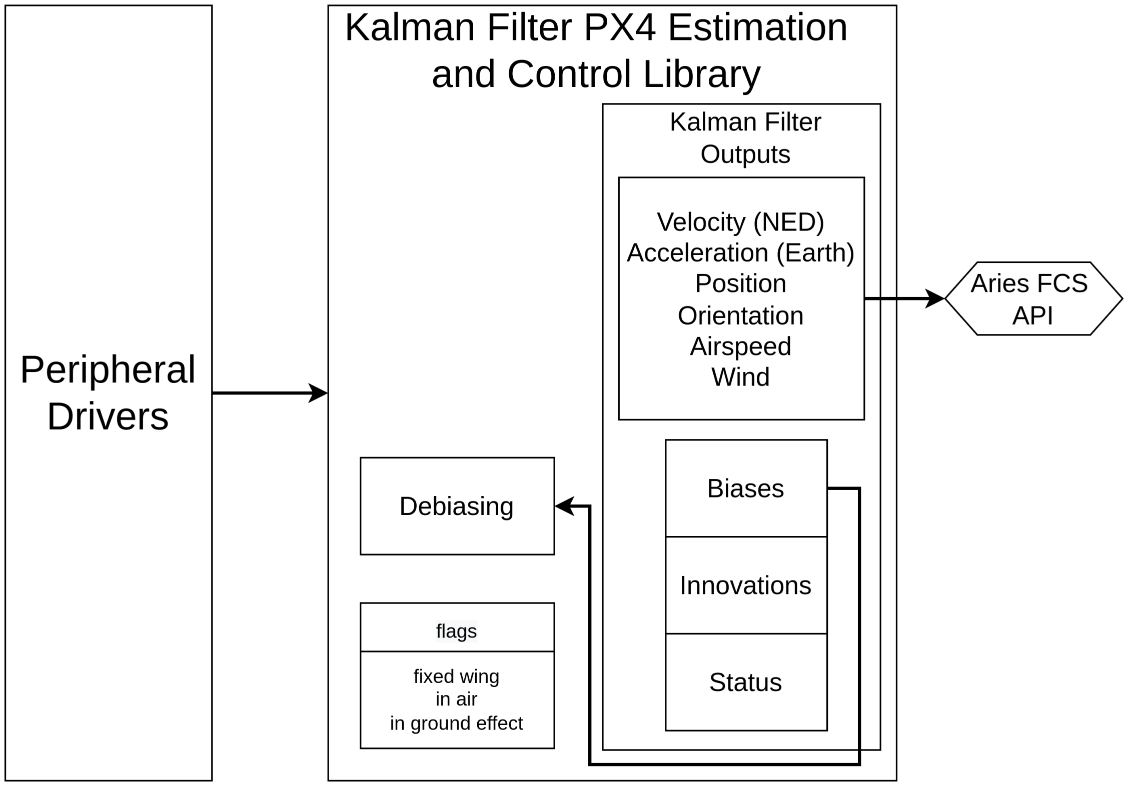


FIGURE 5.3: Zoomed In View of the FCS Kalman Filter

These diagrams were created to verify exactly how each value is used in the FCS data flow. The diagrams also ensure that no potentially unverified data is used as a point of comparison without an understanding of how it interacts with the autopilot software. The goals of this research are to define the operating bounds of the Aries FCS through Hardware-in-the-loop Simulation based on flight simulation data to analyze what effects the inputs have on the system outputs. It should also be noted that fixed-wing flight simulation has been used for existing data collection and will continue to be used in the development of the functional monitors.

Functional Monitoring Architecture

The high-level functional monitoring architecture in support of the Run-Time Assurance Framework developed is shown below in Figure 5.4. This architecture is integrated into the FCS based on the implementation information previously described in Chapter 5. In the figure, the functional monitors receive the actuator outputs from the autopilot, the raw sensor data (as displayed by the sensors/peripherals block of the FCS high-level diagram of Figure 5.1), the mission requirements, and all data the FCS typically sends to the GCS. The functional monitor's outputs consist of sensor or value confidence information or flags relating to the detection of an error on a particular sensor. These outputs can either be sent to the user via the GCS or to the Context Aware Monitor described in [109]. The primary focus of the functional monitors described in this research are intended to monitor and detect problems with the sensor data, which can be caused by cyber-attacks in the form of data corruption, malicious firmware attacks, data spoofing, or by sensor failure or sensor noise.

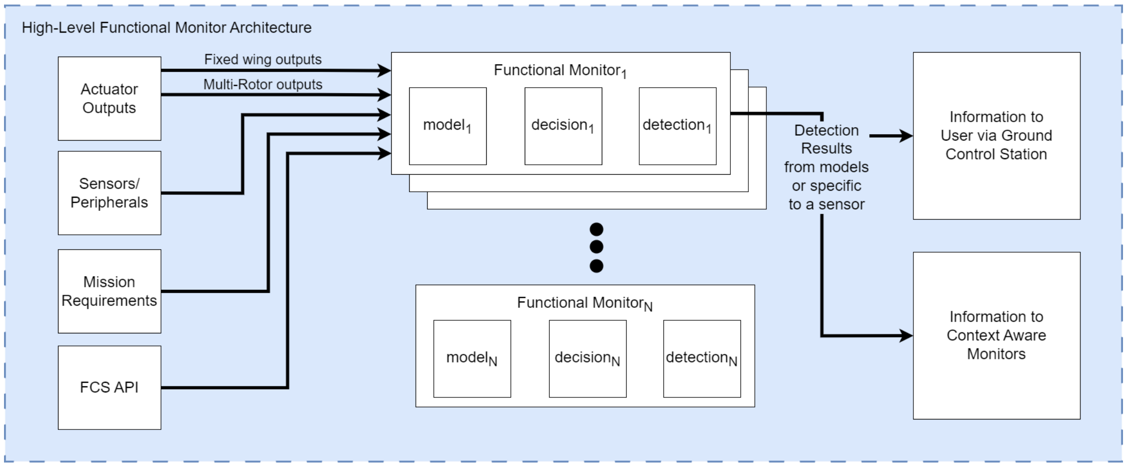


FIGURE 5.4: High-Level Block Diagram of the Functional Monitor Architecture

In addition to the detection of sensor attacks, the construction of the architecture allows for functional monitoring at a higher system level. This capability would include the analysis of waypoint message errors from man-in-the-middle attacks, such as a flight path being changed maliciously, or denial-of-service attacks, where the FCS could be flooded with messages to limit critical communication between the FCS and GCS. The implementation of monitors for the higher system level function, such as checking against mission requirements, is out of the scope of the research described herein.

As shown in Figure 5.4, the functional monitor has 3 main components: the model, decision, and detection blocks.

- Model Block — this component is where the empirically based models, the construction of which are described in Chapter 5.1, live within the functional monitor
  - Input: any of the required inputs shown to the left of the figure depending on the purpose of the model which can include the actuator outputs, sensor/peripheral data, mission requirements, and data from the FCS
  - Output: the post-model calculation based on the inputs
  - Interaction: model output is sent directly to the decision block
- Decision Block — this component implements the various metrics tested, discussed in Chapter 5.2, that provide information on the comparison of data
  - Input: model output, any of the inputs on the left side of the diagram as needed for the necessary comparisons
  - Output: the selected values from any of the metrics tested
  - Interaction: receives data from the model, sends its calculation to the detection block
- Detection Block — this component uses the results from the decision metrics along with the bounds/threshold to provide detection information of a particular sensor if it detects a variation in that sensor's data if the value goes outside the threshold
  - Input: decision block output, the respective boundary or threshold depending on which metric the data from the decision block was calculated with
  - Output: a detection result of a model or specific sensor

  - Interaction: receives data from the decision block and compares with the bounds/threshold, as discussed in Chapter 5.3, to provide vital information to the end user

The operating bounds and thresholds are calculated from the results of the decision metrics collected during a normal flight simulation when no simulated attacks were conducted. This method ensures the values calculated are independent of the type of simulated attack conducted.

5.1 System Models used in the Functional Monitor

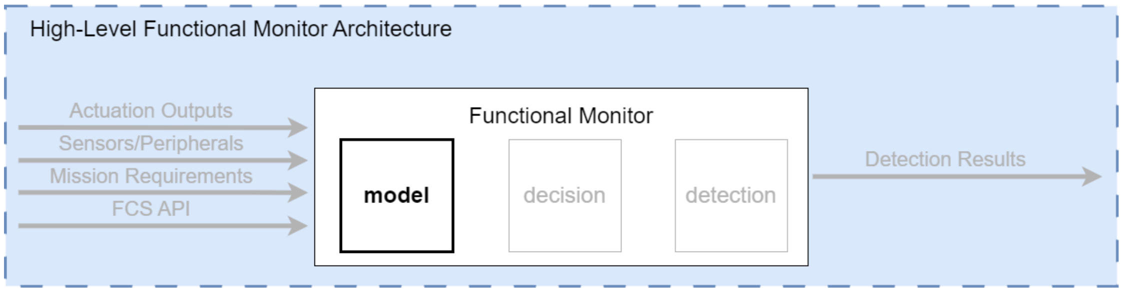


FIGURE 5.5: Simplified Functional Monitor Architecture with Model Emphasized

The first component of the real-time functional monitor is the model, as emphasized in the simplified block diagram of the Functional Monitor Architecture in Figure 5.5. The models are used as a standard for the normal system functionality, against which the actual system function can be compared during runtime. For this research, the models are derived from empirical data collected during flight simulations of the VCU Aries FCS. The goal of the system is to be usable in an application where the developer likely does not have domain knowledge of aircraft dynamics or the ability to collect data from real flights. In an actual application, the data would likely be collected, or verified, using actual aircraft data from flight testing. However, that process of data collection is out of the scope of this research.

The limitation of the empirical modeling approach utilized is that the data is representative of the FCS used in within this research. The purpose of the introduction of empirical modeling is to provide the general framework to use when developing models for the FCS of an UAS. The success of the empirical process tested is based upon the accuracy of the resulting detections achieved by the functional monitor when these models are implemented.

The empirical modeling approach is different from theoretical and data-driven modeling. Theoretical modeling focuses on what type of relationships exist whereas empirical modeling described the specifics of those relationships that are not based on formal logic. Data-driven modeling is commonly used when referring to machine learning when a set of observations are used to develop a characterization of a system. Data-driven modeling is unlike empirical modeling which is based upon a more scientific process for model creation from state observation [114].

The creation, usability, and testing of the functional monitor models is described below. The results observed are unique to the FCS under test, but the process for collecting the data and converting it into the model is not specific to the Aries FCS platform and is intended to inform the general model development process for other, similar monitoring applications.

In order to develop detections for the sensors in use, three different models were identified to encompass a majority of the system operation. These models include an energy model, heading model, and turn radius model. First, the energy model is built upon an expansion of the effect the system input of throttle has on the output energy of the system, specifically the airspeed and vertical velocity. Second, the heading model is based upon determining the heading from the magnetometer for later comparison with the GPS. Lastly, the turn radius model is a computed indication of if the aircraft is or is not in a turn.

Each empirical relationship represents a single model. Multiple model blocks can be combined in a single Functional Monitor as needed. The relationships that can be obtained from flight data can be used to detect faults or cyber-attacks on a few of the main sensors, including airspeed, barometer, GPS, and IMU. The design of the functional monitor within this research is not based upon any common operating procedures or limits, but those could be utilized to support the detection process. Based on the design of the models, the common operating limits such as the never exceed speed or maneuvering speed could be used as reference points in providing detections.

5.1.1 Development of the Energy Model

Proof of Viability of the Energy Model

The development of an energy model for the aircraft was conducted by collecting in-flight data of normal aircraft flight in the simulator. Initially, data was collected with the throttle fixed at 100.0%, for repeatability, and the pitch of the aircraft changed to determine the relationship between airspeed and the change in altitude (vertical velocity). The pitch values are not included in this relationship and are only used to collect the necessary flight characteristics of the aircraft in order to develop a model independent of pitch. The data obtained from this test was graphed as shown in Figure 5.6 and a line of best fit applied, as represented by the standard form 2nd degree polynomial in Equation 5.1, where x is the airspeed and y is the vertical velocity. The corresponding coefficients are displayed in Table 5.1.

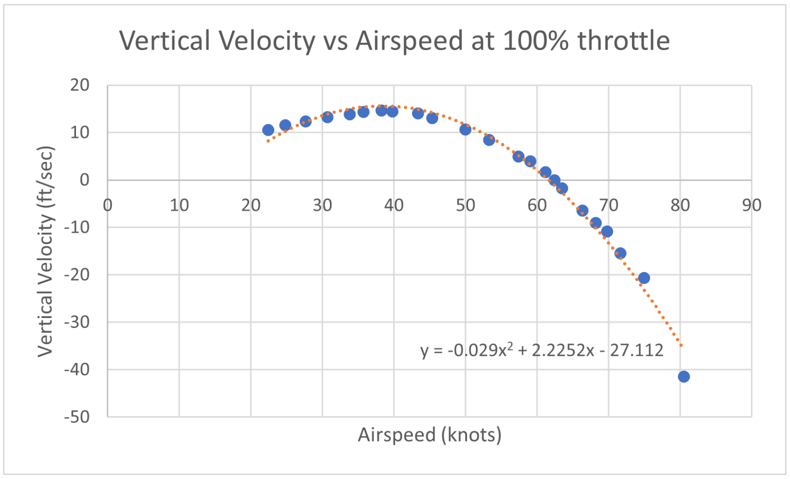


FIGURE 5.6: Graph of Vertical Velocity versus Airspeed Taken at 100.0% Throttle

$$y = -0.029x^2 + 2.2252x - 27.112 \tag{5.1}$$

C_2	C_1	C_0
-2.90E-2	2.23	-2.71E1

TABLE 5.1: Coefficients for Equation 5.1

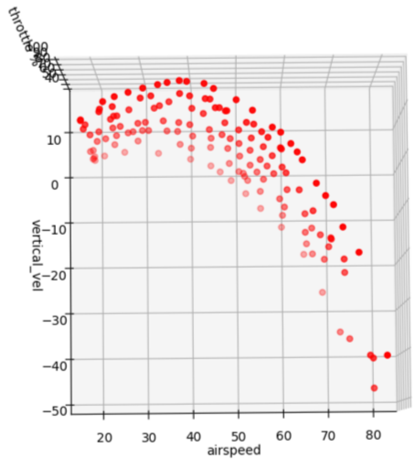
Using a fixed throttle value of 100.0% for this data collection of the input energy to output observations allowed for development of a "proof of concept." The relationship is defined as a representation of how certain outputs are affected by the input, and how they relate to each other by mathematical equations, variations in specific data during a simulated airspeed attack are exhibited.

Complete Energy Model Development

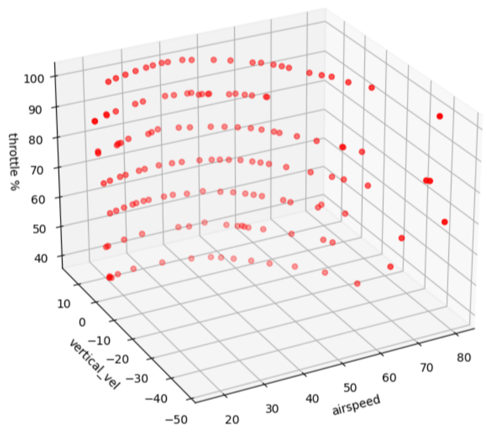
The relationship between the input throttle (i.e., energy into the system) and specified outputs of airspeed and vertical velocity (energy out of the system) was used in the development of a more encompassing model of system function. The data that was initially collected for the 100.0% throttle flight was expanded to display the relationship between airspeed and vertical velocity at a wider range of throttle inputs. The data collection is used to create empirical models of the throttle, airspeed, vertical velocity relationship to narrow down the location of a cyber-attack and hopefully accurately detect an attack. The relationships are based on the idea of energy into the system should be equal to energy out of the system. For example, if there is no energy input (i.e., the throttle is idle), but there is energy output from the system (i.e., the aircraft is gaining altitude), this clearly indicates that there is an error in the data that is reporting system functionality.

The first steps in determining what the previously mentioned model would look like were to collect the remaining fixed-wing flight simulation data. The goal of the energy model is to look at the relationship between airspeed and change in altitude (i.e., vertical velocity) and how those are affected by varying throttle ranges. The data was collected during stabilized flight by fixing the throttle at a set value and adjusting the pitch of the aircraft in order to create vertical velocity to see the effects this change has on the airspeed. In other words, at the same throttle value, if some motor thrust is used to generate a vertical velocity, less can be used to create airspeed. These tests were conducted as similarly as possible for every 10.0% of throttle to collect the remaining data for determining the correlation between input and output energy until the aircraft can no longer maintain altitude from the given throttle input.

This data was graphed as a 3D plot in the graphs of Figure 5.7. To better illustrate the relationship, the 3D plot was initially graphed as individual data points, with a top-down view showing vertical velocity versus airspeed only in Figure 5.7a and a side view with the z-axis of throttle in Figure 5.7b. This data was collected over multiple simulation runs with the aircraft in stabilized mode.



(A) Vertical Velocity vs Airspeed Discretized by Throttle % (top view)



(B) Vertical Velocity vs Airspeed Discretized by Throttle % (side view)

FIGURE 5.7: 3D Graphs of Vertical Velocity vs. Airspeed vs. Throttle

To process the data and generate the best fit equation, the Python Curve Fit function from the SciPy Optimize library was used [115]. The Curve Fit function first takes in the name of a function call that contains the general form of the equation with each corresponding coefficient desired, an array of the input data, and an array of the collected output data. In this situation, the input data are the throttle percent (x) and airspeed (y) values and the output, or expected results, is the vertical velocity (z) data. This particular Python library uses a non-linear least squares regression in producing its outputs.

The least squares regression algorithm is a statistical method used for determining a line of best fit of an unknown dependent variable from a set of known independent variables. The goal is to minimize the sum of the squared differences between the result of the best fit function at each data point and the actual result from the data, then to predict the dependent variable behavior when a new set of inputs that may not have been a part of the fit data is seen. The non-linear least squares variant is an extension of the least squares regression method for use with a larger data set and when the function to be fit is a polynomial of degree of two or greater [116, 117].

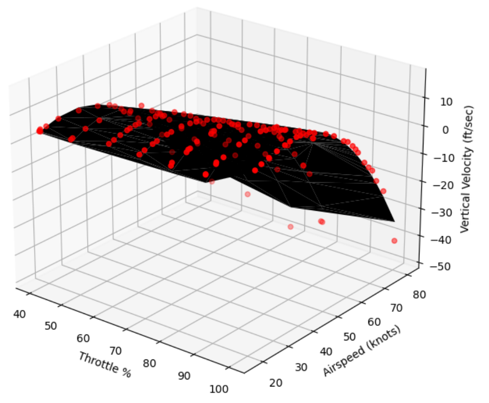
The coefficients produced from the Python Curve Fit non-linear least squares regression as previously described were written into an equation format as shown in its standard form in Equation 5.2 and the coefficients in Table 5.2. In the equation, x is the throttle percentage, y is the airspeed directly from the airspeed sensor, and z is the resulting calculated vertical velocity from this model. The x and y inputs from the data used to generate the best fit equation were fed back in to see how close the calculated vertical velocity is to the original data. The original points, as red points, along with the calculated values, graphed as a 3D surface in black, are shown in Figure 5.8.

$$\begin{aligned} z = C_{22}x^2y^2 + C_{21}x^2y + C_{20}x^2 \\ + C_{12}xy^2 + C_{11}xy + C_{10}x \\ + C_{02}y^2 + C_{01}y + C_{00} \end{aligned} \tag{5.2}$$

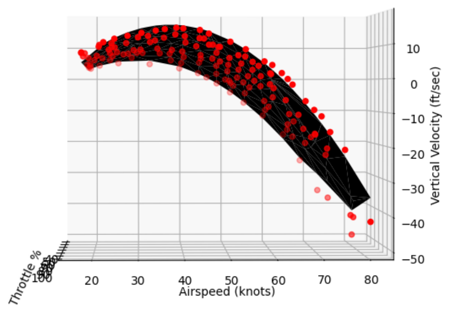
C_{xy}	C_{*2}	C_{*1}	C_{*0}
C_{2*}	3.00E-6	-2.08E-4	3.47E-3
C_{1*}	-4.56E-4	4.07E-2	-6.12E-1
C_{0*}	-6.73E-3	-4.60E-2	6.37

TABLE 5.2: Coefficients for Equation 5.2

After initial testing of the data produced from Equation 5.2 of the model collected during stabilized flight, it was determined that the results contained a continually varying error from the expected value when the aircraft was flying under autopilot control. This was found to be the result of the autopilot continually corrected the pitch and throttle of the aircraft for level flight, thus affecting its airspeed. These throttle and airspeed values applied into the stabilized energy model equation resulted in



(A) Corner View of All Axes



(B) Front View with Throttle Into The Page

FIGURE 5.8: 3D Graphs of the Throttle, Airspeed, Vertical Velocity Relationship from Stabilized Flight

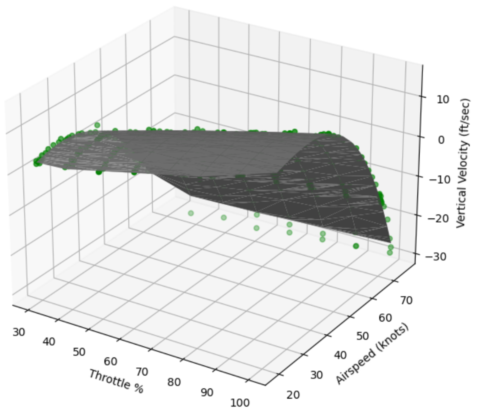
continuously changing vertical velocity values that did not match the expected behavior of the aircraft.

A similar process was used to recollect the aircraft's relevant airspeed and vertical velocity values associated with specific throttle percentages by setting the target airspeed and maximum climb rate over the maximum achievable so that the autopilot would always request full throttle. The maximum throttle percent was adjusted for each 10.0% stage (since autopilot was engaged, 30.0% throttle could be included in this test run) and the aircraft was commanded to climb and descend while the maximum pitch angle allowed was limited to get the same type of data from autopilot flight as the stabilized flight. The resulting 3D graphs for the autopilot collected data are shown below in Figure 5.9 with the collected data points in green and the surface plotted from the equation in grey. The surface comes from the best fit equation expanded from the coefficients in table 5.3 based on the standard form of Equation 5.3. Just like the equation generated from the stabilized flight data, x is the throttle percentage and y is the airspeed directly from the airspeed sensor, and z is the resulting calculated vertical velocity from this model.

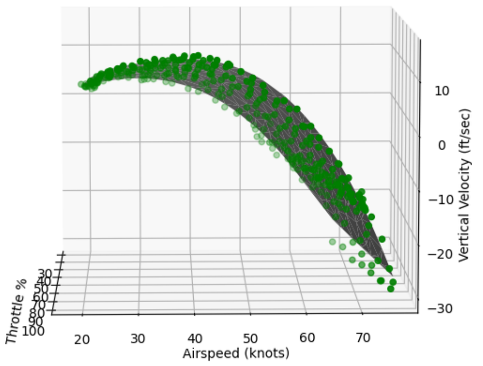
$$
\begin{aligned}
z = C_{23}x^2y^3 + C_{22}x^2y^2 + C_{21}x^2y + C_{20}x^2 \\
C_{13}xy^3 + C_{12}xy^2 + C_{11}xy + C_{10}x \\
C_{03}y^3 + C_{02}y^2 + C_{01}y + C_{00}
\end{aligned}
\tag{5.3}
$$

C_{xy}	C_{*3}	C_{*2}	C_{*1}	C_{*0}
C_{2*}	0.00	1.82E-6	-1.59E-4	2.85E-3
C_{1*}	-3.37E-6	2.02E-4	5.96E-3	-1.00E-1
C_{0*}	-3.62E-5	-1.07E-2	5.03E-1	-5.66

TABLE 5.3: Coefficients for Equation 5.3



(A) Corner View of All Axes



(B) Front View with Throttle Into The Page

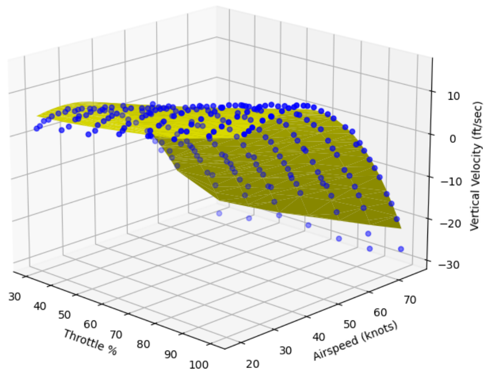
FIGURE 5.9: 3D Graphs of the Throttle, Airspeed, Vertical Velocity Relationship From Autopilot Flight

Another value provided by the curve fit function is the covariance of the computed parameters from the best fit algorithm. Initially the covariance was in the range of $1.0 * 10^{21}$ indicating a significant amount of over-fitting. Many iterations of the output equation format and corresponding graph (to ensure the equation maintained closeness to the input data) as well as the corresponding covariance values from each equation were examined. The input data was also reviewed to determine if any of the values provided any indication of the cause of the over-fitting. It was found that due to the method in which the data was collected, there were multiple test runs that provided similar data for the same vertical velocities and airspeed values. After the data was improved to remove values with significant similarities and the curve fit rerun, the covariance was reduced to the range of $7.0 * 10^8$. The final and simplest equation that still maintained closeness with the original data and had reduced covariance is written below in standard form by Equation 5.4 with the coefficients in Table 5.4. The improved result was then graphed in the same manner as the previous equations and is seen below in Figure 5.10, with the data points in blue and the surface of the best fit equation in yellow. To illustrate the accuracy of the calculations of Equation 5.4 versus the original observations, the average error across the surface is -0.08 ft/sec.

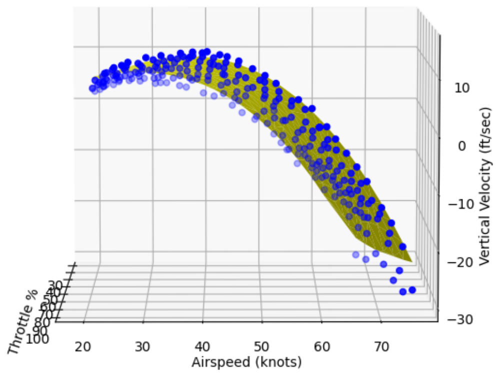
$$z = C_{03}y^3 + C_{11}xy + C_{01}y + C_{10} \tag{5.4}$$

C_{xy}	C_{*3}	C_{*2}	C_{*1}	C_{*0}
C_{2*}	—	—	—	—
C_{1*}	—	—	4.94E-3	-4.49E-2
C_{0*}	-1.68E-4	—	2.28E-1	—

TABLE 5.4: Coefficients for Equation 5.4



(A) Corner View of All Axes



(B) Front View with Throttle Into The Page

FIGURE 5.10: 3D Graphs of the Throttle, Airspeed, Vertical Velocity Relationship From Autopilot Flight — Corrected Model

To demonstrate the accuracy and provide a visual representation of why the energy model required a few variations, each of the calculated z values (vertical velocity) from the three equations previously discussed are shown below in Figure 5.11. In the graph, the black line in the background maintaining close to 0.0 is the actual vertical velocity reported by the FCS during a short normal flight. Based upon the airspeed and throttle percent during this flight, each of the energy models were used to calculate a resulting vertical velocity. The green line shows the results from Equation 5.2, the red line shows the results from Equation 5.3 after the airspeed input was increased to the 3rd order, and lastly the blue line represents the output of Equation 5.4 after the over-fitting was reduced.

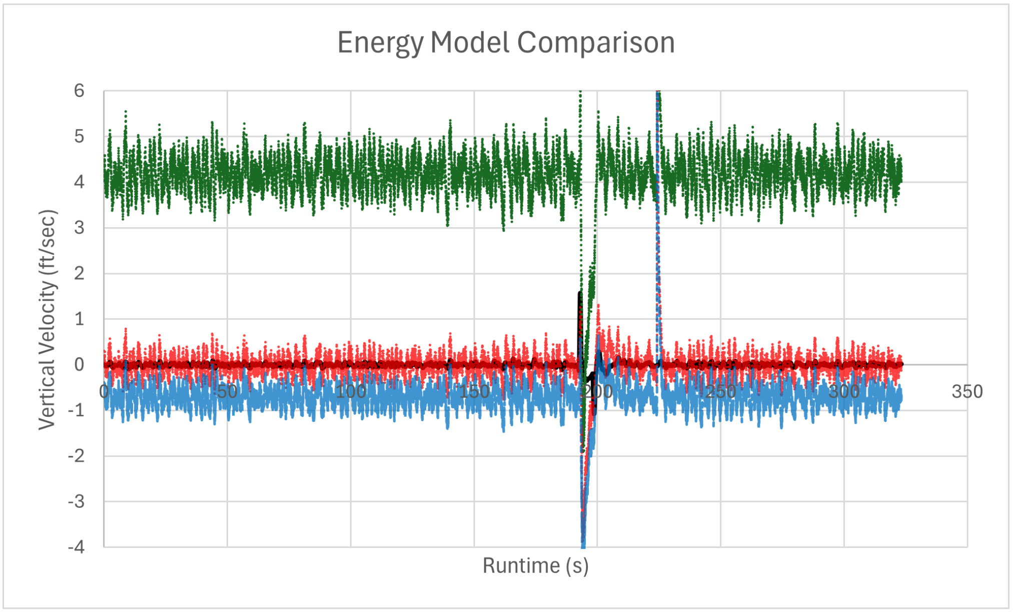


FIGURE 5.11: Comparison of the Calculated versus Actual Vertical Velocity

The ability of the energy model to indicate a difference in indicated versus actual vertical velocity during an attack was also verified. The graph showing this test is seen below in Figure 5.12. Similar to the previous graph, the black line represents the indicated vertical velocity from the FCS, the blue line indicates the output from the energy model, and the orange dashed line represents when a simulated altitude attack was initiated. This graph proves the energy model is effective in providing useful information that the dynamics of the aircraft based upon the input energy, throttle, and airspeed of the aircraft. During this simulated attack, the autopilot does not indicate any change in altitude whereas the energy model does indicate a change in altitude which is accurate of the aircraft's physical movements.

Other curve fits were run to further reduce the covariance, but these resulted in the model having too much error from the original data. This error rendered it unusable for

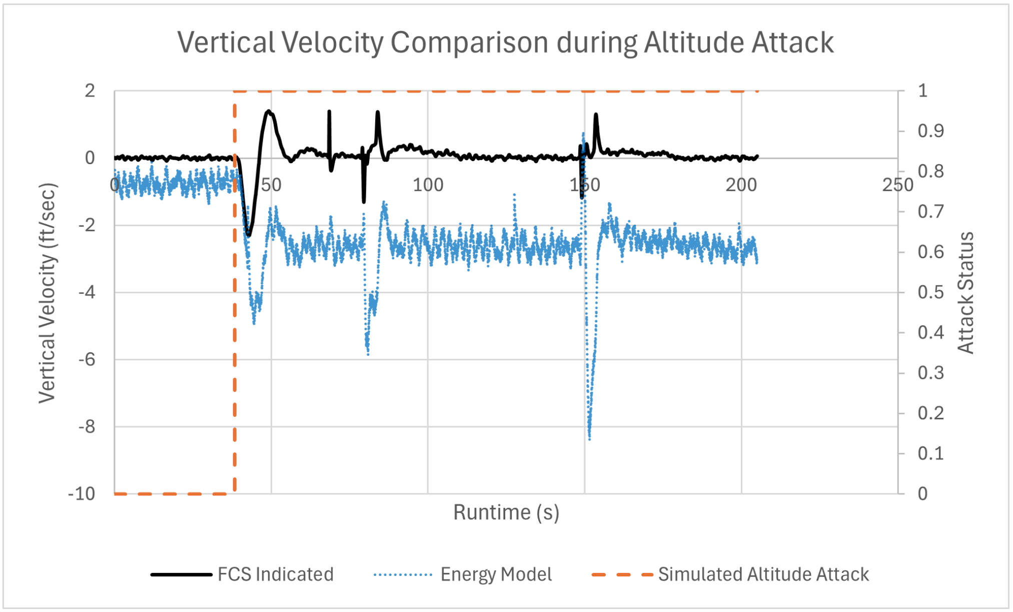


FIGURE 5.12: Comparison of the Calculated versus Actual Vertical Velocity During an Altitude Attack

the purposes of detecting variations in the aircraft data during runtime. The resulting cause of the over-fit was determined to be a result of the nature of the input data being specific to the aircraft in which the model is designed for. The focus of the work lies in developing a method to achieve a model specific to whatever aircraft is in use, not for making a one size fits all equation. This uniqueness of data can result in the possibility of using an over-fit data set.

The first best fit equation from the autopilot data only went to the second order for both the airspeed and throttle input. The vertical velocity calculated from that equation was much improved for all flight types over the data recorded during the stabilized flight. One issue that still remained was a small inherent error that maintained a constant delta when the aircraft was flying straight and level, which didn't track the ascents and descents correctly. The best fit was rerun with the airspeed going to the third degree, as seen in Equation 5.3, and the calculated vertical velocity was perfectly locked in to the expected values.

The need for increasing the order of airspeed was determined from an analysis of the typical aircraft physics equations. Power is equivalent to force times velocity, where velocity is the airspeed of the aircraft, as shown in Equation 5.5. The power in this example is the total force F_{total} the aircraft experiences. The total force is the combination of the aerodynamic drag power P_{drag} and the power of the change in gravitational potential energy $P_{\Delta PE}$ displayed in Equation 5.6.

$$P = F * V \tag{5.5}$$

$$P_{total} = P_{drag} + P_{\Delta PE} \tag{5.6}$$

Broken down into its individual components, Equation 5.7 shows that P_{drag} is the drag force F_D times velocity. The change in gravitational potential energy power is the derivative with respect to height of the standard potential energy equation of *mgh*, as shown on the left-hand side of Equation 5.8. In this equation, m is the mass of the aircraft, g is the acceleration due to gravity, and h is the altitude. The resulting derivative is shown on the right-hand side of Equation 5.8, where $V_{vertical}$ is the vertical velocity.

$$P_{total} = (F_D * v) + (m * g * V_{vertical}) \tag{5.7}$$

$$\frac{d}{dh}(m * g * h) \Leftrightarrow m * g * V_{vertical} \tag{5.8}$$

Next, the standard aerodynamic drag force equation is defined below in Equation 5.9. In this drag force equation, C_d is the drag coefficient, ρ is the air density, A is the cross-sectional area of the aircraft, and v is the airspeed. As a result, it is indicated that aerodynamic drag is proportional to the square of velocity.

$$F_D = \frac{1}{2} * C_d * \rho * A * v^2 \tag{5.9}$$

Equation 5.7 is solved for $V_{vertical}$ to obtain the relationship in Equation 5.10. The solved equation indicates that vertical velocity is a function of the drag force times velocity, where drag force is a function of velocity squared. This relationship proves the correlation between vertical velocity to airspeed cubed is proportional. Since airspeed is the velocity component in equations described, Equation 5.10 is expanded with the substitution of the drag force equation, shown by Equation 5.11.

$$V_{vertical} = \frac{P_{total} - (F_D * v)}{m * g} \tag{5.10}$$

$$V_{vertical} = \frac{P_{total} - (\frac{1}{2} * C_d * \rho * A * v^3)}{m * g} \tag{5.11}$$

To illustrate these variations, the airspeed and vertical velocity relationship, defined by Equation 5.1, details the energy model of the functional monitoring architecture in Figure 5.4. The input for this specific model is airspeed in knots and the output is the calculated vertical velocity based on the equation.

The idea of the energy model came about from the plan of attempting to detect attacks and data faults independent of the GPS sensor. Through the development of a model based upon the aircraft's energy state, airspeed and altitude errors can both be identified, thus fulfilling the objective of error detection in the absence of GPS data. One caveat is that in order to determine which of the two sensors, altitude or airspeed, is the cause of the error, the GPS data must be used, with the assumption that it is accurate, to provide a point of comparison for the altitude and airspeed data.

5.1.2 Development of the Heading Model

When determining the scope of the empirical model for analysis of the GPS data for any variations, taking into account what data the GPS sensor typically provides is vital. Specifically, this data includes latitude, longitude, altitude, heading, and ground speed of the aircraft. A portion of the magnetometer (i.e., compass) data is used to calculate the model for analyzing GPS data walk-offs. The calculation used for this model first required using the magnetic declination table to calculate the specific declination value at the current latitude and longitude of the aircraft. Then the heading based upon the magnetometer sensor (commonly abbreviated as mag) was computed based on Equation 5.12 below. The arc tan of the Y and X mag values was calculated, multiplied by -1 to obtain the correct orientation, and converted from radians to degrees. The computed heading based on the mag data is added to the magnetic declination to obtain a calculated heading value.

$$Computed\ Heading_{mag} = -1 * atan2(Y_{mag}, X_{mag}) * \frac{180.0}{\pi} \tag{5.12}$$

Flight data was collected, as was conducted with the energy model, to show the effectiveness of the heading model calculation during a normal flight and one with a GPS walk-off attack being conducted. First, in Figure 5.13, the heading reported by the GPS is shown by the black line and the heading calculated from the above model based on the mag data is shown in blue. As seen here, the model agrees with the actual heading when no attack is being conducted. Figure 5.14 shows the same comparison of data except when a GPS walk-off attack was initiated as shown by the orange dashed line. The graph shows an obvious disagreement between the two heading values, indicating that this model will also be useful in providing data for attack detection analysis.

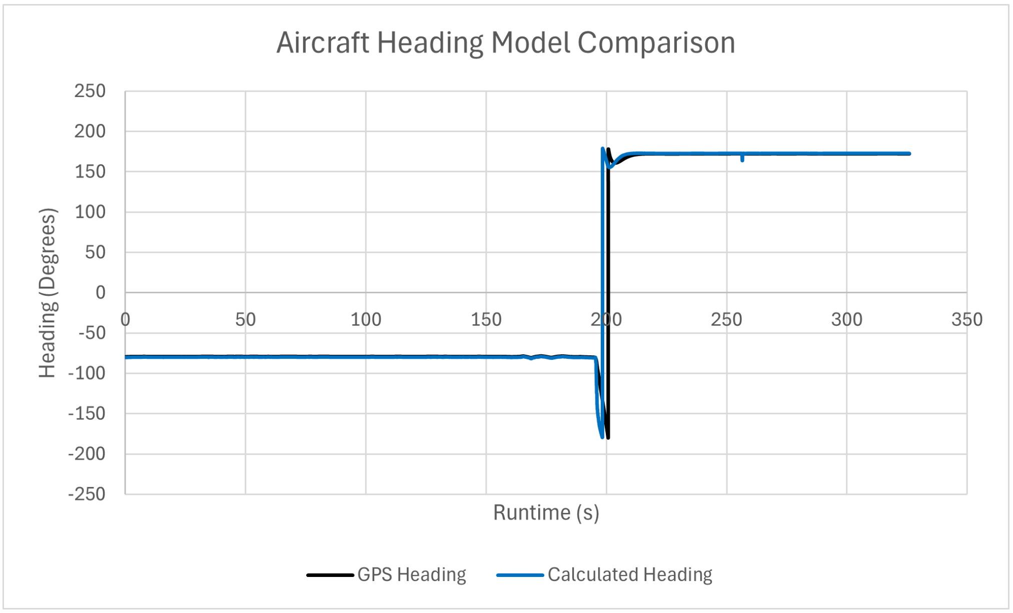


FIGURE 5.13: Comparison of the Calculated versus Actual Heading

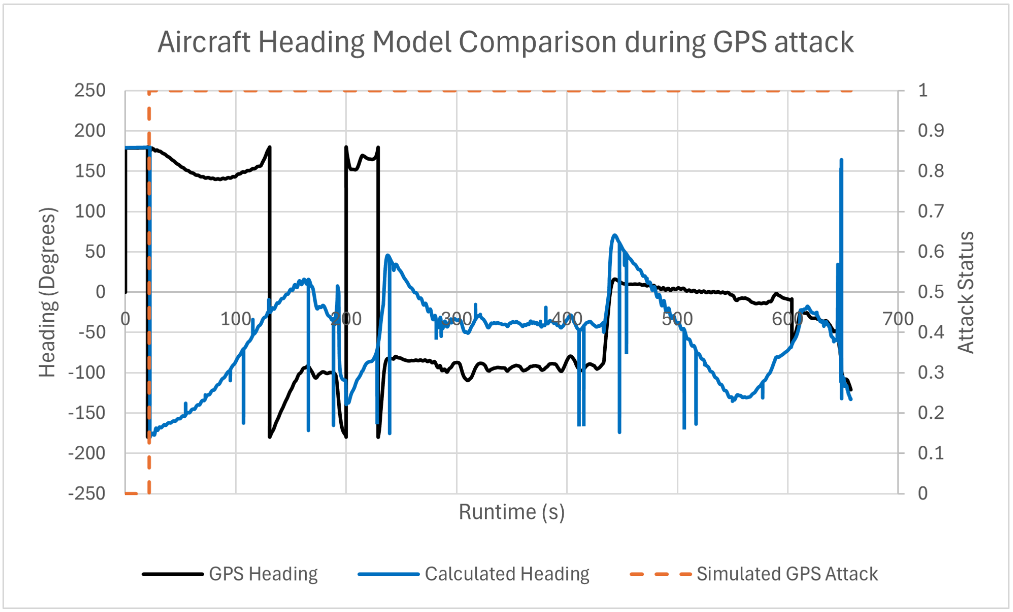


FIGURE 5.14: Comparison of the Calculated versus Actual Heading During a GPS Walk-Off

5.1.3 Development of the Turn Radius Model

To review the two models so far, the energy model considers altitude and airspeed, and the heading model applies to the GPS data. When development began on detecting a walk-off of the IMU sensor, it was found that the pitch walk-off affects the altitude, thus being detectable by the energy model, and the yaw walk-off affects heading, being detectable with the heading model. This created the need for another model, as neither of the existing models could detect the roll walk-off of the IMU. The typical physics of an aircraft in terms of roll is that when no other changes occur to the aircraft's control surfaces, a roll in either direction will cause the aircraft to turn, resulting in a change in the heading.

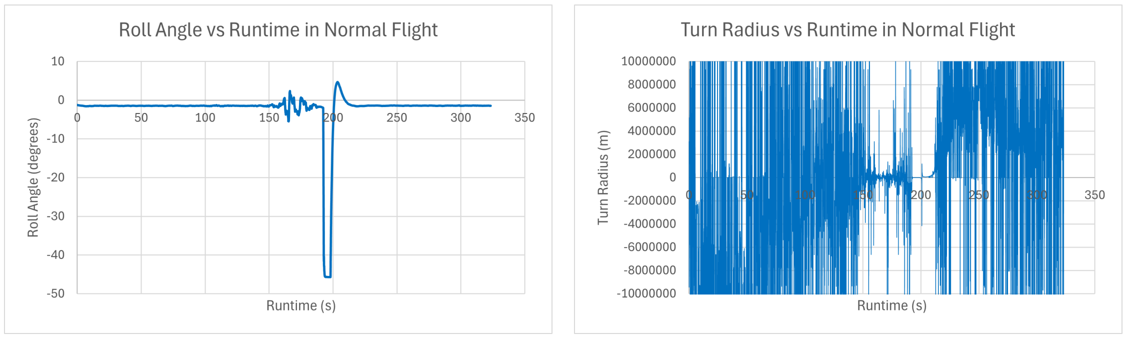
The main goal with this new model was to determine if the data provided by the sensors and FCS indicate a turn and if the indicated roll from the FCS corresponds to the change in aircraft heading. To develop this empirical model, the idea of calculating the turning radius of the aircraft, as presented in [118] was used. The equation used to calculate the turning radius of the aircraft is described below by Equation 5.13. The numerator of the equation is the airspeed in meters per second from the airspeed sensor as the velocity of the aircraft. This value is then divided by the rate of change of the aircraft's yaw. The yaw value is provided by the FCS and the derivative of which is computed and converted to radians. The resulting value is the turning radius of the aircraft.

$$Turning\ Radius = \frac{Airspeed_{m/s}}{Yaw\ Rate_{rad/s}} \tag{5.13}$$

The turning radius model has two main observable features. The first being if there is minimal change in the aircraft's heading, the denominator will be relatively small. The small change in heading would result in a significantly large radius, indicating a roll angle of close to 0.0. Second, as the yaw rate increases, i.e., the roll angle increases inducing a turn of the aircraft, the radius becomes much smaller.

The graphs of Figure 5.15 illustrate the process of the turning radius calculation from a flight with no attacks running. In the upper left-hand corner, Figure 5.15a shows the roll angle of the aircraft during this test flight as reported by the FCS. As seen here, there is one turn (i.e., change in heading) that occurred. Figure 5.15b shows the corresponding calculated turning radius during this flight based on the rate of change of the yaw provided by the FCS. This graph shows that when the roll is fairly steady, the turning radius is significantly large. The continual changes from large negative to large positive numbers are due to the minor corrections made by the autopilot, resulting in large swings in the calculated turning radius. It is also shown that when the roll increases for the turn, this increases the yaw rate, thus lowering the scale of the turning radius.

This same process and calculation method was used to display the usefulness of the turn radius model during a roll walk-off attack in both the autopilot and stabilized flight modes. The implementation of this testing was completed utilizing the method previously discussed in Chapter 4.4. During this autopilot flight, a roll attack of +10.0° was tested. The corresponding difference between the actual roll angle of the aircraft, the grey line, and the reported roll from the FCS, the blue line, is shown in Figure 5.16a. As is true in all graphs displaying a simulated attack, the orange dashed line indicates



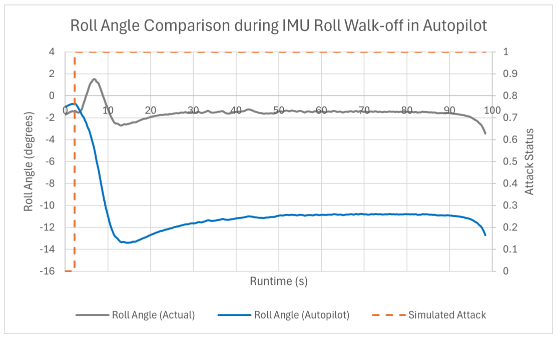
(A) Roll Angle During (B) Calculated Turning Radius

FIGURE 5.15: Turn Radius Model Example During Normal Flight

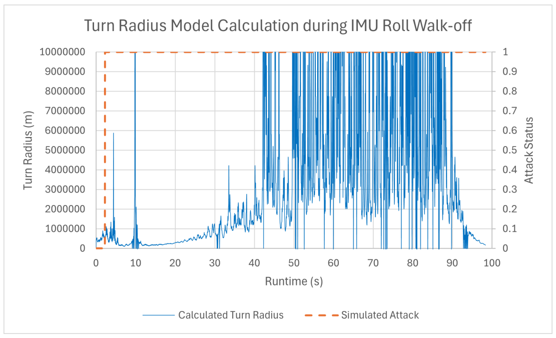
when the attack was initiated.

Over the runtime for this attack, initially due to the nature of the autopilot attempting to re-level the aircraft based on incorrect information, some minor adjustments are made to the aircraft. These adjustments result in large changes in the yaw rate and thus a small turning radius, as indicated early on in Figure 5.16b. As runtime continues and the roll value levels off, the turning radius begins to increase to the scale of indicating that the aircraft is not turning. This phenomenon is accurate based on the way in which the autopilot corrects the IMU walk-off attack in the roll axis. The key difference here to be reviewed is that based on the physical flight characteristics seen in the simulator along with the raw yaw data. The data indicated by the turning radius calculation exhibits no change in the aircraft's heading. The roll angle from the FCS indicates that there should be a change in heading. This difference leads to the method of using this model for detections, which will be discussed in more detail in Chapter 5.4.

The results observed during a stabilized flight when the IMU roll walk-off was active, as shown below in Figure 5.17, exhibit the opposite effect as that observed during the autopilot flight. The comparison of the roll values between the actual and indicated by the FCS are shown in Figure 5.17a. To summarize the effect of this attack during a stabilized flight, as the roll value is walked-off, the main processing of the FCS attempts to re-level the aircraft. The result is the aircraft rolling the opposite direction of the walk-off value. The FCS interprets the final result as a level aircraft, when instead it is



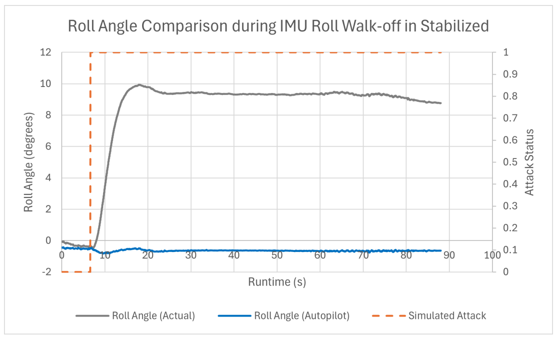
(A) Roll Angle Comparison Between the Actual and Indicated Values



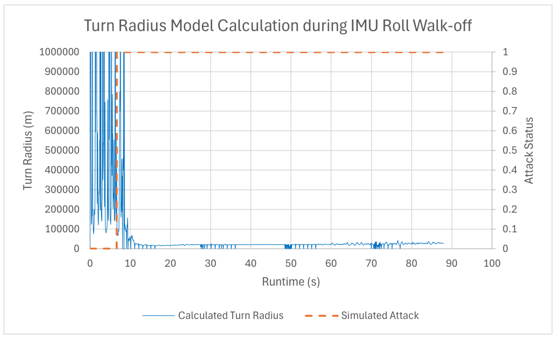
(B) Corresponding Turning Radius Calculated from the FCS Data

FIGURE 5.16: Turn Radius Model Example During IMU Roll Walk-Off in the Autopilot Flight Mode

actually holding at the negative of the roll walk-off value. In flight, this induces a turn of the aircraft. The resulting turn should create a change in yaw rate, thus result in a low turning radius, indicating the aircraft is in fact changing its heading. This phenomenon is shown in the turning radius calculation of Figure 5.17b. Again, the important factors to note here are that the turning radius indicates the aircraft is changing its heading. The assumption from this observation is that the roll must have diverged from 0.0°, but the FCS continues to report a roll of around 0.0° which shows no indication of turning.



(A) Roll Angle Comparison Between the Actual and Indicated Values



(B) Corresponding Turning Radius Calculated from the FCS Data

FIGURE 5.17: Turn Radius Model Example During IMU Roll Walk-Off in the Stabilized Flight Mode

5.1.4 Summary of Model Creation

The three models developed cover a broad range of different observations of the in-flight data. The energy model can be used to identify a variation in any altitude or airspeed based error including from the altitude sensor, airspeed sensor, and IMU pitch data. The heading model can be used for identifying variations in the GPS data and IMU yaw. Of the sensors implemented, this leaves variation detection of the IMU roll, which was accomplished by the turn radius model. The models discussed were initially tested for one simulated walk-off attack at a time.

Each model showed promising results for the specific sensor intended, with the difficulty being that the energy model can show a variation of the airspeed or altitude sensor but is unable to determine which sensor exactly without other information. The idea is to use the models in combination to test the possibility of determining which sensor is likely at fault through the use of how much a particular model results in a positive detection when appropriate. This process will be discussed further in Chapter 5.4.

5.2 Decision Metrics used in the Functional Monitor

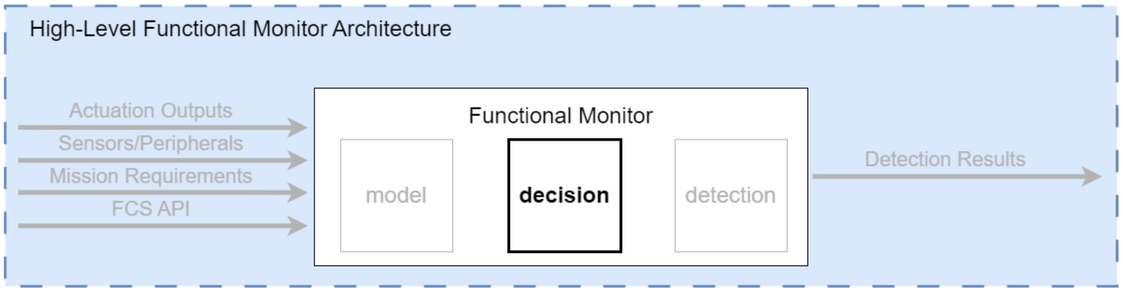


FIGURE 5.18: Simplified Functional Monitor Architecture with Decision Emphasized

The decision block of the functional monitor implements a decision metric based upon a mathematical method for analyzing a difference between input observations. Figure 5.18 emphasizes the location of the decision block in the overall Functional

Monitor Architecture in development. The decision metrics are intended to play a significant role in the detection of variation in the data caused by cyber-attacks and physical sensor faults on the FCS. To determine the efficacy of the models discussed in section 5.1 for cyber-attack detection, an error indicator metric was developed and is discussed in this chapter.

In addition to the error indicator metric, run-time implementations of a few statistical based metrics, extended from the work presented in [119], are evaluated below. Throughout the search for methods used to identify variations, drifting, and errors within sensor data, a few common statistical methods were reviewed. The methods were selected based on their usefulness for providing key information of data variance when limited points of comparison are available. The methods tested include cumulative sum, exponentially weighted moving average, xbar and range, and correlation coefficient. The metric definitions and how they are implemented are described below. All the metrics are implemented such that they are calculated during run-time of the functional monitoring system to provide real-time results to the user. The results can also be used for offline analysis of the effectiveness of each method in providing usable and quantifiable error margins.

Flight simulation data was collected both without an attack, and during the time a simulated attack was initiated, to demonstrate the usefulness of each of the following decision metrics under test. Following each metric description are a series of graphs to illustrate a portion of the types of results each metric provides. For the purposes of displaying such results, the input data to each metric is a variation of altitude-based data and will be explained in more detail for each method. The specific values calculated from a metric are in either blue or grey, depending on if there are one or two calculated values in a single graph. In the graphs that have a simulated attack present, the orange dashed line represents when the simulated attack was started. The simulated attack run for the purposes of this data collection is the altitude walk-off attack as described in Chapter 4.1.

5.2.1 Error Indicator Metric

The error indicator metric is based on the scale of the error, along with what the error can indicate over time. First, the error, E, is calculated as shown in Equation 5.14, where X is the specific sensor value in question, to determine how far away a value is from the expected state. E_M of Equation 5.15 is the magnitude of the error as calculated from Equation 5.14. Next, E_T (as shown in Equation 5.16) is helpful in determining how long a value stays incorrect due to this value having "memory" and accumulating over time. Lastly, E_Δ (as shown in Equation 5.17) provides information on the rate of change and direction based upon the current and previous error values.

$$E = X_{actual} - X_{calculated} \tag{5.14}$$

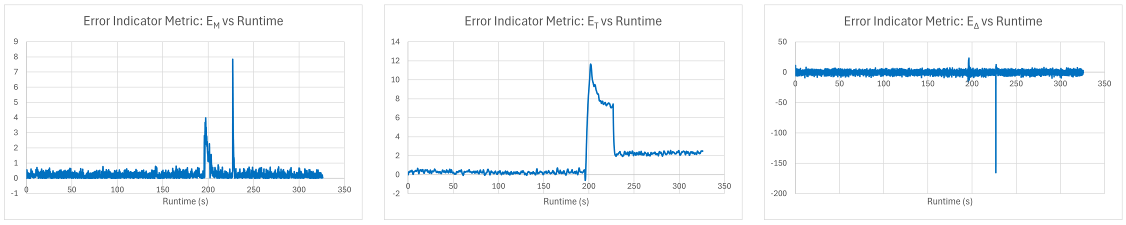
$$E_M = |E| \tag{5.15}$$

$$E_T += \; E * dt \tag{5.16}$$

$$E_\Delta = \frac{E - E_{prev}}{dt} \tag{5.17}$$

The graphs within Figure 5.19 display the results of the error indicator metrics during a normal flight. The data used for the calculations displayed by the graphs consist of the vertical velocity as reported by the FCS as the X_{actual} value and the calculated vertical velocity from the energy model (as discussed in Chapter 5.1.2) as the $X_{calculated}$ value. The left-most graph shows the E_M error (as calculated by Equation 5.15) in Figure 5.19a representing the absolute value of the difference between the actual and calculated value. As seen here, the error remains close to 0.0 as would be expected based on the accuracy of the energy model. As a result, the E_T value in Figure 5.19b (as calculated by Equation 5.16) shows how the error builds up over time, which in this case is minimal. The rightmost graph in Figure 5.19c below displays the E_Δ error (as calculated by Equation 5.17). This value represents the rate of change of the value providing insight into if and how much a value is changing over each time time-step.

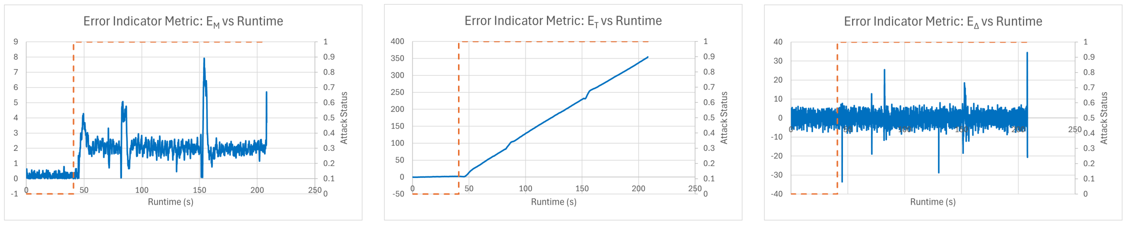
The next set of graphs are aligned in the same order as the graphs of Figure 5.19



(A) E_M (B) E_T (C) E_Δ

FIGURE 5.19: Error Indicator Metrics During a Normal Flight

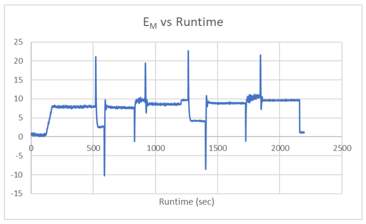
above to provide a side-by-side comparison between each value in normal flight and with a simulated attack running. As seen in Figure 5.20a, the E_M value showing a difference between the actual and calculated values accurately show a difference between these values. This difference is reflected by the increasing E_T graph in Figure 5.20b showing the error is maintained over the run-time. Lastly, Figure 5.20c displays the E_Δ of the metric. Of the values created from the error indicator metrics, the value found to be the most important is the average of the E_Δ. Before the attack was initiated, the average rate of change was 0.00379. During the whole run-time the attack was active, the average value for E_Δ was 0.0209. It was noted that the faster a walk-off attack was applied, the greater the rate of change was. This difference shows how this method could be used during a simulated attack.



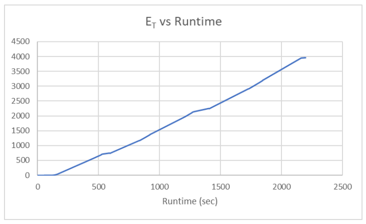
(A) E_M (B) E_T (C) E_Δ

FIGURE 5.20: Error Indicator Metrics During a Simulated Attack

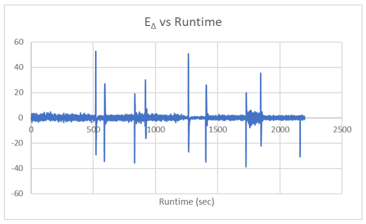
This method isn't without its limitations. The rate of change of the error is only useful when the error has a continuous change, such as a continual data walk-off attack. One of the implemented attacks is based on walking-off the value until a set point is reached and then maintaining that offset. A maintained offset will have a rate of change of around 0.0 which would be reflected in the E_Δ graph, thus having a value that is not indicative of an error. Figure 5.21 displays such a scenario, where the error builds up before leveling off at a predefined difference. This error is seen in Figure 5.21a. The corresponding E_{Δ} in Figure 5.21c accurately displays the described response of no variation in the value, with the average value of -0.00541 when an attack was actively running. The E_T value representing the total change over the whole system run-time did not prove to be useful, as any small error can build up even when no attack is running. The cumulative nature of the E_T value results in an unknown scale of the value when an attack occurs. Thus, the need for the continued research into other methods that are sensitive to data variations that hold a constant value instead of a continual walk-off.



(A) E_M of Vertical Velocities



(B) E_T of Vertical Velocities



(C) E_{Δ} of Vertical Velocities

FIGURE 5.21: Data During Walk-Off and Hold at Constant Δ During Airspeed Attack

5.2.2 Cumulative Sum

Cumulative sum (CUSUM) is a statistical measure based on aggregating the values of a sequence of past and present data to calculate positive or negative deviations from a predetermined target value [120, 121]. The features of CUSUM allow for detecting both positive and negative out-of-control values of a particular parameter under constant change. CUSUM is the more favorable option for applications where small shifts in the process average must be detected.

The first component of the CUSUM requires a standard deviation, denoted σ, of the specified data in question. In all metrics that utilize standard deviation, this same method was used. It is important to note the method used is only an estimate, not a true standard deviation, as it is calculated during run-time in a single pass, whereas the typical standard deviation requires a two-pass method. The standard deviation function used is shown in Equation 5.18. To calculate σ, each on pass the sample of data, X, is obtained and added into an accumulated variable, $History_{Sum}$. The same sample, X, is squared then accumulated into $History_{Squared\ Sum}$. The total number of samples are also counted. In the standard deviation equation, the accumulated sum is squared and then subtracted by the squared sum from each pass divided by the total samples minus one.

$$\sigma_X = \frac{History_{Squared\ Sum} - (History_{Sum})^2}{Number\ of\ Samples - 1} \tag{5.18}$$

Next, the deviation factor Z, at each iteration n, is calculated using the past and present deviations within the data in question, is not a cumulative value, and is calculated at each pass through the cumulative sum function. As shown in Equation 5.19, the current sample X_n and the current average of all sample values $\overline{X}$ are calculated and subtracted from each other, then divided by the standard deviation of X as calculated in the previous step. The Z_n is used as a way of normalizing the data around the target in-control value, then scaling by the standard deviation. This method of normalization removes the need for incorporating a target value into the set of CUSUM equations. Not including a target value, in this case 0.0, was useful for all use cases of cumulative sum since the input value is already centered around 0.0.

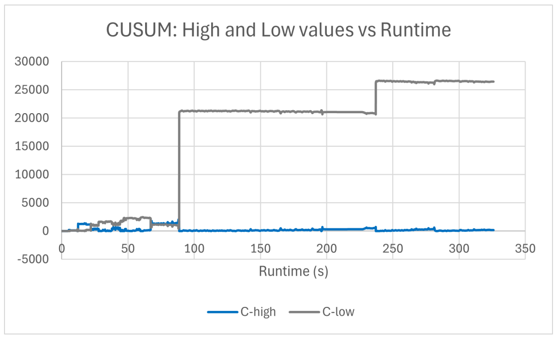
$$Z_n = \frac{X_n - \overline{X}}{\sigma_X} \tag{5.19}$$

Lastly, for the calculation, the high and low CUSUM factors are calculated. First, run-time data was collected of the deviation factor to determine what scale Z_n falls within to determine a range of usable reference values K_{factor}. Below in Equations 5.20 and 5.21 are the formulas for calculating the high and low CUSUM factors, respectively. These factors, as seen in both equations, utilize its respective previous value, $C_{high/low_{n-1}}$, then either add, for C_{high}, or subtract, for C_{low}, the Z_n value calculated from Equation 5.19. Next, the K_{factor} is subtracted from the result of the first operation. The value calculated from the previous addition and subtractions is then compared with 0.0 to determine the max between the two values, which is then stored as the final result to the CUSUM factor value. The reference value can also be considered a sensitivity factor based on the structure of the high and low factor equations. If K is too large, the resulting high and low factors would be 0.0 for the entire run-time, and if K is too small, every minor change in Z would result in a spike in the resulting factors values.

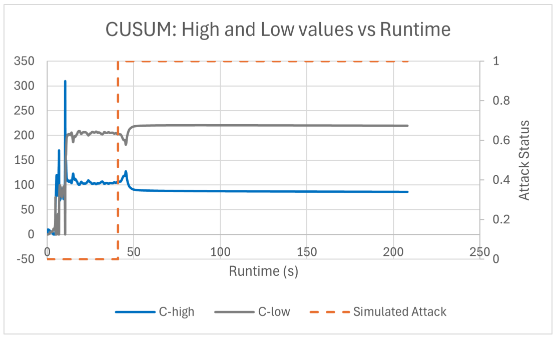
$$C_{high_n} = max[0.0, C_{High_{n-1}} + Z_n - K_{factor}] \tag{5.20}$$

$$C_{low_n} = max[0.0, C_{Low_{n-1}} - Z_n - K_{factor}] \tag{5.21}$$

To demonstrate the type of results the cumulative sum method provides for C_{high_n} and C_{low_n}, the calculated values from a data collection of a normal flight and a flight with the same simulated attack as used for the error indicator metric were tested. Figure 5.22 shows both of these results. Figure 5.22a on the left shows the high and low values based on the same input observations of the vertical velocity, as reported by the FCS, and the calculated vertical velocity, from the energy model. The difference between these two values was the input X_n. At first glance, the usability of the cumulative sum is unclear based on the graph during normal flight. When the simulated attack was run, the high and low values continue to not be conclusive indicators of the error, as seen in Figure 5.22b.



(A) Normal Flight

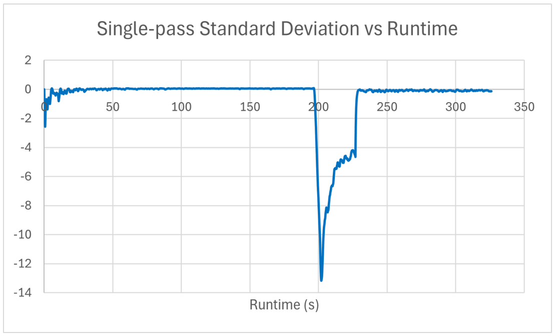


(B) Simulated Attack

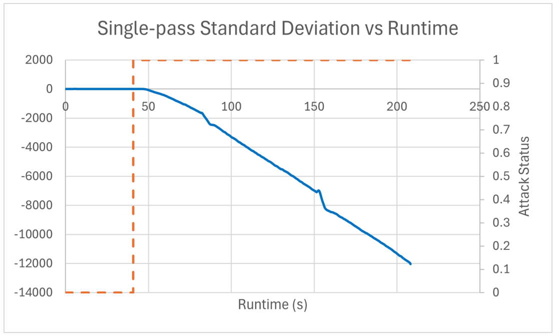
FIGURE 5.22: Cumulative Sum Metric

The inconclusive results obtained from the high and low calculations led to more through analysis of the intermediate values within the CUSUM process. Upon this analysis, the single-pass standard deviation and the running average of the sample data,

X_n, were reviewed and discovered to potentially have usable detection results. The values achieved by the standard deviation calculation of Equation 5.18 are shown below in Figure 5.23. In the first graph, Figure 5.23a, the standard deviation of the data sample, the difference between the two aforementioned values remains relatively flat around 0.0. The change in the graph can be equated to an aircraft turn, but it is important to note the value does return to approximately 0.0. When the simulated attack was initiated, the standard deviation of the data immediately begins to change as seen in Figure 5.23b where the orange dashed line indicates the attack was activated.



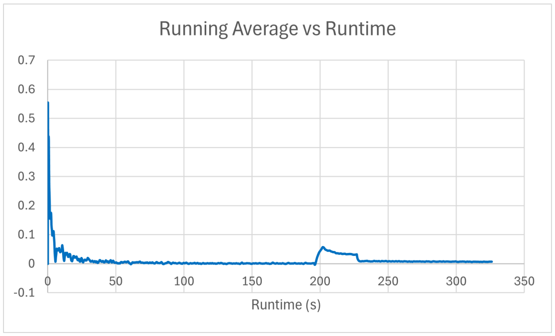
(A) Normal Flight



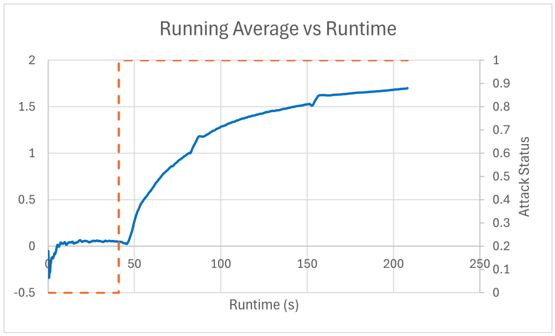
(B) Simulated Attack

FIGURE 5.23: Standard Deviation of the Input Data

The other value that showed potential in attack detection was the running average of the data sample and as shown in Figure 5.24. Since the sample was based on the difference between two values that should ideally be very similar, the overall average should remain around 0.0 as long as nothing has caused variation in the data. The running average during a normal flight, in Figure 5.24a, is shown to be very small, as would be expected. When the simulated attack beings, the average over the runtime begins to deviate from 0.0 within the first few seconds, as seen in Figure 5.24b.



(A) Normal Flight



(B) Simulated Attack

FIGURE 5.24: Running Average of the Input Data

5.2.3 Exponentially Weighted Moving Average

Exponentially Weighted Moving Average (EWMA) is another statistical method tested for its usability in detecting sensor errors. EWMA is unique in that it is used for monitoring the averages of data within a specific process based on a predefined weight to place more emphasis on present data or historical values. The closer the weight is to 1, the more emphasis present data has on the resulting output. The weight, λ, is calculated with Equation 5.22 where N is the number of historical values that influence the current calculation [122].

$$\lambda = \frac{2}{N+1}, \ (where N \geq 1) \tag{5.22}$$

The calculation of EWMA is shown below in Equation 5.23. This equation resembles that of the typical low pass filter, but with a different method of determining the weights. First, the weight is multiplied with the current data sample X_t. Next, the calculated EWMA from the previous time step, $EWMA_{t-1}$, is multiplied by the weight subtracted from one. Lastly, the two resulting values are added together.

$$EWMA_t = \lambda * X_t + (1 - \lambda) * EWMA_{t-1} \tag{5.23}$$

This method also contains a component to calculate the variance of the calculated EWMA statistic. This calculation follows Equation 5.24 where λ is the value calculated previously and σ_X is the standard deviation of the input observation. The standard deviation of the value used in the variance calculation of the EWMA method is calculated as described above in Equation 5.18 using a single-pass method to provide an estimated standard deviation.

$$s^2_{EWMA} = \frac{\lambda}{2-\lambda} * \sigma^2_X \tag{5.24}$$

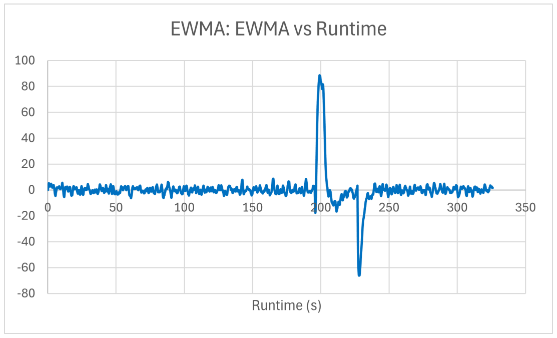
The last part of this statistic are the control limits. EWMA has control limits to monitor when the data is within the appropriate range. The control limits use the target value of EWMA, which in this case is 0.0, as no error should be present among the values in question, and the square root of the variance from Equation 5.24. The upper and lower bounds can be scaled by k, which gets multiplied to the root of the variance, depending on the range of data that is acceptable as under control. Equations 5.25 and 5.26 show this calculation for the upper and lower control, respectively. If the calculated $EWMA_t$ value from Equation 5.23 is within the upper and lower control limits, the sample can be considered under control.

$$EWMA_{ucl} = E_0 + k * \sqrt{s^2_{EWMA}} \tag{5.25}$$

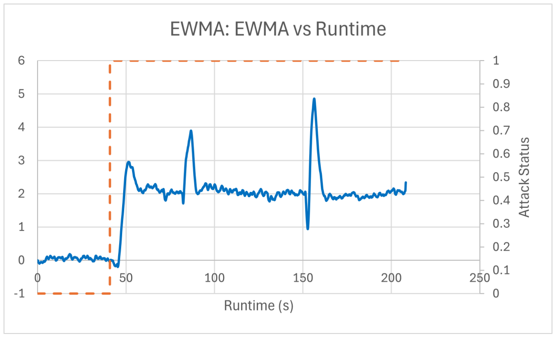
$$EWMA_{lcl} = E_0 - k * \sqrt{s^2_{EWMA}} \tag{5.26}$$

Below is a set of graphs to demonstrate the effectiveness of the EWMA metric in providing useful information when a variation within the data occurs. For the purposes of the calculations, the value of $N = 100$ was used for how many observations to be considered in the memory for the λ calculation. The data rate of the detection process is 20Hz, meaning that 100 values is 5 seconds worth of data. Figure 5.25 shows the graph of the calculated $EWMA_t$ values as calculated by Equation 5.23. The graph on the left, Figure 5.25a shows the calculated EWMA during a normal flight. Again, the input observation for X_t is the difference between the FCS reported vertical velocity and the energy model calculated vertical velocity. Other than a likely turn that occurred in the flight simulation during data collection, the EWMA remains close to 0.0.

Figure 5.25b represents what happens to the calculated EWMA value during a simulated attack. As seen in the graph, when this walk-off attack beings, indicated by the orange dashed line, the moving average increases and eventually settles at some value. The value it settles on is not unique and can change depending on the type of attack or how much the attack affects the aircraft's flight.



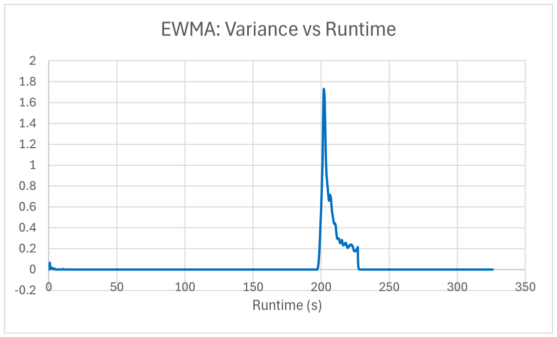
(A) $EWMA_t$ Normal Flight



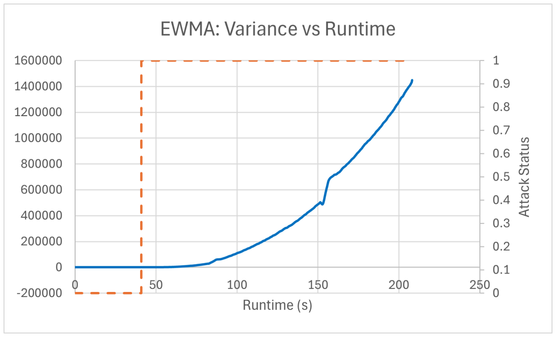
(B) $EWMA_t$ Simulated Attack

FIGURE 5.25: Variation from Exponentially Weighted Moving Average Calculation

The exponentially weighted moving average process also contains a calculation for the variance of the data, which is seen in the graphs of Figure 5.26. When no attack is running, the variance of the input data remains low and returns low even after a small variance is registered, as exhibited by Figure 5.26a. When the simulated attack is initiated, the variance begins to increase significantly, the value of which going past $1.0 * 10^5$ within the first 60 seconds of the attack being initiated and growing rapidly, as shown by the data of Figure 5.26b.



(A) Variance (s^2_{EWMA}) Normal Flight



(B) Variance (s^2_{EWMA}) Simulated Attack

FIGURE 5.26: Exponentially Weighted Moving Average Calculation

This large scale is a drawback of using variance for any error detections as the data contained within this graph is very large when only a small error was present in the data. If the variance was used on another simulated attack or even a different model that has significantly diverged from the ideal value, this variance would grow too large to be useful. A large variance can occur even when no attack is present, making this method not as viable as others described, due to the unknown bounds of the scale the variance results can have.

5.2.4 Xbar Control Charts

Another statistical measure considered was the Xbar control chart method. Xbar control charts are used to observe the average value of a particular process when the subgroup of data contains continuous data. There are three main types of Xbar control charts which are selected depending on the data subgroup size: individual and moving range, Xbar and standard deviation, and Xbar and range. Control limits are calculated alongside each Xbar chart to determine if the process is within normal limits. When the process average goes beyond the control limits, the process is not performing as intended. In the context of the flight controller, a process going beyond the control limits can mean the flight controller is experiencing a cyber-attack effecting one or more sensors.

IMR Charts The individual and moving range (IMR) chart is the combination of the individual chart and moving range chart. An individual chart, or I chart, is used for displaying single data points and tracks the average and shifts in any process with data collection occurs at regular time intervals. A moving range chart, or MR chart, is similar in that it requires data collection at regular intervals but tracks the variation in data points. The main use of IMR charts is when only one set of continuous data is available to look for process behaviors, as all of these chart methods calculate upper and lower controls [123].

Xbar S Charts Next, Xbar and standard deviation (Xbar S) charts are typically reserved for continuous data with subgroups of greater than ten. Xbar by definition is the average of a single sample value from each subgroup. The S component is the standard deviation of all the data of all subgroups. The combination of the Xbar and S charts result in the analysis of the average and standard deviation over time but does require the data to have a normal distribution [124].

Xbar R Charts The third method is the Xbar and range (Xbar R) charts which are used when there are more than two but less than ten subgroups of continuous data. This method, like Xbar S, starts with the average of one sample per subgroup, but instead of using standard deviation for the second chart, the range of the data is used. This range, R, is calculated by taking the maximum value of the set of values from each subgroup and subtracting the minimum value. This combination creates an analysis of the average value and range among the values over time and also requires normal distribution [125].

Based on the use cases of the three statistical chart methods, the Xbar R was selected due to the span of subgroup sizes that can be encompassed by the equations. In all considered uses of this method, between two and four subgroups of data are either readily available or can be calculated at run-time to feed into the typical set of Xbar R equations. The other two methods, IMR and Xbar S, were not selected since the applicable subgroup sizes did not match with the data available.

The equations and implementation of Xbar R are described below since Xbar R was the approach chosen to compare its effectiveness against other statistical metrics. As previously mentioned, $\overline{X}$ is calculated by taking the average of each value, one from each subgroup. All the $\overline{X}$ values for all the data samples collected or iterations stored are averaged together to calculate $\overline{\overline{X}}$. Next the range value is then computed for each data collection, R, and all ranges are averaged together to get $\overline{R}$. During the implementation of this method, 50 samples of data were stored for the calculations resulting in $\overline{\overline{X}}$ and $\overline{R}$ are the averages of 50 values calculated by the per sample average and per sample range, respectively.

The first step in using the Xbar R equations is determine the subgroup size and using the Xbar R control chart constants for determining the A_2, D_3, and D_4 constants, obtained from [124]. The first set of equations below, Equations 5.27 and 5.28 are used for calculating the upper and lower control limits, defined as ucl and lcl, of X, respectively. To solve these two equations, first the A_2 value for the correct subgroup size is selected from the constants table then multiplied with the average of the range values, $\overline{R}$. This value is then added to the average of the average X values, $\overline{\overline{X}}$, for the upper control limit, or subtracted from $\overline{\overline{X}}$ for the lower control limit.

Subgroup Size	A_2	D_3	D_4
2	1.880	0.000	3.267
3	1.023	0.000	2.574

TABLE 5.5: Constants for Xbar and Range Control Charts

$$X_{ucl} = \overline{\overline{X}} + (A_2 * \overline{R}) \tag{5.27}$$

$$X_{lcl} = \overline{\overline{X}} - (A_2 * \overline{R}) \tag{5.28}$$

Similar to the control limits calculations of X, the range value also has its own set of upper and lower control limits. To calculate the upper limit of range of Equation 5.29, R_{ucl}, the table is again consulted for the proper value of the D_4 constant then multiplied with the average of the range values, $\overline{R}$. The lower limit in Equation 5.30, R_{lcl}, is calculated using the D_3 constant and multiplying with $\overline{R}$.

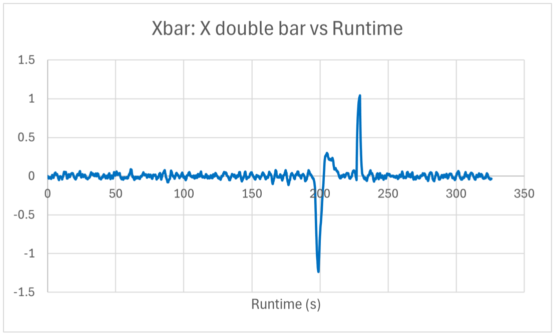
$$R_{ucl} = D_4 * \overline{R} \tag{5.29}$$

$$R_{lcl} = D_3 * \overline{R} \tag{5.30}$$

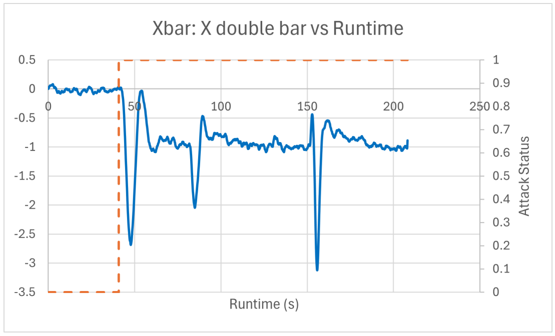
Since the two main components of the Xbar and Range metric are the previously described $\overline{\overline{X}}$ and $\overline{R}$ values, these two values were graphed to view their results during the run-time of both a normal and a simulated attack flight. First are the graphs of $\overline{\overline{X}}$

in Figure 5.27. For the purposes of this initial test, the two values that were averaged together at each time step, achieving the $\overline{X}$, were the FCS vertical velocity and the energy model vertical velocity. The averages from each time step were all averaged together over 5 seconds of runtime to calculate the $\overline{\overline{X}}$, the same time segment of data used for the EWMA metric for an accurate comparison.

In Figure 5.27a, it is seen that the average of the data over the most recent 5 second time interval remains negligible when no attack is present. This low value is the expected average as the average for the data was designed to be centered around 0.0 to create a comparison for all methods. In the instance the altitude, airspeed, or any other typically non-zero value is used for a comparison, the scale of those values would vary, resulting in any detection method difficult to fit to all scenarios. When the simulated attack is activated, Figure 5.27b shows the $\overline{\overline{X}}$ change very quickly and clearly indicated that a variation among the input data is present.



(A) Normal Flight

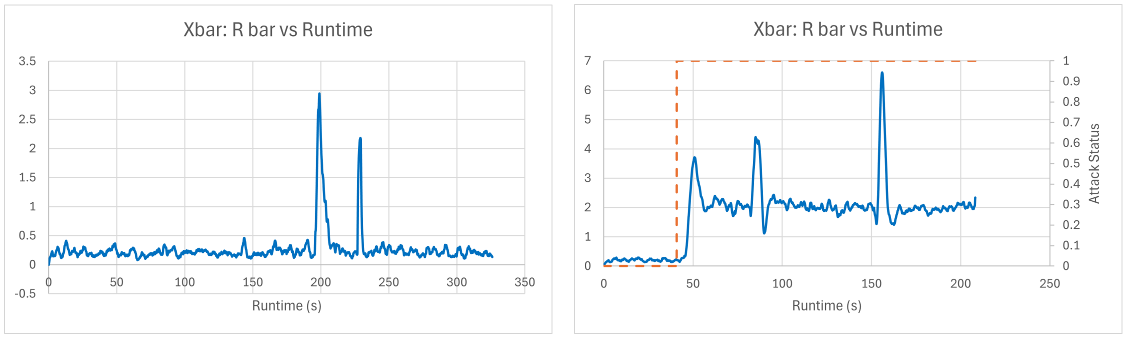


(B) Simulated Attack

FIGURE 5.27: Xbar and Range Metric: $\overline{\overline{X}}$

The second component to the Xbar and Range method is the range. This range value is based on tracking the difference between the least and greatest value among a data sample as each time step. In this test, since only the two previously mentioned values were used, the range R was just the difference between the two. Similar to $\overline{\overline{X}}$, $\overline{R}$ is based on the average of all the range values over the last 5 seconds of runtime. Figure 5.28 shows the resulting data for the average range value in the two instances used for the previous comparison.

The graph, represented by Figure 5.28a, shows the $\overline{R}$ of the data over the normal flight data collection, indicating a minimal error is present between the two input observations. With the simulated attack, the average range begins to increase when the variation in the data is present, as exhibited by Figure 5.28b.



(A) Normal Flight (B) Simulated Attack

FIGURE 5.28: Xbar and Range Metric: $\overline{R}$

5.2.5 Correlation Coefficient

The last method considered and tested was the correlation coefficient. The focus of correlation is to measure the strength and direction between two values of interest with a linear relationship. While first thoughts indicated the relationships under test are likely non-linear, this method was still implemented to determine if any values could be of use. The correlation coefficient provides information on the regression of the data as it changes with respect to the other input. The correlation is done by essentially applying a line of best fit to the data. The resulting slope of that line is the resulting coefficient, bounded between -1.0 and 1.0. A coefficient of 1.0 indicates a positive correlation, meaning that the two observations are increasing at the same rate over time. A negative correlation, indicated by a value of -1.0, indicates that as one point increases, the other decreases. Lastly, if the coefficient is around 0.0, there is no correlation between the two observations and that there are no commonalities between the driving forces that create the results observed [126].

The calculation of the correlation coefficient, ρ, is a two-step process, first requiring the calculation of the covariance between the two observations. Covariance is used to evaluate the total variation between two random observations and their respective expected values. This is then used to determine if the relationship is proportional or inverse, but does not specify any dependency between the observations [127].

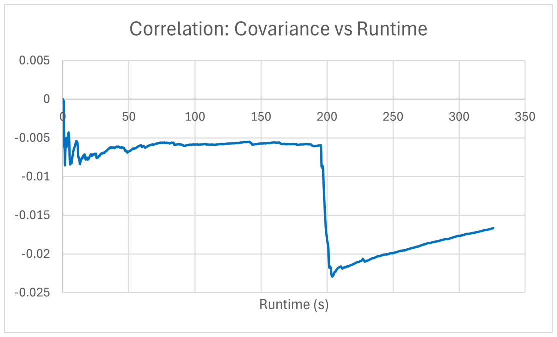
To compute the covariance, as shown in Equation 5.31, the difference between each sample X and the average of all collected samples $\overline{X}$ is computed for both observations. These differences are multiplied together, then summed up from the beginning of run-time to the most recent data point. This summation is then divided by the number of values less 1 to compute the sample covariance instead of the population variance.

$$Cov(X_1, X_2) = \frac{\sum_{i=0}^{n} \Big((X_{1_i} - \overline{X}_1) * (X_{2_i} - \overline{X}_2) \Big)}{N - 1} \tag{5.31}$$

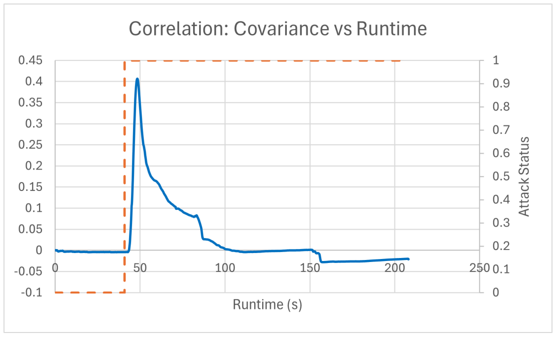
Next, the standard deviation of each observation is computed in the same single-pass method as previously described by Equation 5.18. The correlation ρ with respect to the two samples is then calculated by taking the covariance from Equation 5.31 and dividing it by the product of standard deviations of the two observations.

$$\rho_{X_1,X_2} = \frac{Cov(X_1, X_2)}{\sigma_{X_1} * \sigma_{X_2}} \tag{5.32}$$

In a similar method to the previous metrics, the covariance component of the correlation metric and the correlation coefficient were graphed to determine if this method was usable in providing unique features for proper data variation detections. The input data for this test was the same as used previously, the vertical velocity from the FCS as X_1 and the energy model X_2. The values were run through the corresponding methods and equations as discussed for the correlation coefficient method. The resulting covariance between X_1 and X_2 as calculated by Equation 5.31 is shown in Figure 5.29. When no attack is running, the covariance shown in Figure 5.29a is very minimal and is unclear if any useful information would be obtained when a simulated attack is run. Figure 5.29b confirms this idea that even during an attack, there is an initial spike in the covariance followed by a return to normal. This return leads to the conclusion that looking at the covariance alone does not provide an indication the sources of data contain an error.



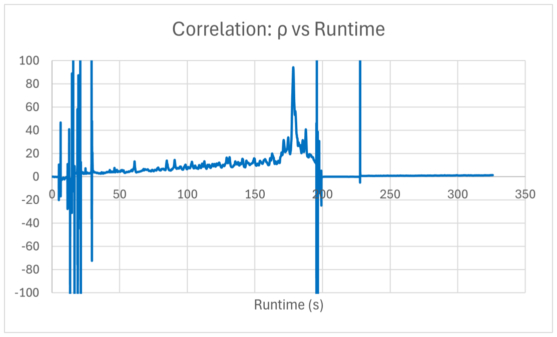
(A) Normal Flight



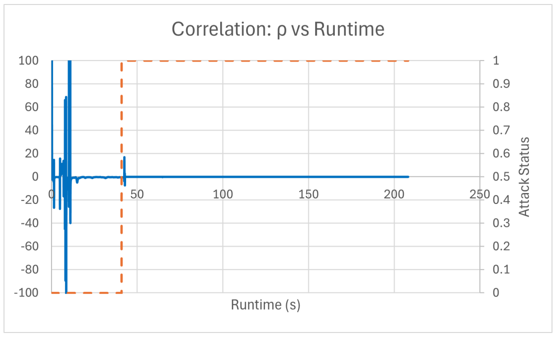
(B) Simulated Attack

FIGURE 5.29: Covariance of X_1 and X_2

Next, the correlation coefficient with respect to the input data, ρ_{X_1,X_2}, was graphed in Figure 5.30 to determine if this method could provide any indications of data variations. Figure 5.30a displays the ρ value during the data collection of a normal flight. As seen in the graph, when the data should be maintaining the same value as each other, the correlation coefficient is inconclusive of the intended observation. This inconclusiveness is matched in the values during a simulated attack of Figure 5.30b. This proves that both values provided during the calculation of the correlation coefficient metric are not usable in providing any inference on issues within the run-time data.



(A) Normal Flight



(B) Simulated Attack

FIGURE 5.30: Correlation Coefficient (ρ)

5.2.6 Summary of Control Limits

Of the statistical metrics discussed, the exponentially weighted moving average and Xbar control charts both have specific methods for calculating upper and lower control limits. The idea behind the control limits to be calculated on the data offline after all the values are available. This would provide for an accurate upper and lower bound control limits to indicate if any value is outside the determined nominal range, that would be considered inaccurate data. A calculation for the control limits was attempted at run-time, but the result is that any change in the input data results in a change in the control limit, ensuring that all values received are within range.

After reviewing the results of these control limits, two methods were tested to see if adding an external factor onto the control limits could reduce how much it is affected by sudden spikes in the data. These two methods were low pass filtering and averaging. Running the control limits through a low pass filter and averaging the data over a specific time segment did allow for sudden data anomalies to go outside the allowable thresholds. This method became too specific to each implementation of the metric and the scale of input data, resulting in abandoning the use of the run-time control limit calculations. Other approaches to create a more uniform process for determining the correct threshold for valid data were explored and will be discussed in Chapter 5.3.

5.3 Operating Bounds and Threshold Calculations from the Decision Metrics

The results of the decision metrics collected during all flight scenarios, normal and with various simulated attacks, to see if any distinguishable features could be identified among these datasets. For the methods that had unique features, a repeatable method to extract the boundaries was required. This limit on what constitutes normal aircraft behavior is used by the detection mechanism, discussed in the next section, to detect if a particular value has veered outside its expected behavior. The two techniques tested for this purpose were the confidence interval and the prediction interval, which were only usable when the decision metrics could be implemented, requiring a comparison of data.

5.3.1 Confidence Interval

The confidence interval, CI, was the first method used to determine the aforementioned threshold. It is a statistical inference method to estimate the range that a set of observations is likely to fall within, based upon specific calculated values from previously observed data. The data collected during normal flight simulation test runs of the metrics when no attacks were conducted was used in order to obtain the average and standard deviation of the data in normal flight for use in calculating the CI. This followed the method below in Equation 5.33 where X is the specific component of a statistical metric used, z is the number of standard deviations that are considered acceptable data, and N is the number of data samples in X. In implementing the CI, a value of 3 was used for the number of acceptable standard deviations from the average as 2 resulted in too narrow of a margin resulting in too many false detections [128].

$$CI = \overline{X} \pm \left(z * \frac{\sigma_X}{\sqrt{N}} \right) \tag{5.33}$$

Upon further data collection of the detection function testing and its effectiveness, it became apparent that the CI equation created too narrow of margins to compensate for the noise that can occur in the data from the recorded flight data. It was clear that another method for determining the threshold for the values was required.

5.3.2 Prediction Interval

The prediction interval, PI, was tested and determined to provide a wider range of acceptable data which reduced the false detections without jeopardizing the correct detection rate. The PI is a method of statistical inference used to forecast the range in which future variables of specific observations are most likely to remain within [129].

The calculation of the PI is similar to the CI in the values it requires: the average $\overline{X}$, the standard deviation σ_X, and the number of samples N of the observed data. The difference is the method of calculation, as seen below in Equation 5.34. The standard deviation is multiplied with a value dependent on the number of data samples used to compute the average and standard deviation, and is then added to or subtracted from the observation mean. Equations that provide a range of values are typically added and subtracted with the mean of the data so that the maximum and minimum ranges are scaled around the mean. This is helpful in instances where the data typically has some inherent error that always shows in the average of the data, as is the case in most of the aircraft simulation data under test.

$$PI = \overline{X} \pm \left(\sigma_X * \sqrt{1 + \frac{1}{N}} \right) \tag{5.34}$$

As an example to illustrate the difference between the calculated values from each equation, assume the average value is $\overline{X} = 0.61$ and the standard deviation is $\sigma_X = 10.27$. It should also be assumed that these values remain relatively similar when the data size is increased. The data in Table 5.6 represents the calculated values of the confidence interval and prediction interval at different sample sizes *N*. As seen in the CI data, as the number of values in the observed data set used to calculate the average and standard deviation increase, the value drops off significantly before beginning to settle out on a particular value as N gets fairly large. On the other hand, the PI values across the same set of data exhibit minimal variation even as the data set size changes significantly. This leads to PI being a more stable solution for calculating the range of acceptable data, since the input data size for each metric under test can vary. These variations are dependent on how long the flight simulations are run and the rate at which data was logged from the flight.

N	CI	PI
10	10.353	11.381
50	4.967	10.982
100	3.691	10.931
500	1.988	10.890
1000	1.584	10.885
5000	1.046	10.881

TABLE 5.6: Confidence Interval (CI) and Prediction Interval (PI) Comparison against Different values of N

5.3.3 Non-Mathematical Boundary/Threshold Method

Based on the architecture developed, the confidence and prediction intervals could only be applied to the values obtained from the decision metrics. The metrics contain observations from the energy and heading models. These two models are unique in that the calculated results are estimates that are directly comparable with other sensor or FCS computed values.

The turn radius model required a different implementation than the energy or heading models, as previously mentioned in Chapter 5.1. Due to the scale of the calculations and the type of value provided from the turn radius model, there is no directly comparable value as part of the FCS or the sensors. As a result, the decision metrics are not implementable regarding the turn radius model, therefore the CI or PI calculation can not be performed. The threshold determination for the turn radius is based upon the graph displaying the relationship between the roll angle and corresponding scale of the turning radius, is illustrated by Figure 5.31. As can be seen, as the roll angle increases, the turn radius drops rapidly. Accounting for noise in the roll data caused by constant minor corrections from the autopilot controller, the turn radius that corresponds to a roll angle of 2.0° was selected and tested, the results of which are presented in Chapter 6.

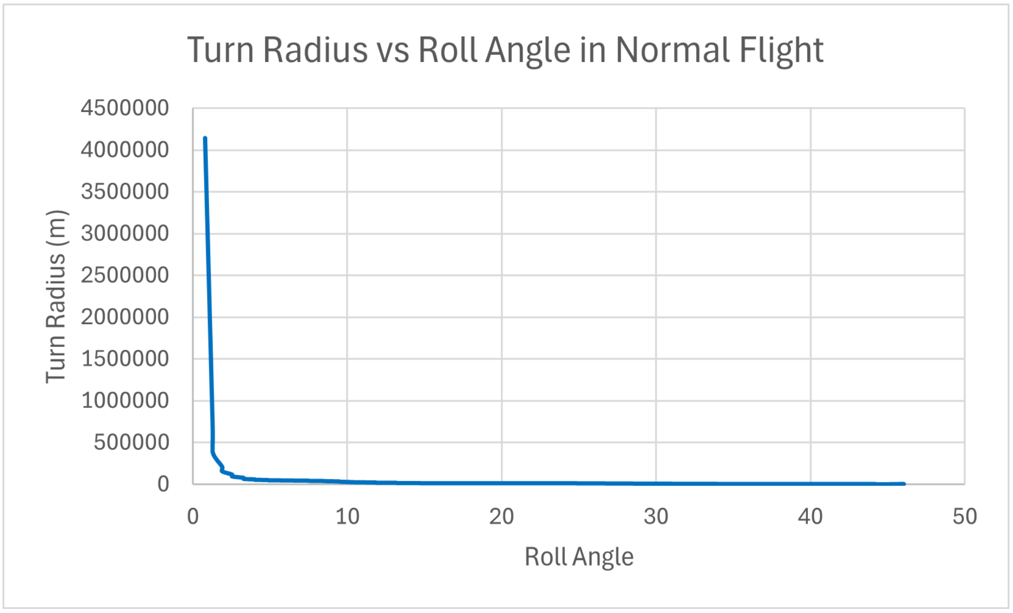


FIGURE 5.31: Turning Radius Calculation and Scale versus Indicated Roll Angle from the FCS

5.4 Detection Algorithm used in the Functional Monitor

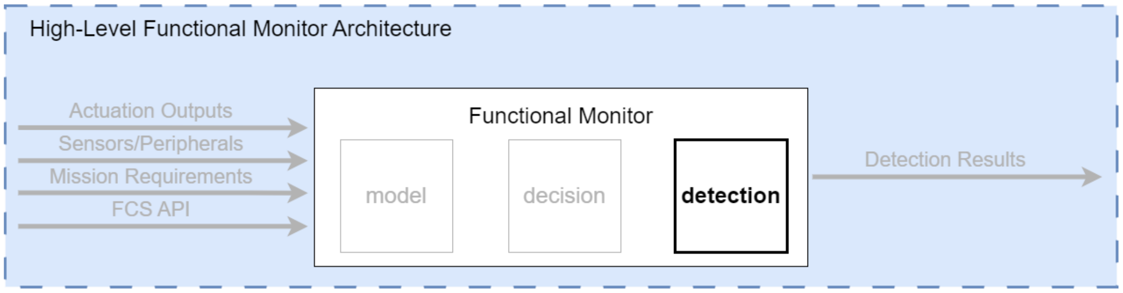


FIGURE 5.32: Simplified Functional Monitor Architecture with Detection Emphasized

The last component of the real-time functional monitor is the detection block, the location of which is emphasized in Figure 5.32. The detection block implements a specific algorithm for conducting the attack detections, independent of which statistical method is being used. The pseudo-code for the detection algorithm is shown below in Algorithm 2. The function begins with the inputs of the current value of whichever metric is being analyzed, the loop time dt, and a delay value for the detection. The delay value is used to filter out sudden spikes in the data that return to normal within a few cycles. The values under observation must remain outside the threshold for a certain amount of time before being considered an error.

In the first *if* block, the value is checked to see if it is above a preset threshold and if the attack has not yet been flagged, followed by checking if this error has occurred for the minimum time. If these conditions have been met, the attack is flagged, and if not then the current duration counter is incremented. The *else if* is used to lower the threshold to release the attack, resulting in fewer toggles between a detection flag from true to false and vice versa. The *else* block is for releasing the attack and is also dependent on the time the value goes back to within range before resetting the detection flag. This code also has a runtime counter of the attack for later use to determine the accuracy of the methods used.

The detection algorithm developed for the turn radius model is based on the turn radius calculation and the roll angle from the FCS. The threshold is determined from the data of Figure 5.31 to select a corresponding set of turn radius and roll angle. These two values are used in combination and are compare against this selected value to see if the specific value is above or below a threshold. For example, a threshold of $5.0 * 10^5$ m for the turn radius would correspond to approximately 1.5°. The detection algorithm for these values would be used to ensure that the calculated turn radius from the model matches the measured roll angle of the aircraft. If the turn radius is below the threshold, but the roll is around 0.0°, this would indicate an error among the data. The resulting detection of an inaccuracy between the magnitude of the aircraft's roll and its turning radius would be flagged as an attack on the IMU.

As previously mentioned in the model creation summary of Chapter 5.1, detections of the pitch and yaw vary slightly from the roll detection, even though these three measurements originate from the same sensor. When the pitch walk-off is applied, the resulting correction of the angle by the autopilot induces either an increase or decrease in altitude. This is detected by the energy model and is used in combination with the FCS pitch and the vertical velocity in order to distinguish the pitch walk-off from an airspeed or altitude walk-off. The values obtained during this simulated attack show

Algorithm 2: This pseudo code function shows the general format for conducting an attack detection algorithm

```
function run_detection (current_value, dt, min_delay);
if (abs(current_value) > threshold) && (detection_flag == False)) then
    //determine if the value is above the threshold for a minimum amount of
      time
    if current_duration > min_delay then
        detection_flag = True;
        total_detected_duration += dt;
        current_duration = 0.0;
    else
        current_duration += dt;
    end
else if (abs(current_value) > (threshold*0.707)) && (detection_flag == True)) then
    //for creating a lower threshold for value checking
    detection_flag = True;
    total_detected_duration += dt;
else
    if (detection_flag == True) && (current_duration < min_delay)) then
        detection_flag = True;
        total_detected_duration += dt;
        current_duration = 0.0;
    else
        //fully release detection flag
        detection_flag = False;
        current_duration = 0.0;
    end
end
runtime_of_this_detection_algorithm += dt;
return;
```

that while the FCS indicated pitch is negative, the aircraft is climbing and when the pitch is positive, the aircraft is descending. Using this mismatch in what is physically possible with the aircraft along with the energy model indicating an issue, the IMU attack is flagged.

The yaw walk-off is partially registered by the heading model but not fully, meaning that the detection rate from the heading model is less than 50.0% but not insignificant. Another value is needed to differentiate between whether the detection is being flagged by an error in the GPS or IMU is to analyze the error between the GPS heading and the FCS indicated heading. If an error is present while the heading model detection rate is low, the IMU attack is flagged. The error between these two values was not used for simply detecting a GPS walk-off, as another metric was needed in order to accurately determine which sensor exactly contained the data error.

During a simulated GPS walk-off, due to the change in heading in the aircraft, the turn radius model begins to provide some positive detections, but the detection rate is less than half but not insignificant. In order to identify if the GPS is the source of the error, data from the exponentially weighted moving average from the heading model detection algorithm and the turn radius model are used. The GPS attack is flagged if the heading model provides a positive detection and the turn radius detection rate is below 50.0%.

If the energy model and resulting exponentially weighted moving average decision metric flags an error, that indicates an error on either the airspeed or altitude sensor. In order to determine which sensor is the cause, data from the GPS is used as a point of comparison. If the error is indicated from either of the two sensors just mentioned, and the altitude from the barometer and GPS are in agreement, then the error originated from the airspeed sensor and is flagged accordingly. If the barometer and GPS altitude are not in agreement after the metric had indicated an error, then the altitude sensor is flagged.

Chapter 6

Results

To test the effectiveness of the functional monitor and the techniques described herein, many flight simulations were run and various combinations of the decision metrics, as introduced in Chapter 5.2, were implemented. The data used for the decision metrics included the calculations from the models of Chapter 5.1 and the corresponding sensor or FCS data. The results from these tests were collected in log files of each flight and analyzed with the threshold and detection algorithms, introduced in Chapters 5.3 and 5.4, respectively. The overall results of applying these techniques to detect cyber-attacks or sensor faults on simulated aircraft operations are described in this chapter.

The functional monitor architecture described in Chapter 5 in support of Run-Time Assurance showed the overall process that was used to implement the functional monitors discussed in this dissertation. To provide a more detailed context of the architecture, an example containing the components discussed in previous chapters is shown below in Figure 6.1. The difference between this diagram and the previous is that this diagram shows an example test case of the architecture, including labelling each component and the specific values that were used.

For the model block, the energy model was tested, which takes the inputs of the throttle percent from the actuator output block and the airspeed from the sensor/peripherals block. The results from the model are then passed to the decision metric block, in this case the exponentially weighted moving average (EWMA) was implemented. The decision block requires a point of comparison from the FCS for the value received from the model; in this case the vertical velocity. The output from

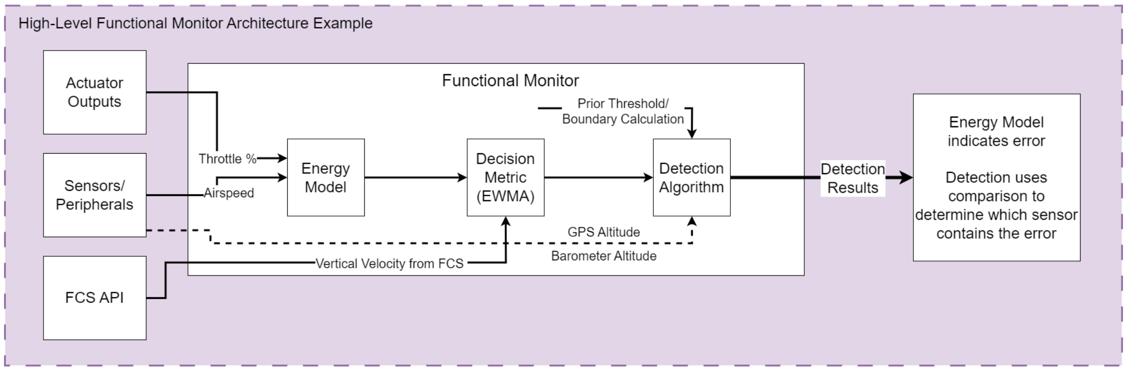

High-Level Functional Monitor Architecture Example
Actuator Outputs
Sensors/ Peripherals
FCS API
Functional Monitor
Throttle %
Airspeed
Energy Model
Decision Metric (EWMA)
Prior Threshold/ Boundary Calculation
Detection Algorithm
GPS Altitude
Barometer Altitude
Vertical Velocity from FCS
Detection Results
Energy Model indicates error
Detection uses comparison to determine which sensor contains the error

FIGURE 6.1: Block Diagram of an Implemented Functional Monitor Architecture

the decision metrics are then forwarded to the detection algorithm, which contains the proper operating bounds for the value in question.

The detection algorithm then outputs information related to whether or not the EWMA metric from the model indicates an error. In order to provide more details as to which sensor is the cause of the issue, data from other sensors is fed to the detection block to compare against. This comparison can provide a more accurate guess as to the location of the error. It should be noted that the other sensor must be assumed to be trustworthy in order to be used in this error location determination. This location determination based on other sensor data was not part of the initial research goals, but was included as a means to further the testing of the Functional Monitor architecture.

The graphs below within Figures 6.2 and 6.3 display the results of a few decision metrics using the data from the energy model and heading model, respectively. The graphs show the detection rate in percent of the decision metrics during four tested flight scenarios, where the letters below correspond to the sub-figure letter:

(A) Normal flight: a standard waypoint sequence with no simulated walk-off attacks

(B) Airspeed walk-off: the airspeed sensor underwent a simulated walk-off attack

(C) Altitude walk-off: the altitude sensor underwent a simulated walk-off attack

(D) GPS walk-off: the GPS data underwent a simulated walk-off attack

In each graph, the left-most column, labeled expected, displays the intended detection rate based on the type of simulated walk-off attack and the models used. The data collected for the purposes of this testing was retrieved from the log files when each of the corresponding walk-offs were conducted.

6.1 Energy Model Results

First, Figure 6.2 shows the detection rates of the decision metrics based upon the energy model discussed in Chapter 5.1.1. Each of the remaining columns of data represent a different implementation of the decision metrics and are described in Table 6.1.

X-axis label	Description
Expected	This data is used to display what the detection rate should be (0% if the model should not detect an attack and 100% if it should)
Method E-1	The E_Δ from the error indicator metric based on the energy model output and the FCS vertical velocity
Method E-2	The E_Δ from the error indicator metric based on the single-pass standard deviation of the energy model output result and the FCS vertical velocity
Method E-3	The rate of change of the calculated standard deviation of the error between the energy model output and the FCS vertical velocity
Method E-4	The rate of change of the running average of the error between the energy model output and the FCS vertical velocity
Method E-5	The computed $\overline{\overline{X}}$ from the Xbar and R metric based on the average of the energy model output and the FCS vertical velocity from each run cycle
Method E-6	The computed $\overline{R}$ from the Xbar and R metric based on the range of the energy model output and the FCS vertical velocity from each run cycle
Method E-7	The exponentially weighted moving average value calculation based on the energy model output and the FCS vertical velocity

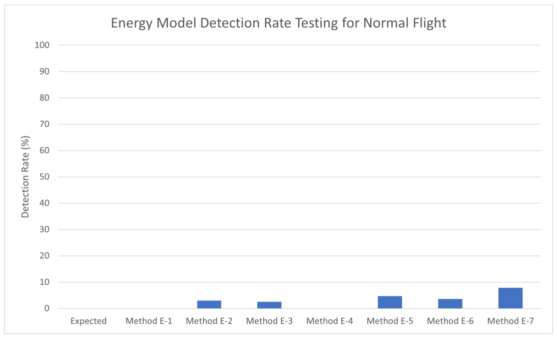
TABLE 6.1: Description for each column of the graphs and the detection method represented in Figure 6.2

As seen in Figure 6.2a, the expected detection rate is 0.0% as no attacks were running. All seven methods tested display detection rates of less than 10.0%. Any detection during a normal flight is considered a false positive. Initially, all methods began showing promising results with having low false positive rates. Once the walk-offs were initiated, certain models began to outperform the rest.

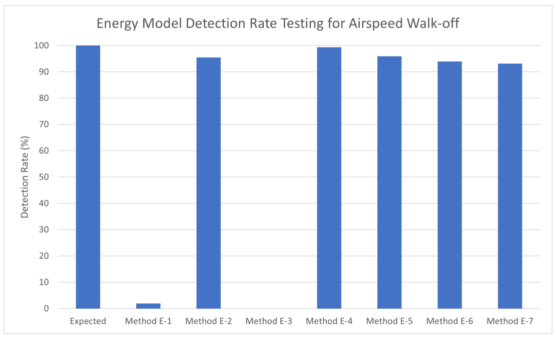
Since the energy model applies to both airspeed and altitude walk-offs, the detections during the airspeed and altitude walk-offs of Figures 6.2b and 6.2c, respectively, should all be indicating a significant detection rate. This was true of five of the initial seven methods tested. The two values of importance used in the metrics for the detection rate tested were the FCS and the energy model vertical velocities. The E_{Δ} of these values (Method E-1) and the derivative of the raw standard deviation of the difference between these two values (Method E-3), both exhibit a near 0.0% detection rate, indicating that these methods are not useful in detecting a sensor data error.

When the GPS walk-off was tested, the detections based upon the energy model should all be close to 0.0% as seen in the first column of Figure 6.2d. No detections should be registered by the energy model, which would lead to a 0.0% detection of any issues among the input observations. Any detections shown here would be considered false negatives. As seen in the graph, some decision metrics provide significant false negatives due to the nature of the GPS walk-off affecting the aircraft's overall flight, potentially in all three axes. Since Method E-3 has already been ruled out due to its lack of performance during the airspeed and altitude walk-offs, the next best metrics is Method E-7.

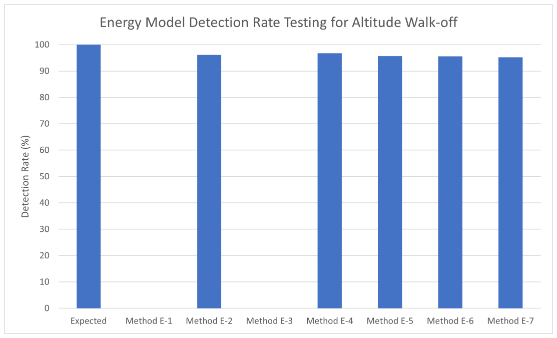
As described in Table 6.1, this method is the implementation of the exponentially weighted moving average calculation based upon the vertical velocities from the energy model and FCS. The other methods not mentioned, E-2, E-4, E-5, and E-6, were all ruled out as usable methods due to their significant false negative rates during the GPS walk-off.



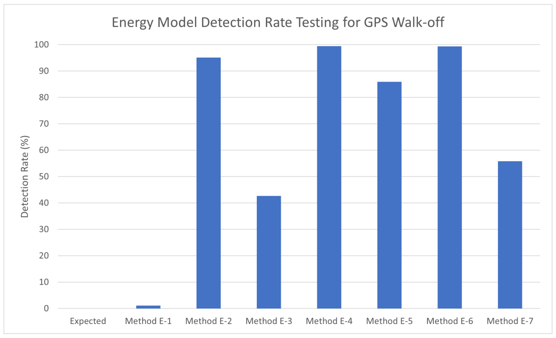
(A) Normal Flight



(B) Airspeed Walk-Off



(C) Altitude Walk-Off



(D) GPS Walk-Off

FIGURE 6.2: Detection Rate Analysis from Energy Model

6.2 Heading Model Results

The next set of testing presented in Figure 6.3 shows the detection rates of the decision metrics based upon the heading model discussed in Chapter 5.1.2. Each of the remaining columns of data represent a difference implementation of the decision metrics and are described in Table 6.2. The data collections from the heading model and corresponding decision metrics used are similar is structure but vary in order and implementation to the methods from the energy model that were previously discussed.

The two values of importance used in the metrics for the detection rate tested were the GPS heading and the calculated heading from the heading model. As seen in Figure 6.3a, the expected detection rate is 0.0% as no attacks were running. Five of the seven methods tested display detection rates of less than 10.0%. The rate of change of the standard deviation based on the observations indicated, Method G-3, and the $\overline{\overline{X}}$ value from the Xbar and Range metric, Method G-4, have false positive rates of over 20.0%.

Both the airspeed and altitude walk-off should have a 0.0% detection rate from the

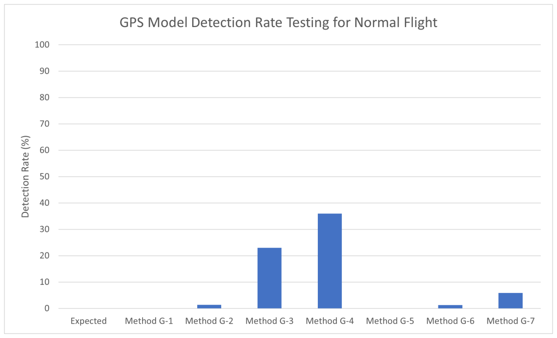
X-axis label	Description
Expected	This data is used to display what the detection rate should be (0% if the model should not detect an attack and 100% if it should)
Method G-1	The E_Δ from the error indicator metric based on the heading from heading model and the GPS heading
Method G-2	The rate of change of the running average of the error between the heading from heading model and the GPS heading
Method G-3	The rate of change of the calculated standard deviation of the error between the heading from heading model and the GPS heading
Method G-4	The computed $\overline{\overline{X}}$ from the Xbar and R metric based on the average of the heading from heading model and the GPS heading from each run cycle
Method G-5	The computed $\overline{R}$ from the Xbar and R metric based on the range of the heading from heading model and the GPS heading from each run cycle
Method G-6	The E_Δ from the error indicator metric based on the single-pass standard deviation of the heading from heading model and the GPS heading
Method G-7	The exponentially weighted moving average value calculation based on the heading from heading model and the GPS heading

TABLE 6.2: Description for each column of the graphs and the detection method represented in Figure 6.3

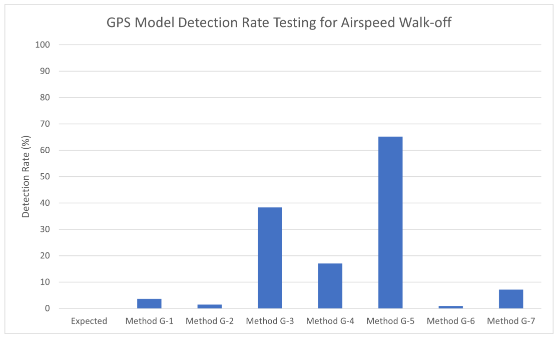
heading model, as illustrated by Figures 6.3b and 6.3c. Based upon the data presented by these two graphs, some methods show false negative rates, leading to those not being good for accurately detecting when there is an error with the GPS data. Other methods that had low false positive rates under normal flight circumstances, continued to report low to minimal false positive rates during the airspeed and altitude simulated attacks.

Lastly, the detections from the GPS walk-off attack are presented in Figure 6.3d. As shown by the first column, the expected detection rate should be close to 100.0% as the data observed should indicate an attack is occurring. Most of the methods have a detection rate of over 90.0% leading to promising usability of a few different methods.

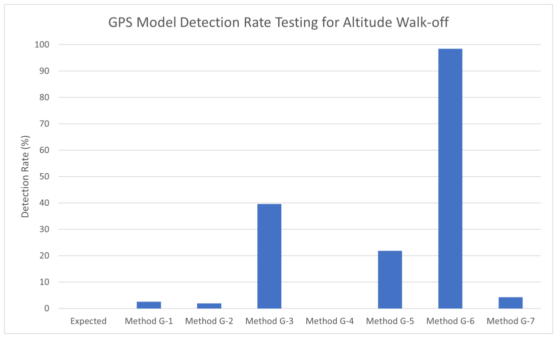
When all the detection rates are compared against each other and their expected values, one method stands out for GPS walk-off detections. The exponentially weighted moving average from the headings provided by the GPS and calculated from the heading model, Method G-7, outperforms the other methods as the most accurate among the flight scenarios tested. The result of the EWMA metric being the most accurate during the error detection testing is consistent among both the best for the heading model and the energy model described earlier.



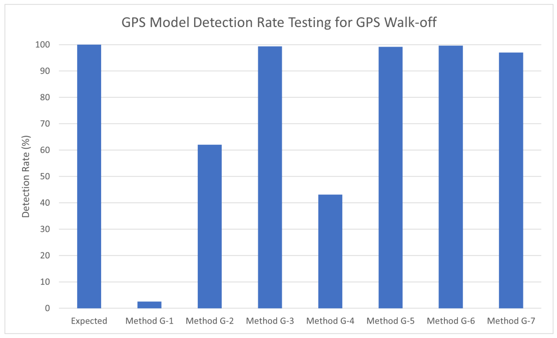
(A) Normal Flight



(B) Airspeed Walk-Off



(C) Altitude Walk-Off



(D) GPS Walk-Off

FIGURE 6.3: Detection Rate Analysis from Heading Model

6.3 Turn Radius Model Results

The process described above for the detection rate testing of the energy and heading models compared the detection rates of the various decision metric methods. In contrast, only one method was tested for the turn radius model. Similar to the other methods, the goal was to determine the effectiveness of detecting errors among the input data and the compared value used. In the detection algorithm for the turn radius model, the calculated radius and the FCS indicated roll were compared with the previously determined bounds (previously discussed in Chapter 5.3). The results of the comparisons of this data from the model and FCS were tested to verify the detection rate during the various simulated walk-off attacks previously tested against.

The detection rate results are shown below in Figure 6.4. The items across the x-axis are the simulated walk-off attack types of the turn radius model. As would be expected, the results of this detection show a negligible to 0.0 detection rate during the normal flight, airspeed walk-off, altitude walk-off, IMU pitch walk-off, and IMU yaw walk-off scenarios. During the previously mentioned scenarios, the expected detection rate is 0.0%. Any detection occurring during these tests would be considered false-positive detections.

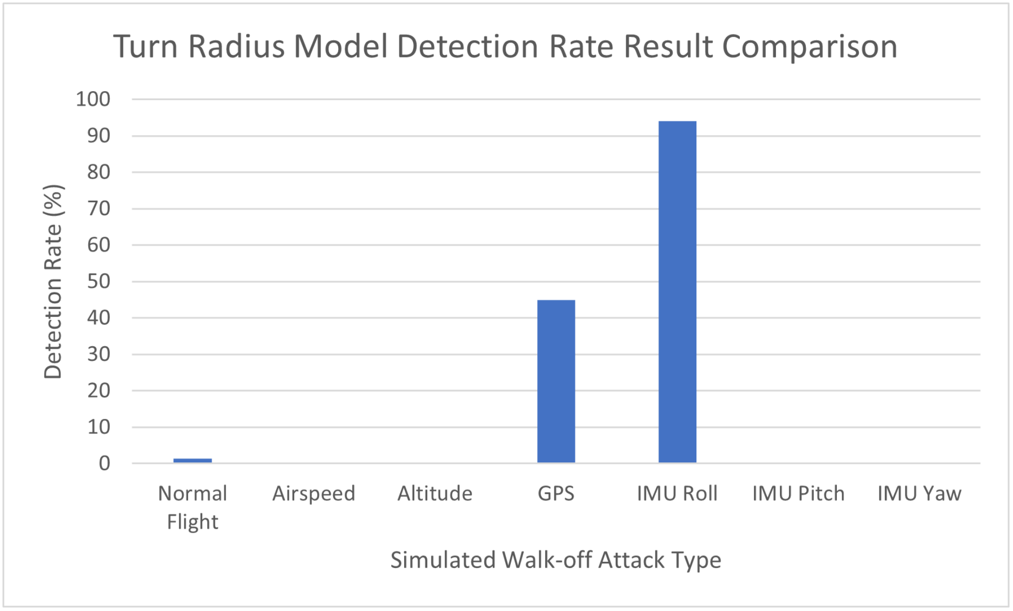


FIGURE 6.4: Detection Rate Analysis of Turn Radius Model

Detections are shown in the GPS walk-off and IMU roll walk-off tests. The detection rate from the turn radius model is ideally 100.0% as this is the scenario this model is built for since it is not covered by either the energy or heading models. As seen by the highest detection rate on the graph, the turn radius model provides for a greater than 90.0% detection rate during the IMU roll walk-off.

An unexpected value was the approximately 45% detection rate by the turn radius model data on the GPS walk-off. This high false positive rate is because the GPS walk-off targets the heading value, and the change in heading over time is one of the inputs to the turn radius model. The combination of these factors leads to the possibility of false positives which are seen in the data. As with all the detection methods discussed, a combination of positive detections along with taking the detection rate percentage into account can achieve accurate sensor detections.

6.4 Results From The Fully Implemented Functional Monitor

After an analysis of the comparisons between the developed methods discussed above and other decision and detection metrics discovered, all available metrics were implemented into the run-time altogether. The four simulated walk-off attacks that were discussed in Chapter 4 were conducted during one flight test with all detection systems running simultaneously. The comprehensive results from the detection algorithms implemented are shown in the graphs below. In the detection results displayed in the graphs, except for the intentional delay included in the algorithm, there were no false negatives exhibited by the functional monitors. While false positives can be annoying to the end user, understanding how mission-critical false negatives are is crucial. The detection algorithms encompass the sensor error detections of the airspeed sensor, barometric pressure sensor, GPS, and IMU. In the following graphs, the simulated attack is displayed by the solid black line and the detections from the corresponding algorithm are displayed by the dotted black line.

The simulated airspeed walk-off attack and corresponding detections are shown in Figure 6.5. As seen at the beginning of the graph, the simulated attack was initiated early in the run-time, as indicated by the solid black line. The detection shows a few instances of false positive data scattered throughout the run-time, but the algorithm correctly detects the airspeed attack as seen when the black dotted line changes from low to high and stays high for the duration of the simulated attack. The delay between when the simulated airspeed attack was initiated and the attack was detected was 7.866 seconds.

In the next graph, Figure 6.6, the simulated altitude walk-off attack is conducted. The methodology used for detecting the altitude provides for the least number of false positives, as seen by the few spikes later in the runtime. Similar to the data of the previous graph, the altitude detection accurately flags the barometric pressure sensor

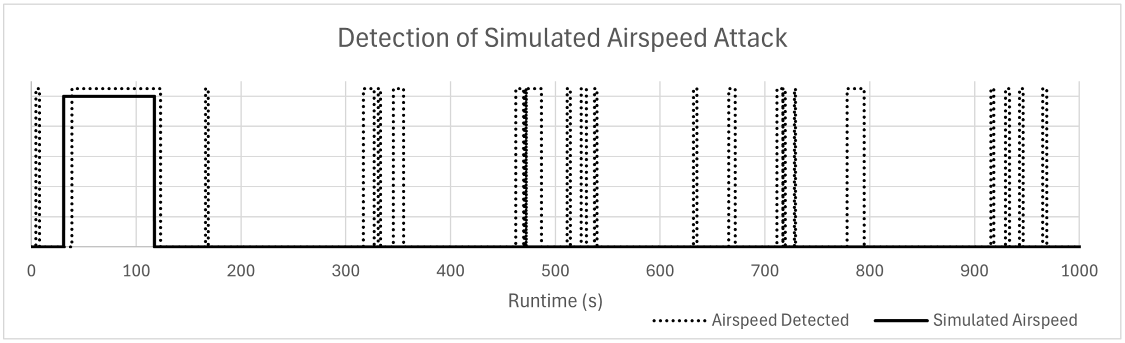


FIGURE 6.5: Simulated Airspeed Walk-off Attack and Functional Monitor Detections

error for the duration of the altitude walk-off. The delay between when the simulated altitude attack was initiated and the attack was detected was 7.366 seconds.

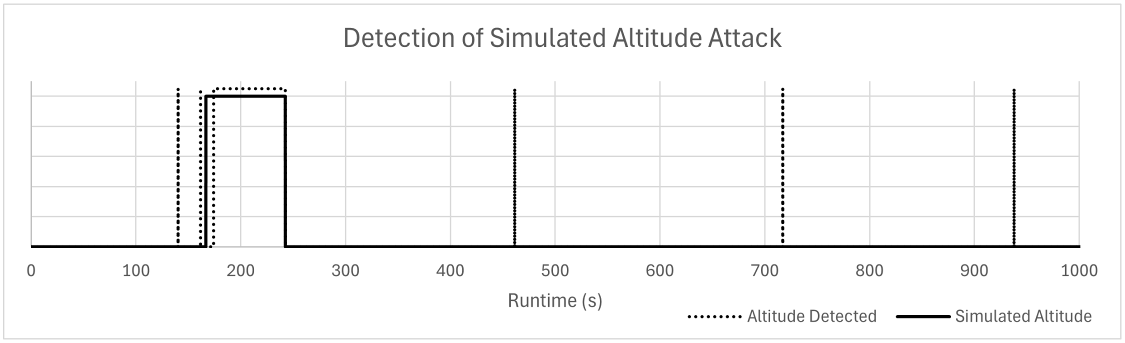


FIGURE 6.6: Simulated Altitude Walk-off Attack and Functional Monitor Detections

The results of the simulated GPS walk-off are presented in Figure 6.7. The GPS detections result in the most false positives, but these do not affect the overall detections, as the false positives are still such an insignificant portion of the run-time. Due to the method used for initiating the GPS error detection relying on the detection rate of two models, the GPS detections exhibit a slightly longer delay before displaying a positive detection, but it does remain high for the duration of the attack. The delay between the initiation of the simulated GPS attack and the corresponding detection is a little longer than the others, at 13.354 seconds.

Lastly, the IMU simulated walk-off attack data is shown in Figure 6.8. There are three indicated attacks by the solid black line as the roll, pitch, and yaw walk-offs were tested individually. The walk-offs were conducted in the order of roll, pitch, and then

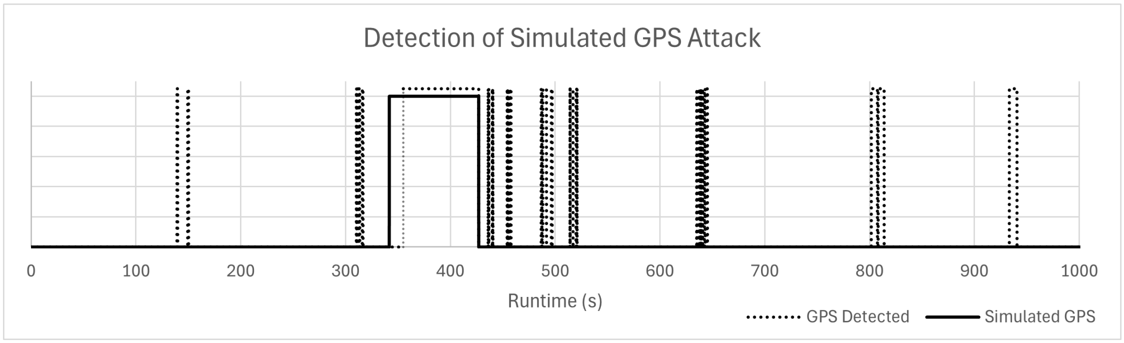


FIGURE 6.7: Simulated GPS Walk-off Attack and Functional Monitor Detections

yaw. Since the three axes are all part of the same sensor and therefore should be flagged as an error on the IMU sensor, the three walk-off attacks were graphed together. There are a few false positive IMU sensor error detection early on in the runtime, but once the roll walk-off was initiated, the detection algorithm jumped back and forth between high and low for a short time before settling out on an accurate positive detection with a delay of 25.45 seconds. Once the attack had taken place for a period of time, the detection settled out and no longer demonstrated any uncertainty.

After the roll attack was disabled, the detection determination returned to low until the pitch attack began. As expected, the detection method provided an accurate positive detection and remained so until this attack was disabled. The delay for the detection on the IMU for the pitch walk-off is 4.6 seconds. The last axis requiring simulated walk-off tested was the yaw axis. As seen towards the end of run-time, when the yaw attack was instituted, the corresponding detection algorithm provided the proper detection and remained in the detected state of high for the duration of the yaw walk-off before returning low once the attack was disabled. The delay of the yaw walk-off was recorded as the shortest at 2.85 seconds.

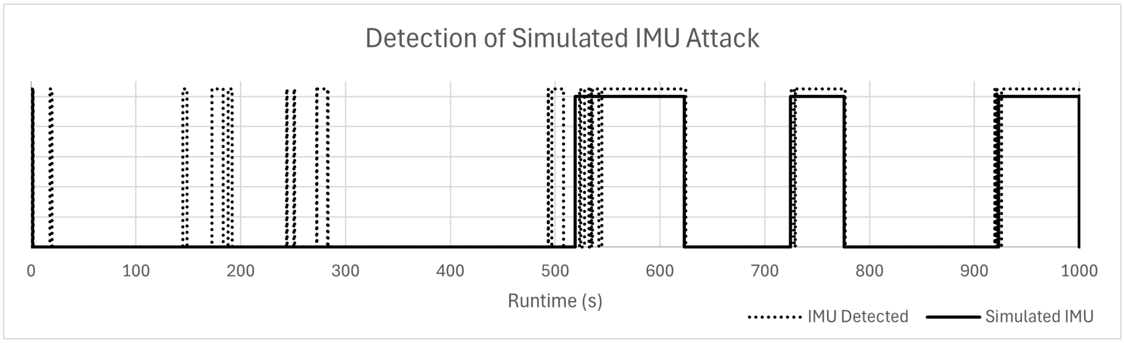


FIGURE 6.8: Simulated IMU Walk-off Attack and Functional Monitor Detections

Chapter 7

Conclusions and Future Work

Ensuring security of cyber-physical systems (CPS) is an ever-growing area of effort as more security results in a more trustworthy system. The combination of both an increased number of physical components and code complexity in typical CPS results in the assurance of security and proper functionality becoming more difficult, especially with a lack of known trustworthy components. This addition of more varied components, often consisting of common-off-the-shelf sensors, comes with the difficulty of an unverified supply chain. In an unverified supply chain, unauthorized access to the components can lead to the insertion of dubious or malicious firmware or hardware modifications.

The development of the functional monitor for the purposes of Run-Time Assurance presented in this dissertation was completed and tested on the cyber-security testbed architecture. The testbed layout consisted of the hardware-in-the-loop simulation and the embedded cyber-attack detector for implementation of specific components of the HECAD architecture as required. The HILS setup allowed for interfacing between a flight simulator and the VCU Aries FCS for the implementation of the simulated walk-off sensor attacks. The HECAD setup provided the platform for implementation of the functional monitoring architecture. The components of the functional monitoring architecture include the empirical models, decision metrics, and the detection algorithms to provide information to the end user about the status of various sensors used within the FCS.

The empirical models of the system were developed through inspection of various flight characteristics. The energy model was developed through analysis of the attributes of the input throttle energy to the flight dynamics of airspeed and change in altitude, then developing an equation to represent this relationship. The heading model from the computation of physical heading from magnetometer data for analysis of GPS accuracy. Lastly, the turn radius model was created from the airspeed versus change in indicated heading for computing how much the aircraft is physically turning.

Significant work was carried out on the decision metrics. This work included development of a metric for errors over time along with various statistical metrics including cumulative sum, exponentially weighted moving average, xbar and range control chats, and correlation coefficient. The detection methods also required examination for determining the proper thresholds and system operating bounds.

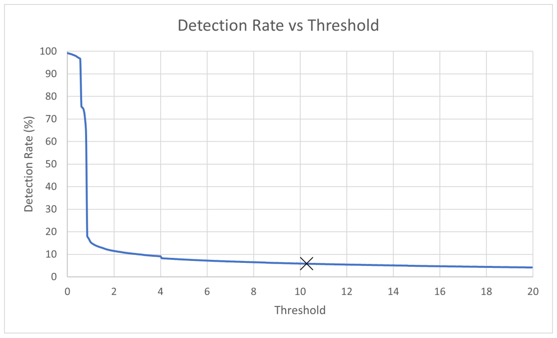
The results achieved provided clear distinctions among the decision metrics implemented. The threshold calculations yielded promising results of what constitutes an error in the sensor data and what does not when combined with the detection algorithm for flagging the error. When the full functional monitoring architecture was constructed, the observed detections when running each of the simulated attacks demonstrated the achievement of correct detections of the errors and determination of the sensors in which the walk-off attack was occurring.

It should be noted that the simulated walk-off attacks were used as a means of validation for the detection methods. The decision metrics are not based on the simulated attacks developed. For the creation of any cyber-attack detection method, the best possible detection was created while ignoring any specifics of the attacks. The overall development of the system boundaries and thresholds for the detection algorithm was done independent of the simulated attacks.

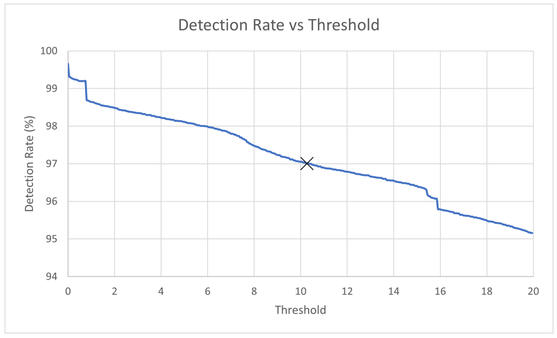
7.1 Future Work

A major goal of future work of the functional monitoring architecture would be to optimize the threshold and bounds determination used in the detection algorithm. The method introduced in this paper calculates a usable and repeatable value that provides accurate detections to keep the detection rate in the 90.0% percent range for positive detections and below 10.0% false positives. Optimization could improve both the false positive and accurate detection rate through more accurate determination of the threshold curve.

One technique to utilize for this process would be to conduct a sweep of threshold values from 0.0, which would consider all values out of range, to a value where no values are in range. The graphs of this sweep, as shown below in Figure 7.1, are an example of some preliminary results of this methodology. The "X" represents the calculated threshold from the detection algorithm. The results can be used to identify a more ideal threshold to enhance the detection rate. If the goal is to achieve a higher detection rate, this would increase the false positive detection. If the goal is for a lower false positive rate, the result would be a lower correct detection rate.



(A) Normal Flight



(B) Generic Simulated Attack

FIGURE 7.1: Detection Threshold Sweeps

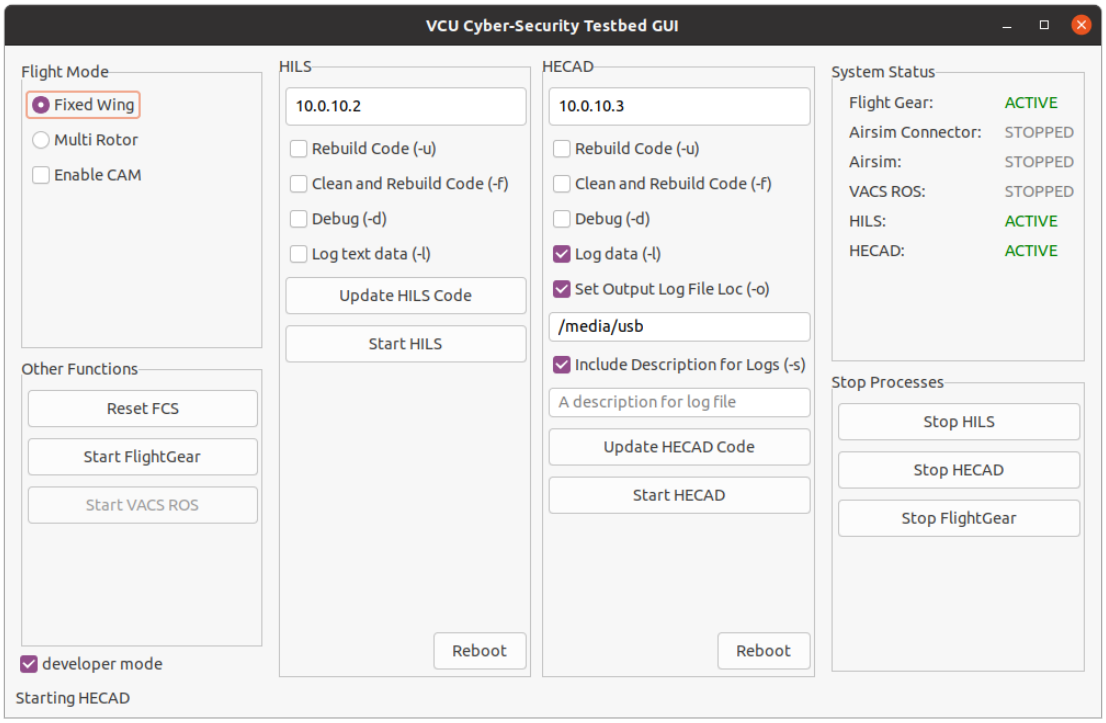
Another possibility of future work would be to incorporate actual data from sensors failing due to age, instead of from a cyber-attack or physical fault. A process for conducting accelerated aging on MEMS sensors along with an analysis of the failure characteristics that can be seen in the data from these sensors is introduced in [130]. This data could be used to develop detection mechanisms for sensor errors introduced by the normal sensor aging and wear-out process.

Appendix A

Improved User Experience for the Cyber-Security Testbed

Early on in the development of the software for both the HILS and HECAD systems, transferring the updated software to the FPGA then having to recompile the code each time and if there was an error, going back to fix it and starting the process over again began taking a significant amount of time away from development progress. To combat this added delay, a graphical user interface, GUI, was developed in Python using the wxPython package [131]. Overall, the implementation of the GUI significantly improved the usability of the cyber-security testbed and increased productivity. This GUI was also shared with other users of the testbed with the appropriate options available for improving and reducing the setup time of the process of utilizing the testbed.

The GUI development was an incremental process in order to debug the process of running scripts remotely over secure shell (ssh) and inside screen sessions. A series of radio buttons, check boxes, buttons, and text displays were used during the development process. The completed GUI is shown below in Figure A.1. In the upper left-hand corner, the Flight Mode box has options to switch the method of flight simulation between fixed-wing or multi-rotor, with certain buttons displaying for the appropriate purpose depending on the selection. Since for the purposes of the research within this document fixed-wing flight was used, the features will be described from the context of having fixed-wing be the selected flight mode. Next, is the Other Function box in the lower left-hand side. In this box are the options to start FlightGear along with the option to reset the FCS, which prior to the GUI was done by either physically removing and replacing power or pressing a button on the HECAD FPGA which required the HECAD software to be running.



Flight Mode: [✓ Fixed Wing] Multi Rotor
Enable CAM: [ ]
[✓] developer mode

HILS: 10.0.10.2
[ ] Rebuild Code (-u)
[ ] Clean and Rebuild Code (-f)
[ ] Debug (-d)
[ ] Log text data (-l)

HECAD: 10.0.10.3
[ ] Rebuild Code (-u)
[ ] Clean and Rebuild Code (-f)
[ ] Debug (-d)
[✓] Log data (-l)
[✓] Set Output Log File Loc (-o)
/media/usb
[✓] Include Description for Logs (-s)
A description for log file

Flight Gear:	ACTIVE
Airsim Connector:	STOPPED
Airsim:	STOPPED
VACS ROS:	STOPPED
HILS:	ACTIVE
HECAD:	ACTIVE

Starting HECAD

FIGURE A.1: GUI Screenshot

In the center of the GUI are the two main boxes for easier development on the HILS and HECAD FPGAs. At the top of each is a text box to input the IP address of the hardware. Next are a series of check boxes for rebuilding the code, conducting a clean and then full build from scratch of the software executable, enabling the GDB debugging flag, and enabling data logging. For the data logging on the HECAD, a separate USB device was used to minimize the number of read and write cycles on the main MicroSD card containing the embedded Linux image. This allowed for expandable storage for logging as the files can be fairly large depending on the logging rate. Since the logging location is removable, the option to change where the log file saves from the default location to a user specified location was also included. One issue that was found early on when testing the algorithms implemented on HECAD was that the only uniqueness of the files was the timestamp included in the filename which made remembering what was tested in each file difficult. The option to add a user specified description of the test run was added into the GUI and is stored as the first line of the log file.

Below the configurable options in the HILS and HECAD boxes are two buttons, one to update the code on the target device and the other to start the executable. The update code button performs a time sync between the host machine and target so that modification times of files are preserved when rebuilding code to save build time if the most recent version of a file has already been compiled. This button is only available if the developer mode check box in the bottom left-hand corner is selected. The start buttons take the configured options and passes them to a python script on the target machines over ssh as command line arguments, which are then processed by the python script and the program executable is run in a screen session. A unique feature of this python execution script is that if the screen session is already open, it preforms a ctrl+c on the current program running and then restarts the executable with the user selected options so that the user does not need to reattach to the screen session, which was a frustration early on in the GUI development. The option to reboot either target device was added to eliminate the need to log into the device and send a reboot command if required.

On the right-hand side are the system status and stop processes boxes. The system status was created by multi-threading the GUI process and another process to poll the status of each of the subsystems in use to alert the user if something was wrong or the code did not compile successfully by displaying the text of error for the status. The buttons within the stop processes box can be used to completely end any running processes, programs, or screens on either target device as labeled, or terminate the flight simulator.

Bibliography

[1] Katie Tarasov. *A first look at Amazon's new delivery drone, slated to start deliveries this year*. en. Nov. 2022. URL: https://www.cnbc.com/2022/11/11/a-first-look-at-amazons-new-delivery-drone.html.

[2] Eric Bangeman. *Amazon begins drone deliveries in California and Texas | Ars Technica*. Dec. 2022. URL: https://arstechnica.com/gadgets/2022/12/amazon-begins-drone-deliveries-in-california-and-texas/.

[3] Amazon Staff. *Amazon Prime Air prepares for drone deliveries*. en. Section: Transportation. June 2022. URL: https://www.aboutamazon.com/news/transportation/amazon-prime-air-prepares-for-drone-deliveries.

[4] W. Johnson and C. Silva. "NASA concept vehicles and the engineering of advanced air mobility aircraft". en. In: *The Aeronautical Journal* 126.1295 (Jan. 2022). Publisher: Cambridge University Press, pp. 59–91. ISSN: 0001-9240, 2059-6464. DOI: 10.1017/aer.2021.92. URL: https://www.cambridge.org/core/journals/aeronautical-journal/article/nasa-concept-vehicles-and-the-engineering-of-advanced-air-mobility-aircraft/AA7E668D759491B1889299819A2F2715.

[5] Isadora Garcia Ferrão et al. "Security and Safety Concerns in Air Taxis: A Systematic Literature Review". en. In: *Sensors* 22.18 (Jan. 2022). Number: 18 Publisher: Multidisciplinary Digital Publishing Institute, p. 6875. ISSN: 1424-8220. DOI: 10.3390/s22186875. URL: https://www.mdpi.com/1424-8220/22/18/6875.

[6] John Melton. *UAM Research at NASA*. 2018. URL: https://ntrs.nasa.gov/api/citations/20190028684/downloads/20190028684.pdf.

[7] Mahmood Shafiee et al. "Unmanned Aerial Drones for Inspection of Offshore Wind Turbines: A Mission-Critical Failure Analysis". en. In: *Robotics* 10.1 (Mar. 2021). Number: 1 Publisher: Multidisciplinary Digital Publishing Institute, p. 26. ISSN: 2218-6581. DOI: `10.3390/robotics10010026`. URL: `https://www.mdpi.com/2218-6581/10/1/26`.

[8] Junwon Seo, Luis Duque, and Jim Wacker. "Drone-enabled bridge inspection methodology and application". en. In: *Automation in Construction* 94 (Oct. 2018), pp. 112–126. ISSN: 0926-5805. DOI: `10.1016/j.autcon.2018.06.006`. URL: `https://www.sciencedirect.com/science/article/pii/S0926580517309755`.

[9] Andrew Fabian. "Reliable Navigation for SUAS in Complex Indoor Environments". In: *Theses and Dissertations* (Jan. 2020). DOI: `https://doi.org/10.25772/QBY2-P192`. URL: `https://scholarscompass.vcu.edu/etd/6484`.

[10] Ankush Desai et al. *SOTER: A Runtime Assurance Framework for Programming Safe Robotics Systems*. arXiv:1808.07921 [cs]. Apr. 2019. DOI: `10.48550/arXiv.1808.07921`. URL: `http://arxiv.org/abs/1808.07921`.

[11] Sebastian Schirmer et al. "A Hierarchy of Monitoring Properties for Autonomous Systems". In: 2023. DOI: `10.2514/6.2023-2588`.

[12] Hongjun Choi et al. "Detecting Attacks Against Robotic Vehicles: A Control Invariant Approach". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 801–816. ISBN: 978-1-4503-5693-0. DOI: `10.1145/3243734.3243752`. URL: `https://doi.org/10.1145/3243734.3243752`.

[13] Raul Quinonez et al. "SAVIOR: securing autonomous vehicles with robust physical invariants". In: *Proceedings of the 29th USENIX Conference on Security Symposium*. SEC'20. USA: USENIX Association, Aug. 2020, pp. 895–912. ISBN: 978-1-939133-17-5.

[14] Matthew Leccadito et al. "A survey on securing UAS cyber physical systems". In: *IEEE Aerospace and Electronic Systems Magazine* 33.10 (Oct. 2018). Conference

Name: IEEE Aerospace and Electronic Systems Magazine, pp. 22–32. ISSN: 1557-959X. DOI: 10.1109/MAES.2018.160145.

[15] Alan Kim et al. *Cyber Attack Vulnerabilities Analysis for Unmanned Aerial Vehicles | Infotech@Aerospace Conferences*. Sept. 2012. URL: https://arc.aiaa.org/doi/10.2514/6.2012-2438.

[16] Matthew Leccadito. "A Hierarchical Architectural Framework for Securing Unmanned Aerial Systems". In: *Theses and Dissertations* (Jan. 2017). DOI: https://doi.org/10.25772/0DK3-E418. URL: https://scholarscompass.vcu.edu/etd/5037.

[17] Justin G Fuller. "Run-Time Assurance: A Rising Technology". In: *2020 AIAA/IEEE 39th Digital Avionics Systems Conference (DASC)*. ISSN: 2155-7209. Oct. 2020, pp. 1–9. DOI: 10.1109/DASC50938.2020.9256425.

[18] Martin Leucker and Christian Schallhart. "A brief account of runtime verification". en. In: *The Journal of Logic and Algebraic Programming*. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07) 78.5 (May 2009), pp. 293–303. ISSN: 1567-8326. DOI: 10.1016/j.jlap.2008.08.004. URL: https://www.sciencedirect.com/science/article/pii/S1567832608000775.

[19] Kerianne L. Hobbs et al. "Runtime Assurance for Safety-Critical Systems: An Introduction to Safety Filtering Approaches for Complex Control Systems". In: *IEEE Control Systems Magazine* 43.2 (Apr. 2023). Conference Name: IEEE Control Systems Magazine, pp. 28–65. ISSN: 1941-000X. DOI: 10.1109/MCS.2023.3234380. URL: https://ieeexplore.ieee.org/document/10081233.

[20] Lui Sha, R. Rajkumar, and M. Gagliardi. "Evolving dependable real-time systems". In: *1996 IEEE Aerospace Applications Conference. Proceedings*. Vol. 1. Feb. 1996, 335–346 vol.1. DOI: 10.1109/AERO.1996.495894.

[21] Lui Sha. "Dependable system upgrade". In: *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*. ISSN: 1052-8725. Dec. 1998, pp. 440–448. DOI: 10.1109/REAL.1998.739777.

[22] Lui Sha. "Using simplicity to control complexity". In: *IEEE Software* 18.4 (July 2001). Conference Name: IEEE Software, pp. 20–28. ISSN: 1937-4194. DOI: 10.1109/MS.2001.936213.

[23] Tanya L. Crenshaw et al. "The Simplex Reference Model: Limiting Fault-Propagation Due to Unreliable Components in Cyber-Physical System Architectures". In: *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*. ISSN: 1052-8725. Dec. 2007, pp. 400–412. DOI: 10.1109/RTSS.2007.34.

[24] Stanley Bak et al. "The System-Level Simplex Architecture for Improved Real-Time Embedded System Safety". In: *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*. ISSN: 1545-3421. Apr. 2009, pp. 99–107. DOI: 10.1109/RTAS.2009.20.

[25] John D. Schierman et al. *Runtime Assurance Framework Development for Highly Adaptive Flight Control Systems:* en. Tech. rep. Fort Belvoir, VA: Defense Technical Information Center, Dec. 2015. DOI: 10.21236/AD1010277. URL: http://www.dtic.mil/docs/citations/AD1010277.

[26] Michael Aiello et al. "Run-Time Assurance for Advanced Flight-Critical Control Systems*". en. In: *AIAA Guidance, Navigation, and Control Conference*. Toronto, Ontario, Canada: American Institute of Aeronautics and Astronautics, Aug. 2010. ISBN: 978-1-60086-962-4. DOI: 10.2514/6.2010-8041. URL: https://arc.aiaa.org/doi/10.2514/6.2010-8041.

[27] Peter H. Feiler, David P. Gluch, and John J. Hudak. *The Architecture Analysis & Design Language (AADL): An Introduction:* en. Tech. rep. Fort Belvoir, VA: Defense Technical Information Center, Feb. 2006. DOI: 10.21236/ADA455842. URL: http://www.dtic.mil/docs/citations/ADA455842.

[28] S.E. Chodrow, F. Jahanian, and M. Donner. "Run-time monitoring of real-time systems". In: *[1991] Proceedings Twelfth Real-Time Systems Symposium*. Dec. 1991, pp. 74–83. DOI: `10.1109/REAL.1991.160360`.

[29] Brian Wheatman et al. "RADICS: Runtime Assurance of Distributed Intelligent Control Systems". In: *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. ISSN: 2325-6664. June 2021, pp. 182–187. DOI: `10.1109/DSN-W52860.2021.00038`.

[30] Vlada Dementyeva et al. "Runtime Assurance for Intelligent Cyber-Physical Systems". en. In: *2022 ACM/IEEE 13th International Conference on Cyber-Physical Systems (ICCPS)*. Milano, Italy: IEEE, May 2022, pp. 288–289. ISBN: 978-1-66540-967-4. DOI: `10.1109/ICCPS54341.2022.00035`. URL: `https://ieeexplore.ieee.org/document/9797627/`.

[31] Howard Barringer et al. "Rule-Based Runtime Verification". en. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Bernhard Steffen and Giorgio Levi. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 44–57. ISBN: 978-3-540-24622-0. DOI: `10.1007/978-3-540-24622-0_5`.

[32] Matthew Clark et al. *A Study on Run Time Assurance for Complex Cyber Physical Systems:* en. Tech. rep. Fort Belvoir, VA: Defense Technical Information Center, Apr. 2013. DOI: `10.21236/ADA585474`. URL: `http://www.dtic.mil/docs/citations/ADA585474`.

[33] Thomas Reps et al. "The use of program profiling for software maintenance with applications to the year 2000 problem". en. In: *Software Engineering — ESEC/FSE'97*. Ed. by Mehdi Jazayeri and Helmut Schauer. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1997, pp. 432–449. ISBN: 978-3-540-69592-9. DOI: `10.1007/3-540-63531-9_29`.

[34] Edmund M. Clarke. "Model checking". en. In: *Foundations of Software Technology and Theoretical Computer Science*. Ed. by S. Ramesh and G. Sivakumar. Lecture

Notes in Computer Science. Berlin, Heidelberg: Springer, 1997, pp. 54–56. ISBN: 978-3-540-69659-9. DOI: 10.1007/BFb0058022.

[35] Klaus Havelund and Grigore Roşu. “Efficient monitoring of safety properties”. en. In: *International Journal on Software Tools for Technology Transfer* 6.2 (Aug. 2004), pp. 158–173. ISSN: 1433-2787. DOI: 10.1007/s10009-003-0117-6. URL: https://doi.org/10.1007/s10009-003-0117-6.

[36] Bernd Finkbeiner, Sriram Sankaranarayanan, and Henny B. Sipma. “Collecting Statistics Over Runtime Executions”. en. In: *Formal Methods in System Design* 27.3 (Nov. 2005), pp. 253–274. ISSN: 1572-8102. DOI: 10.1007/s10703-005-3399-3. URL: https://doi.org/10.1007/s10703-005-3399-3.

[37] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. “Synthesizing Enforcement Monitors wrt. the Safety-Progress Classification of Properties”. en. In: *Information Systems Security*. Ed. by R. Sekar and Arun K. Pujari. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 41–55. ISBN: 978-3-540-89862-7. DOI: 10.1007/978-3-540-89862-7_3.

[38] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. “Runtime Verification of Safety-Progress Properties”. en. In: *Runtime Verification*. Ed. by Saddek Bensalem and Doron A. Peled. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, pp. 40–59. ISBN: 978-3-642-04694-0. DOI: 10.1007/978-3-642-04694-0_4.

[39] Yliès Falcone et al. “Runtime enforcement monitors: composition, synthesis, and enforcement abilities”. en. In: *Formal Methods in System Design* 38.3 (June 2011), pp. 223–262. ISSN: 1572-8102. DOI: 10.1007/s10703-011-0114-4. URL: https://doi.org/10.1007/s10703-011-0114-4.

[40] Yliès Falcone et al. “More testable properties”. English. In: *International Journal on Software Tools for Technology Transfer* 14.4 (Aug. 2012). Num Pages: 407-437 Place: Heidelberg, Netherlands Publisher: Springer Nature B.V., pp. 407–437.

ISSN: 14332779. DOI: `10.1007/s10009-011-0220-z`. URL: `https://www.proquest.com/docview/1024201993/abstract/A4B9AB4FBB154738PQ/1`.

[41] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. "What can you verify and enforce at runtime?" en. In: *International Journal on Software Tools for Technology Transfer* 14.3 (June 2012), pp. 349–382. ISSN: 1433-2787. DOI: `10.1007/s10009-011-0196-8`. URL: `https://doi.org/10.1007/s10009-011-0196-8`.

[42] Andreas Bauer, Martin Leucker, and Christian Schallhart. "Comparing LTL Semantics for Runtime Verification". In: *Journal of Logic and Computation* 20.3 (June 2010), pp. 651–674. ISSN: 0955-792X. DOI: `10.1093/logcom/exn075`. URL: `https://doi.org/10.1093/logcom/exn075`.

[43] Andreas Bauer, Martin Leucker, and Christian Schallhart. "Runtime Verification for LTL and TLTL". In: *ACM Transactions on Software Engineering and Methodology* 20.4 (Sept. 2011), 14:1–14:64. ISSN: 1049-331X. DOI: `10.1145/2000799.2000800`. URL: `https://dl.acm.org/doi/10.1145/2000799.2000800`.

[44] A. Pnueli and A. Zaks. "PSL Model Checking and Run-Time Verification Via Testers". en. In: *FM 2006: Formal Methods*. Ed. by Jayadev Misra, Tobias Nipkow, and Emil Sekerinski. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 573–586. ISBN: 978-3-540-37216-5. DOI: `10.1007/11813040_38`.

[45] Zhiwei Wang, Mohamed H. Zaki, and Sofiène Tahar. "Statistical runtime verification of analog and mixed signal designs". In: *2009 3rd International Conference on Signals, Circuits and Systems (SCS)*. Nov. 2009, pp. 1–6. DOI: `10.1109/ICSCS.2009.5412620`.

[46] D.K. Peters and D.L. Parnas. "Requirements-based monitors for real-time systems". In: *IEEE Transactions on Software Engineering* 28.2 (Feb. 2002). Conference Name: IEEE Transactions on Software Engineering, pp. 146–158. ISSN: 1939-3520. DOI: `10.1109/32.988496`.

[47] Stavros Tripakis. "A Combined On-Line/Off-Line Framework for Black-Box Fault Diagnosis". en. In: *Runtime Verification*. Ed. by Saddek Bensalem and

Doron A. Peled. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, pp. 152–167. ISBN: 978-3-642-04694-0. DOI: 10.1007/978-3-642-04694-0_11.

[48] A. Bauer, M. Leucker, and C. Schallhart. "Model-based runtime analysis of distributed reactive systems". In: *Australian Software Engineering Conference (ASWEC'06)*. ISSN: 2377-5408. Apr. 2006, 10 pp.–252. DOI: 10.1109/ASWEC.2006.36.

[49] M. Chen et al. "Failure diagnosis using decision trees". In: *International Conference on Autonomic Computing, 2004. Proceedings.* May 2004, pp. 36–43. DOI: 10.1109/ICAC.2004.1301345.

[50] R.C. Aitken. "Modeling the unmodelable: algorithmic fault diagnosis". In: *IEEE Design & Test of Computers* 14.3 (July 1997). Conference Name: IEEE Design & Test of Computers, pp. 98–103. ISSN: 1558-1918. DOI: 10.1109/54.606006.

[51] M. Sampath et al. "Diagnosability of discrete-event systems". In: *IEEE Transactions on Automatic Control* 40.9 (Sept. 1995). Conference Name: IEEE Transactions on Automatic Control, pp. 1555–1575. ISSN: 1558-2523. DOI: 10.1109/9.412626.

[52] Stavros Tripakis. "Fault Diagnosis for Timed Automata". en. In: *Formal Techniques in Real-Time and Fault-Tolerant Systems*. Ed. by Werner Damm and Ernst Rüdiger Olderog. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2002, pp. 205–221. ISBN: 978-3-540-45739-8. DOI: 10.1007/3-540-45739-9_14.

[53] John Schierman et al. *Run-Time Verification and Validation for Safety-Critical Flight Control Systems | Guidance, Navigation, and Control and Co-located Conferences*. Aug. 2008. URL: https://arc.aiaa.org/doi/10.2514/6.2008-6338.

[54] D. Seto et al. "Dynamic control system upgrade using the Simplex architecture". In: *IEEE Control Systems Magazine* 18.4 (Aug. 1998). Conference Name: IEEE Control Systems Magazine, pp. 72–80. ISSN: 1941-000X. DOI: 10.1109/37.710880.

[55] Christopher Lazarus, James G. Lopez, and Mykel J. Kochenderfer. *Runtime Safety Assurance Using Reinforcement Learning*. arXiv:2010.10618 [cs, eess]. Oct. 2020. DOI: `10.48550/arXiv.2010.10618`. URL: `http://arxiv.org/abs/2010.10618`.

[56] Xiaowan Huang et al. "Software monitoring with controllable overhead". en. In: *International Journal on Software Tools for Technology Transfer* 14.3 (June 2012), pp. 327–347. ISSN: 1433-2787. DOI: `10.1007/s10009-010-0184-4`. URL: `https://doi.org/10.1007/s10009-010-0184-4`.

[57] A.K. Mok and Guangtian Liu. "Efficient Run-time Monitoring Of Timing Constraints". In: *Proceedings Third IEEE Real-Time Technology and Applications Symposium*. June 1997, pp. 252–262. DOI: `10.1109/RTTAS.1997.601363`.

[58] Matthew B. Dwyer, Rahul Purandare, and Suzette Person. "Runtime Verification in Context: Can Optimizing Error Detection Improve Fault Diagnosis?" en. In: *Runtime Verification*. Ed. by Howard Barringer et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 36–50. ISBN: 978-3-642-16612-9. DOI: `10.1007/978-3-642-16612-9_4`.

[59] Richard L. Schwartz, P. M. Melliar-Smith, and Friedrich H. Vogt. "An interval logic for higher-level temporal reasoning". In: *Proceedings of the second annual ACM symposium on Principles of distributed computing*. PODC '83. New York, NY, USA: Association for Computing Machinery, Aug. 1983, pp. 173–186. ISBN: 978-0-89791-110-8. DOI: `10.1145/800221.806720`. URL: `https://dl.acm.org/doi/10.1145/800221.806720`.

[60] R. Razouk and M. Gorlick. "Real-time interval logic for reasoning about executions of real-time programs". In: *ACM SIGSOFT Software Engineering Notes* 14.8 (Nov. 1989), pp. 10–19. ISSN: 0163-5948. DOI: `10.1145/75309.75311`. URL: `https://dl.acm.org/doi/10.1145/75309.75311`.

[61] Klaus Havelund and Grigore Roşu. "Synthesizing Monitors for Safety Properties". en. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Joost-Pieter Katoen and Perdita Stevens. Lecture Notes in Computer

Science. Berlin, Heidelberg: Springer, 2002, pp. 342–356. ISBN: 978-3-540-46002-2. DOI: 10.1007/3-540-46002-0_24.

[62] Oded Maler, Dejan Nickovic, and Amir Pnueli. “Real Time Temporal Logic: Past, Present, Future”. en. In: *Formal Modeling and Analysis of Timed Systems*. Ed. by Paul Pettersson and Wang Yi. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, pp. 2–16. ISBN: 978-3-540-31616-9. DOI: 10.1007/11603009_2.

[63] Klaus Havelund and Grigore Roşu. “Monitoring Java Programs with Java PathExplorer”. en. In: *Electronic Notes in Theoretical Computer Science*. RV'2001, Runtime Verification (in connection with CAV '01) 55.2 (Oct. 2001), pp. 200–217. ISSN: 1571-0661. DOI: 10.1016/S1571-0661(04)00253-1. URL: https://www.sciencedirect.com/science/article/pii/S1571066104002531.

[64] K. Havelund and G. Rosu. “Monitoring programs using rewriting”. In: *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. ISSN: 1938-4300. Nov. 2001, pp. 135–143. DOI: 10.1109/ASE.2001.989799.

[65] D. Giannakopoulou and K. Havelund. “Automata-based verification of temporal properties on running programs”. In: *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. ISSN: 1938-4300. Nov. 2001, pp. 412–416. DOI: 10.1109/ASE.2001.989841.

[66] R. Gerth et al. *Simple On-the-fly Automatic Verification of Linear Temporal Logic | SpringerLink*. 1996. URL: https://link.springer.com/chapter/10.1007/978-0-387-34892-6_1.

[67] Koushik Sen and Grigore Roşu. “Generating Optimal Monitors for Extended Regular Expressions”. en. In: *Electronic Notes in Theoretical Computer Science*. RV '2003, Run-time Verification (Satellite Workshop of CAV '03) 89.2 (Oct. 2003), pp. 226–245. ISSN: 1571-0661. DOI: 10.1016/S1571-0661(04)81051-X. URL: https://www.sciencedirect.com/science/article/pii/S157106610481051X.

[68] Howard Barringer, David Rydeheard, and Klaus Havelund. "Rule Systems for Run-time Monitoring: from Eagle to RuleR". In: *Journal of Logic and Computation* 20.3 (June 2010), pp. 675–706. ISSN: 0955-792X. DOI: `10.1093/logcom/exn076`. URL: `https://doi.org/10.1093/logcom/exn076`.

[69] Hermann Kopetz and Paulo Veríssimo. "Real time and dependability concepts". In: *Distributed systems (2nd Ed.)* USA: ACM Press/Addison-Wesley Publishing Co., May 1993, pp. 411–446. ISBN: 978-0-201-62427-4.

[70] Farnam Jahanian, Ragunathan Rajkumar, and Sitaram C. V. Raju. "Runtime monitoring of timing constraints in distributed real-time systems". en. In: *Real-Time Systems* 7.3 (Nov. 1994), pp. 247–273. ISSN: 1573-1383. DOI: `10.1007/BF01088521`. URL: `https://doi.org/10.1007/BF01088521`.

[71] F. Jahanian and A. Goyal. "A formalism for monitoring real-time constraints at run-time". In: *[1990] Digest of Papers. Fault-Tolerant Computing: 20th International Symposium*. June 1990, pp. 148–155. DOI: `10.1109/FTCS.1990.89350`.

[72] Anik Momtaz. "Runtime Verification for Distributed Cyber-Physical Systems". In: *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*. ISSN: 2575-8462. Sept. 2021, pp. 349–350. DOI: `10.1109/SRDS53918.2021.00044`.

[73] Ankush Desai et al. *SOTER: A Runtime Assurance Framework for Programming Safe Robotics Systems*. en. Aug. 2018. URL: `https://arxiv.org/abs/1808.07921v3`.

[74] Smitha Gautham et al. "A multilevel cybersecurity and safety monitor for embedded cyber-physical systems: WIP abstract". In: *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*. ICCPS '19. New York, NY, USA: Association for Computing Machinery, Apr. 2019, pp. 320–321. ISBN: 978-1-4503-6285-6. DOI: `10.1145/3302509.3313321`. URL: `https://doi.org/10.1145/3302509.3313321`.

[75] Smitha Gautham, Athira V. Jayakumar, and Carl Elks. *Multilevel Runtime Security and Safety Monitoring for Cyber Physical Systems Using Model-Based Engineering*.

Sept. 2020. URL: https://link.springer.com/chapter/10.1007/978-3-030-55583-2_14.

[76] Alexander Weiss et al. “Understanding and Fixing Complex Faults in Embedded Cyberphysical Systems”. In: *Computer* 54.1 (Jan. 2021). Conference Name: Computer, pp. 49–60. ISSN: 1558-0814. DOI: 10.1109/MC.2020.3029975. URL: https://ieeexplore.ieee.org/abstract/document/9324891.

[77] Insup Lee et al. “Runtime Assurance Based On Formal Specifications”. en. In: *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)* 294 (July 1999). URL: https://repository.upenn.edu/handle/20.500.14332/6333.

[78] MoonZoo Kim et al. “Java-MaC: A Run-Time Assurance Approach for Java Programs”. en. In: *Formal Methods in System Design* 24.2 (Mar. 2004), pp. 129–155. ISSN: 1572-8102. DOI: 10.1023/B:FORM.0000017719.43755.7c. URL: https://doi.org/10.1023/B:FORM.0000017719.43755.7c.

[79] Usa Sammapun et al. “Statistical Runtime Checking of Probabilistic Properties”. en. In: *Runtime Verification*. Ed. by Oleg Sokolsky and Serdar Taşıran. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2007, pp. 164–175. ISBN: 978-3-540-77395-5. DOI: 10.1007/978-3-540-77395-5_14.

[80] U. Sammapun, Insup Lee, and O. Sokolsky. “RT-MaC: runtime monitoring and checking of quantitative and probabilistic properties”. In: *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA’05)*. ISSN: 2325-1301. Aug. 2005, pp. 147–153. DOI: 10.1109/RTCSA.2005.84.

[81] A. Prasad Sistla and Abhigna R. Srinivas. “Monitoring Temporal Properties of Stochastic Systems”. en. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Francesco Logozzo, Doron A. Peled, and Lenore D. Zuck. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 294–308. ISBN: 978-3-540-78163-9. DOI: 10.1007/978-3-540-78163-9_25.

[82] Cristina M. Wilcox and Brian C. Williams. “Runtime Verification of Stochastic, Faulty Systems”. en. In: *Runtime Verification*. Ed. by Howard Barringer et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 452–459. ISBN: 978-3-642-16612-9. DOI: `10.1007/978-3-642-16612-9_34`.

[83] Eric Bodden. *A lightweight LTL runtime verification tool for java | Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. 2004. URL: `https://dl-acm-org.proxy.library.vcu.edu/doi/abs/10.1145/1028664.1028776`.

[84] Patrick O’Neil Meredith et al. “An overview of the MOP runtime verification framework”. en. In: *International Journal on Software Tools for Technology Transfer* 14.3 (June 2012), pp. 249–289. ISSN: 1433-2787. DOI: `10.1007/s10009-011-0198-6`. URL: `https://doi.org/10.1007/s10009-011-0198-6`.

[85] Doron Drusinsky. “The Temporal Rover and the ATG Rover”. en. In: *SPIN Model Checking and Software Verification*. Ed. by Klaus Havelund, John Penix, and Willem Visser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2000, pp. 323–330. ISBN: 978-3-540-45297-3. DOI: `10.1007/10722468_19`.

[86] Klaus Havelund and Grigore Rosu. “Java PathExplorer - A Runtime Verification Tool”. In: (June 2001).

[87] Klaus Havelund and Grigore Roşu. “An Overview of the Runtime Verification Tool Java PathExplorer”. en. In: *Formal Methods in System Design* 24.2 (Mar. 2004), pp. 189–215. ISSN: 1572-8102. DOI: `10.1023/B:FORM.0000017721.39909.4b`. URL: `https://doi.org/10.1023/B:FORM.0000017721.39909.4b`.

[88] Doron Drusinsky. “Monitoring Temporal Rules Combined with Time Series”. en. In: *Computer Aided Verification*. Ed. by Warren A. Hunt and Fabio Somenzi. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2003, pp. 114–117. ISBN: 978-3-540-45069-6. DOI: `10.1007/978-3-540-45069-6_11`.

[89] Klaus Havelund. *Using Runtime Analysis to Guide Model Checking of Java Programs | SpringerLink*. 2000. URL: `https://link.springer.com/chapter/10.1007/10722468_15`.

[90] Feng Chen and Grigore Roşu. "Mop: an efficient and generic runtime verification framework". In: *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*. OOPSLA '07. New York, NY, USA: Association for Computing Machinery, Oct. 2007, pp. 569–588. ISBN: 978-1-59593-786-5. DOI: `10.1145/1297027.1297069`. URL: `https://dl.acm.org/doi/10.1145/1297027.1297069`.

[91] Rodolfo Pellizzoni et al. "Hardware Runtime Monitoring for Dependable COTS-Based Real-Time Embedded Systems". In: *2008 Real-Time Systems Symposium*. ISSN: 1052-8725. Nov. 2008, pp. 481–491. DOI: `10.1109/RTSS.2008.43`.

[92] Sebastian Schirmer et al. "A Hierarchy of Monitoring Properties for Autonomous Systems". In: Jan. 2023. DOI: `10.2514/6.2023-2588`.

[93] Johann Schumann et al. "Towards Real-time, On-board, Hardware-supported Sensor and Software Health Management for Unmanned Aerial Systems". en. In: *International Journal of Prognostics and Health Management* 6.1 (2015). Number: 1. ISSN: 2153-2648. DOI: `10.36001/ijphm.2015.v6i1.2243`. URL: `http://papers.phmsociety.org/index.php/ijphm/article/view/2243`.

[94] Sebastian Schirmer and Christoph Torens. "Safe Operation Monitoring for Specific Category Unmanned Aircraft". en. In: *Automated Low-Altitude Air Delivery: Towards Autonomous Cargo Transportation with Drones*. Ed. by Johann C. Dauer. Research Topics in Aerospace. Cham: Springer International Publishing, 2022, pp. 393–419. ISBN: 978-3-030-83144-8. DOI: `10.1007/978-3-030-83144-8_16`. URL: `https://doi.org/10.1007/978-3-030-83144-8_16`.

[95] Yashwanth Annpureddy et al. "S-TaLiRo: A Tool for Temporal Logic Falsification for Hybrid Systems". en. In: *Tools and Algorithms for the Construction and Analysis*

of Systems. Ed. by Parosh Aziz Abdulla and K. Rustan M. Leino. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 254–257. ISBN: 978-3-642-19835-9. DOI: 10.1007/978-3-642-19835-9_21.

[96] Kevin Yang. "Implementation of a Hierarchical Embedded Cyber-Attack Detection System in Unmanned Aerial Systems". In: *Theses and Dissertations* (Jan. 2020). DOI: https://doi.org/10.25772/E5KR-3N80. URL: https://scholarscompass.vcu.edu/etd/6309.

[97] Jeremy Price. "Implementation of a Hierarchical, Embedded, Cyber Attack Detection System for SPI Devices on Unmanned Aerial Systems". In: *Theses and Dissertations* (Jan. 2020). DOI: https://doi.org/10.25772/C6VB-KR73. URL: https://scholarscompass.vcu.edu/etd/6311.

[98] Kevin Yang et al. "Implementation of a Hierarchical Embedded Cyber Attack Detection system for sUAS Flight Control Systems". In: *AIAA Scitech 2021 Forum*. American Institute of Aeronautics and Astronautics, Jan. 2021. URL: https://arc.aiaa.org/doi/abs/10.2514/6.2021-0038.

[99] Nils Ole Tippenhauer et al. "On the requirements for successful GPS spoofing attacks". en. In: *Proceedings of the 18th ACM conference on Computer and communications security*. Chicago Illinois USA: ACM, Oct. 2011, pp. 75–86. ISBN: 978-1-4503-0948-6. DOI: 10.1145/2046707.2046719. URL: https://dl.acm.org/doi/10.1145/2046707.2046719.

[100] Cory Leahy. *Spoofing a Superyacht at Sea*. en-US. July 2013. URL: https://news.utexas.edu/2013/07/30/spoofing-a-superyacht-at-sea/.

[101] Todd E Humphreys et al. "Assessing the Spoofing Threat: Development of a Portable GPS Civilian Spoofer". en. In: (2008).

[102] Jon Warner and Roger Johnston. "A Simple Demonstration that the Global Positioning System (GPS) is Vulnerable to Spoofing". In: *The Journal of Security Administration* (2012).

[103] Matthew W. Gelber, Peter Vaughan Truslow, and Robert H. Klenke. "Testbed for Cyber Attack Detection of Flight Control System Using Zynq FPGA+ARM SOCs". In: *AIAA SCITECH 2023 Forum*. AIAA SciTech Forum. American Institute of Aeronautics and Astronautics, Jan. 2023. DOI: 10.2514/6.2023-1342. URL: https://arc.aiaa.org/doi/10.2514/6.2023-1342.

[104] Joel Elmore. "Design of an All-In-One Embedded Flight Control System". In: *Theses and Dissertations* (Jan. 2015). DOI: https://doi.org/10.25772/63KT-S314. URL: https://scholarscompass.vcu.edu/etd/3981.

[105] Garrett Ward. "Design of a Small Form-Factor Flight Control System". en. In: (2014).

[106] Tim Bakker, Matthew T. Leccadito, and Robert H. Klenke. "Flexible FPGA based Hardware In the Loop Simulator for Control, Fault-Tolerant and Cyber-Physical Systems". In: *55th AIAA Aerospace Sciences Meeting*. AIAA SciTech Forum. American Institute of Aeronautics and Astronautics, Jan. 2017. DOI: 10.2514/6.2017-0549. URL: https://arc.aiaa.org/doi/10.2514/6.2017-0549.

[107] Wiki FlightGear. *Property Tree/Sockets - FlightGear wiki*. Sept. 2022. URL: https://wiki.flightgear.org/Property_Tree/Sockets.

[108] Nancy Leveson and John Thomas. "STPA Handbook". In: (2018).

[109] Carl Elks et al. "Pervasive Runtime Monitoring for Detection and Assessment of Emerging Hazards for Advanced UAM Systems". In: *AIAA AVIATION 2022 Forum*. AIAA AVIATION Forum. American Institute of Aeronautics and Astronautics, June 2022. DOI: 10.2514/6.2022-3541. URL: https://arc.aiaa.org/doi/10.2514/6.2022-3541.

[110] Smitha Gautham et al. *STPA-Driven Multilevel Runtime Monitoring for In-Time Hazard Detection*. Aug. 2022. URL: https://link.springer.com/chapter/10.1007/978-3-031-14835-4_11.

[111] Mohiuddin Ahmed and Al-Sakib Khan Pathan. "False data injection attack (FDIA): an overview and new metrics for fair evaluation of its countermeasure".

In: *Complex Adaptive Systems Modeling* 8.1 (Apr. 2020), p. 4. ISSN: 2194-3206. DOI: 10.1186/s40294-020-00070-w. URL: https://doi.org/10.1186/s40294-020-00070-w.

[112] Brajagopal Tripathi. *What is repudiation in cyber security?* en. Jan. 2023. URL: https://medium.com/@brajagopal.tripathi/what-is-repudiation-in-cyber-security-2eb98a75510d.

[113] William Rose. "Rotation Matrices". en. In: (2015). URL: https://www1.udel.edu/biology/rosewc/kaap686/notes/matrices_rotations.pdf.

[114] Yuejiu Zheng et al. "A capacity prediction framework for lithium-ion batteries using fusion prediction of empirical model and data-driven method". In: *Energy* 237 (Dec. 2021). ISSN: 0360-5442. DOI: 10.1016/j.energy.2021.121556. URL: https://www.sciencedirect.com/science/article/pii/S0360544221018041.

[115] community SciPy. *SciPy v1.11.4 Manual, scipy.optimize.curve_fit*. 2023. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html.

[116] Will Kenton. *Least Squares Method: What It Means, How to Use It, With Examples*. en. 2023. URL: https://www.investopedia.com/terms/l/least-squares-method.asp.

[117] Mathworks. *Nonlinear Least Squares (Curve Fitting) - MATLAB & Simulink*. URL: https://www.mathworks.com/help/optim/nonlinear-least-squares-curve-fitting.html.

[118] VelocityBox. *Turn Radius*. URL: https://www.vboxautomotive.co.uk/downloads/Calculating%20Radius%20of%20Turn%20from%20Yaw.pdf.

[119] Matthew W. Gelber et al. "Functional Monitor Models and Detection Methods for Sensor Data Variance of a Real-Time Cyber-Physical System". In: *AIAA SCITECH 2024 Forum*. AIAA SciTech Forum. American Institute of Aeronautics and Astronautics, Jan. 2024. DOI: 10.2514/6.2024-0761. URL: https://arc.aiaa.org/doi/10.2514/6.2024-0761.

[120] A.J. Ferrer-Riquelme. *Cumulative Sum - an overview | ScienceDirect Topics*. 2009. URL: `https://www.sciencedirect.com/topics/mathematics/cumulative-sum`.

[121] T. Kourti. *Cumulative Sum Control Chart - an overview | ScienceDirect Topics*. 2009. URL: `https://www.sciencedirect.com/topics/mathematics/cumulative-sum-control-chart`.

[122] NIST. *EWMA Control Charts*. 2012. URL: `https://doi.org/10.18434/M32189`.

[123] Ted Hessing. *I-MR Chart*. 2014. URL: `https://sixsigmastudyguide.com/i-mr-chart/`.

[124] Ted Hessing. *X Bar R Control Charts*. 2014. URL: `https://sixsigmastudyguide.com/x-bar-r-control-charts/`.

[125] Ted Hessing. *X Bar S Control Chart*. 2014. URL: `https://sixsigmastudyguide.com/x-bar-s-chart/`.

[126] Jason Fernando. *The Correlation Coefficient*. en. Oct. 2023. URL: `https://www.investopedia.com/terms/c/correlationcoefficient.asp`.

[127] Sebastian Taylor. *Covariance*. en-US. Nov. 2023. URL: `https://corporatefinanceinstitute.com/resources/data-science/covariance/`.

[128] P.A. Mackowiak, S.S. Wasserman, and M.M. Levine. *Confidence Intervals*. 1998. URL: `http://www.stat.yale.edu/Courses/1997-98/101/confint.htm`.

[129] Mark Lewis. *Prediction Interval | Overview, Formula & Calculations*. en. Nov. 2023. URL: `https://study.com/WEB-INF/views/jsp/redesign/academy/lesson/seoLessonPage.jsp`.

[130] Matthew W. Gelber, Peter Vaughan Truslow, and Robert H. Klenke. "Testbed to Characterize MEMS Sensor Degradation During Extended Operation". In: *AIAA SCITECH 2024 Forum*. AIAA SciTech Forum. American Institute of Aeronautics and Astronautics, Jan. 2024. DOI: `10.2514/6.2024-2353`. URL: `https://arc.aiaa.org/doi/10.2514/6.2024-2353`.

[131] Team wxPython. *Welcome to wxPython!* en. Aug. 2021. URL: `https://wxpython.org/index.html`.