SLIVer: Simulation-Based Logic Bomb Identification/Verification for Unmanned Aerial Vehicles

THESIS

Jake M. Magness, Second Lieutenant, USAF

AFIT-ENG-MS-20-M-039

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-20-M-039

SLIVer: Simulation-Based Logic Bomb Identification/Verification for Unmanned Aerial Vehicles

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Engineering

Jake M. Magness, B.S.C.E.

Second Lieutenant, USAF

March 26, 2020

AFIT-ENG-MS-20-M-039

SLIVer: Simulation-Based Logic Bomb Identification/Verification for Unmanned Aerial Vehicles

THESIS

Jake M. Magness, B.S.C.E.
Second Lieutenant, USAF

Committee Membership:

Lieutenant Colonel Patrick J. Sweeney, Ph.D
Chair

Scott R. Graham, Ph.D
Member

Nicholas S. Kovach, Ph.D
Member

Abstract

This research introduces SLIVer, a Simulation-based Logic Bomb Identification/Verification methodology, for finding logic bombs hidden within Unmanned Aerial Vehicle (UAV) autopilot code without having access to the device source code. Effectiveness is demonstrated by executing a series of test missions within a high-fidelity software-in-the-loop (SITL) simulator. In the event that a logic bomb is not detected, this methodology defines safe operating areas for UAVs to ensure to a high degree of confidence the UAV operates normally on the defined flight plan.

SLIVer uses preplanned flight paths as the baseline input space, greatly reducing the input space that must be searched to have confidence that the UAV will not encounter a logic bomb trigger condition during its mission. This research discusses the process for creating a logic bomb in the ArduPilot autopilot software, creating test flight profiles, UAV log file parsing, and the analysis of the methodology. SLIVer can accommodate multiple flight profiles and parses through the corresponding log files to create a safety corridor through which the UAV is able to safely traverse through with a desired level of confidence. By utilizing SLIVer, UAV operators and planners alike are afforded increased confidence that the aircraft will operate normally throughout the duration of a mission. The proof of concept implementation shows that the input space required to validate a UAV mission is reduced by approximately 60%, a far better result than brute force input testing. As UAVs are continually called upon to fill critical civilian and military roles, it is essential that planners and users of these devices have a methodology in place to assure that logic bombs are absent from the device.

Table of Contents

Page

List of Figures

SLIVer: Simulation-Based Logic Bomb Identification/Verification for Unmanned Aerial Vehicles

I. Introduction

1.1 Background and Motivation

Improving capabilities of Unmanned Aerial Vehicles (UAVs) allow them to fulfill tasks of critical importance for civilian and military applications alike. UAVs used for climate monitoring, forest fire monitoring, law enforcement, agriculture, and various military applications such as reconnaissance, surveillance, and carrying out attacks [1, 2]. The utilization of these devices is transforming the world as UAVs automate tasks that were once accomplished by humans. Current predictions show that the number of UAVs in use will triple by 2023, emphasizing the importance of securing them from malicious activity. With current predictions stating that the number of UAVs in use will triple by 2023, it is essential that these devices are secure from adversaries [3].

There are many documented studies [4, 5, 6, 7, 8, 9, 10, 11] demonstrating the lack of security present in today's commercially available UAVs. Not only have vulnerabilities been found on these devices but there are also documented cases where corporations are planting spying software, also known as spyware, on UAVs that store flight data on corporate servers [4, 5]. Given these glaring security issues with UAVs, serious consideration must be made before deploying these devices in critical roles.

This research seeks to address a specific and previously undocumented potential security vulnerability present in UAVs: logic bomb implantation within UAV autopi-

lot code. For casual users and hobbyists, the introduction of unknown code may not be of a huge concern. However, when these UAVs are utilized by entities that are responsible for tasks such as critical infrastructure monitoring, or for military applications, sensitive data and human lives are now at risk. For this reason, it is essential that UAVs are free of logic bombs.

1.2 Problem Statement

UAVs are becoming essential tools to aid in critical civilian and military operations. As these devices see more use, especially to carry out sensitive tasks, the motivation for adversaries to exploit them grows. Unfortunately, UAVs are largely insecure and are plagued with documented vulnerabilities making them easy targets for an adversary to exploit [6, 7, 8]. While many of these exploits stem from poor device management, dated exploits, and protocol issues, there are still exploits that have yet to be addressed [9, 10, 11]. One such avenue of exploitation is logic bombs planted within UAV autopilot software. It does not appear that there are documented efforts to study or defend against this type of attack. This research demonstrates the potential consequences of having a logic bomb within autopilot code and the importance of identifying it before a mission occurs.

As UAVs automate tasks of increasing complexity, many of these devices will be commercially purchased rather than created in-house by corporations or entities. When examining logic bombs implanted within UAV code, utilizing commercially purchased products presents a problem: it is unlikely that the user will have access to the source code. Without access to the source code, the task of searching through code for logic bombs is made far more difficult. Given the growing importance of UAV roles and the potential for the new threat of logic bombs hidden within UAV autopilot code, a methodology that efficiently identifies logic bombs in untrusted code

would be very valuable.

1.3 Research Objectives

This research lays out a logic bomb detection methodology for UAVs and other aerial vehicles. The methodology combines a SITL simulator with statistical flight path analysis to reduce the potential search space required to discover logic bomb trigger conditions. The research objectives for this work are as follows:

- Thoroughly understand UAV operation and previously documented vulnerabilities.
- Understand how logic bombs have been implemented in the past and how those approaches apply to UAV autopilot exploitation.
- Identify previously-developed logic bomb detection techniques and attempt to apply them to the UAV logic bomb detection problem.
- Develop a logic bomb detection methodology for UAV autopilot code.
- Find a UAV autopilot code base that is both open-source and can be simulated.
- Create a test bed that supports a high-fidelity UAV simulation with modified autopilot code.
- Develop a UAV logic bomb triggerable from multiple flight conditions.
- Demonstrate the effectiveness of the logic bomb within the simulation tool.
- Demonstrate that the methodology successfully identifies logic bombs that would affect a planned mission.

The questions that this research seeks to answer are as follows:

- Can a logic bomb be developed in the ArduPilot autopilot software?
- Can this logic bomb be represented in the ArduPilot Software in the Loop simulation tool?
- Is it possible to detect the presence of logic bombs in autopilot code without access to the source code?
- How can a methodology be developed to reduce the risk of logic bomb trigger events without brute forcing all potential trigger conditions in a UAV operating area?

1.4 Hypothesis

The hypothesis for this research is as follows: A methodology can be developed that efficiently tests UAV autopilot code for the presence of logic bombs and validates UAV flight paths before mission execution that significantly improves on brute force approaches.

1.5 Approach

The approach for developing a logic bomb detection methodology is best broken down by defining a few key checkpoints in the methodology development. The first of these steps is to find a suitable autopilot code base that has simulation testing capabilities and grants the user the ability to modify the source code. With the autopilot code base selected, the next step of this research is to create a logic bomb within the autopilot source code. Effectiveness of the logic bomb is evaluated by running the compromised autopilot code in the simulation software and analyzing the flight behavior. The next step is to develop an analysis script that parses the UAV log files into only the entries that this research is concerned with. In addition

to parsing the log files, the analysis script creates a confidence interval of expected locations that the UAV could occupy for the duration of the flight path. Next is the creation of various missions with increasing levels of complexity to exercise the UAV. These missions seek to examine the robustness of the methodology and create a quantitative metric for the effectiveness of the methodology.

From the outlined checkpoints emerges a methodology that can be replicated to evaluate the security of UAV autopilot code. While this methodology won't provide the tester the assurance that no logic bombs exist within the autopilot code entirely, it quickly provides the assurance that a specific mission can be executed with a low probability of triggering specific types of logic bombs. The developed methodology serves as a framework with various other additions outlined through this paper to make it robust and dynamic.

1.6 Contributions

The contributions of this research to the field of UAV security are listed below:

- **Bringing to Light the Importance of UAV Logic Bomb Threats**

  Logic bombs have largely been ignored as a potential attack vector for UAVs. This thesis illustrates the viability and the potential implications of a logic bomb UAV attack.

- **Creation of UAV Autopilot Logic Bomb**

  To date, no publicly available documentation is found on a logic bomb attack to any commercially available UAV autopilot software. This thesis produced a first-of-its-kind attack to the ArduPilot autopilot software.

- **Development of a Logic Bomb Detection Methodology Without Requirement for Source Code**

The research performed developed a logic bomb detection methodology that uses simulation based analysis techniques to find the presence of logic bombs in ArduPilot autopilot code.

- **UAV Log File Analysis Script Development Tailored to Aid in UAV Logic Bomb Detection**

  An analysis script was developed that utilizes ArduPilot log files to aid with simulation based testing for logic bomb detection. This analysis script can examine any logged parameters to help define potential logic bomb input vectors that must be tested.

- **Methodology Analysis**

  This methodology was analyzed by measuring the search space required to validate a given flight path in comparison to the brute force search area. It demonstrated the search space is drastically reduced by using the flight path with its measured level of deviation from the planned course as opposed to searching an entire area of operation via brute force.

1.7 Organization

This thesis is organized into four remaining chapters.

- Chapter II outlines the background topics to illustrate the current academic conversation relating to this research. Background information is discussed regarding UAVs, logic bombs, black box testing and the ArduPilot UAV autopilot code and simulation tools. These topics support the development of a UAV autopilot logic bomb detection methodology.

- Chapter III introduces the Simulation-based Logic bomb Identification/Verification (SLIVer) logic bomb detection methodology. It then presents a proof

of concept experiment including assumptions, variables, and factors that are accounted for throughout the experimentation phase of this research. System and testing considerations are discussed throughout this section with the associated explanations as to experimental choices. The testbed that is utilized is discussed in depth as well as the steps taken to complete the setup procedures for experiment. The different test missions are described that were utilized to evaluate SLIVer.

- Chapter IV discusses the results obtained from the proof of concept experiments. These results are analyzed and quantifiable measurements are taken to evaluate the effectiveness of SLIVer. The results obtained from the experiment are compared against the current best methodology of logic bomb detection in UAV autopilot code: brute force.

- Chapter V summarizes the results of the experiment and provides insight as to what these results mean for the future of the field. This chapter discusses the implications of these results and demonstrates how the research goals were accomplished by this experiment. Finally, this chapter goes on to describe future works projects to be accomplished.

II. Background and Literature Review

2.1 UAV Applications

Experimentation with UAVs began as early as 1918 with Charles Kettering's gyroscope-controlled flying machine that exploded after a set amount of propeller rotations [12]. Since Kettering's first experiments, continual experiments were conducted with pilot aircraft by the military with the prevalent use being radio-controlled target drones. Further applications of unmanned flying vehicles were limited by technical capabilities and it wasn't until the 1970's that UAVs took the form that we associate with as modern UAVs. These Vietnam Era UAVs would be considered rudimentary by today's standards but they accomplished reconnaissance missions with minimal losses and no risk to a human pilot. Vietnam era UAV systems were based off of cruise missile technology and these devices were often referred to as drones with the attitude being they were simple and easily replaceable. While early iterations of UAVs were very limited in their mission set, their advantages were recognized and development continued. The miniaturization and increase in performance of embedded hardware has greatly increased the capabilities of modern UAVs and their role for military and civilian applications continue to grow today.

Modern UAVs are far more capable than their predecessors thanks to the development of powerful and energy efficient computational options which are able to handle processing high-resolution video with high frame rates [13]. Further advancement in wireless technology has made it possible to deliver high-fidelity, real-time video and other key pieces of information from ranges up to 5000km. Additionally, machine learning algorithms and techniques have given UAVs the ability to autonomously carry out complicated mission sets. With the various advancements made to integrated circuit technology, wireless communications techniques and technology as well

as increased automation through machine learning, there are now a wide variety of missions that UAVs can carry out for both civilian and military applications.

In the United States, civilian use of UAVs is skyrocketing with the FAA estimating that by 2020 the United States will have over 30,000 UAVs operating [7]. The increasing usage of UAVs can be linked to the rapid improvement in UAV capabilities which make them an attractive option to solve many modern problems such as helping fight forest fires, infrastructure monitoring, law enforcement, and smart city monitoring [2, 10, 14, 15]. UAVs will soon be able to automate tasks that previously required manned solutions. With numerous new roles that UAVs are filling, they will quickly become an integral part of modern society. As our reliance grows on the capabilities of UAVs, the security, reliability and availability of these devices is paramount for their continued use on a large scale.

UAVs have become an essential tool for modern militaries around the world. Goldman Sachs estimates that military spending on UAVs will reach 70 billion dollars this year due to their affordability and increasing capabilities [15]. UAVs have seen increasing popularity among militaries primarily because UAVs are relatively inexpensive, allow dangerous jobs to be completed without putting a human life in danger, and have the ability to automate tasks that previously required human intervention. UAVs are now performing a wide array of missions but primarily they are used for surveillance, target acquisition, and reconnaissance capabilities [16]. The United States Air Force has classified UAVs into 5 separate groups by maximum gross takeoff weight and a variety of platforms fit within each group, ranging from micro UAVs utilized by ground forces to strategic level aircraft that gather intelligence from over 18,000 feet [17]. These different groups of drones impact the battlefield on a tactical, operational and strategic level providing key information and intelligence to mission planners and operators alike. The growing capabilities and increased investment in

UAV technology indicates that UAVs will see increasing use in militaries around the world.

Increasing investment in UAV technology guarantees that the number of UAV applications will increase. As UAVs take on more tasks, the need for UAV reliability and integrity increases. Multiple researchers have demonstrated how vulnerable UAVs are to both cyber threats as well as physical threats [9, 10, 4, 5, 18]. While techniques to mitigate many of these threats are being developed, one area that hasn't been addressed is the potential for malicious code to be hidden in the software of UAVs. Hidden software in UAVs/base stations has been found in UAVs purchased by the United States Army and the Department of Homeland Security. The impact of hidden malware triggered in UAVs operating in key capacities could prove to be devastating and a method to detect the presence of embedded malicious code must be developed.

2.2 UAV Vulnerabilities

UAV capability and performance has grown considerably since their humble beginning as reconnaissance drones. The availability of powerful integrated circuits and sensor arrays allow UAVs to perform new and increasingly complicated mission sets. However, the introduction of more advanced computer architectures to UAVs has also created some problems that haven't yet been fully addressed. Most notably, many UAVs suffer from a wide array of security vulnerabilities [6, 7]. These vulnerabilities vary greatly as exploits have been found in the command and control systems implemented to communicate with the UAV, the actual operating systems that run on the UAVs, and hidden malware on found on base stations to name a few [4, 5].

In order to discuss UAV vulnerabilities, it is important to first understand the UAV and ground station architecture as shown in Figure 1.

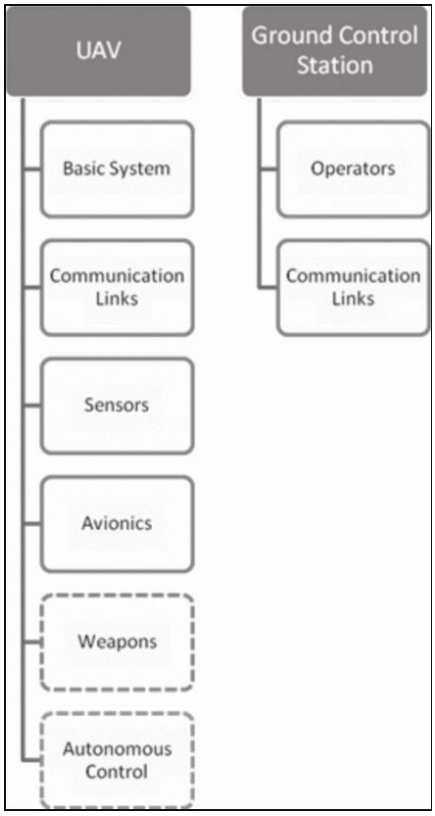

Figure 1: Potential Attack Vectors: Potential vulnerable systems that can be attacked on a UAV [19]

The UAV consists of multiple systems that are responsible for carrying out the mission set. Every UAV has a basic system, communications link, sensors and avionics systems while some may also have a weapons system and autonomous control system depending on the particular UAV, its configuration, and the UAV mission. The basic system is akin to an operating system for the UAV but it is primarily focused on UAV flight operation rather than as a user interface tool [19]. The communication links system is responsible for maintaining communication with the ground station to send information and to receive input from an operator. The sensors system is responsible for managing the array of sensors on the UAV that are used to navigate and gather

information. The sensor suite is relatively simple for UAVs used for recreation purposes but for high-grade military UAVs, it can be extremely advanced with multiple sensors. The final inherent system on a UAV is the avionics system that is comprised of both hardware and software components that allow the UAV to maintain controlled flight. Weapons systems are present on select military UAVs. These systems are responsible for controlling target acquisition, arming, and launching munitions. Autonomous control systems are present on UAVs that are tasked with performing tasks without the requirement for operator intervention. These types of tasks can include but are not limited to package delivery, automated survey, or transit to another location.

The ground control station (GCS) is responsible for maintaining a communications link to the UAV to receive information as well as to allow the operator to send commands to the UAV. In the case of autonomous UAVs, the ground station may not send commands, however communications links with the ground station are almost always maintained to receive sensor information as well as to provide input if necessary. The ground station hardware varies in complexity ranging from a mobile phone to an advanced cockpit like those used to control military grade UAVs. With the differing ground station hardware, it can be very difficult to secure these devices and documented attacks have been carried out as demonstrated by the DJI ground station software sending information back to their home country [4, 5]. The ground station software that is utilized to communicate with the DJI drones was suspected of "providing U.S. critical infrastructure and law enforcement data to the Chinese government" [4]. While the drones were completing their assigned mission, information on these key sites was being gathered. Once completing the flight, the base station would automatically upload the data back to the company. While this feature was later able to be disabled, it is encouraged by the company to upload flight logs

to their servers. This vulnerability demonstrates that companies are willfully selling UAVs that gather and upload critical information back to their home country without making it clear to the user that these actions are being performed.

Figure 2 shows different attack vectors on UAV and ground station systems. While some of the names for the systems have changed, all the previously mentioned systems are still shown in the figure. Through these attack vectors, vulnerabilities have been discovered on a wide variety of consumer off the shelf UAVs as well as incidents with United States Military UAVs [9, 10, 18, 6, 7, 4, 5, 11, 19, 20]. The most prevalent threats are those to the communications links and to sensor arrays. Many of the exploits to the communications links are carried out through traditional network attacks due to the UAV operating via unencrypted communications channels, through insecure wireless protocols via de-authentication attacks, ARP-cache poisoning, and GPS spoofing. Further vulnerabilities have been documented in some devices through the default FTP and Telnet root accounts being unsecured with no password protection, allowing a potential adversary to gain a root shell on the device.

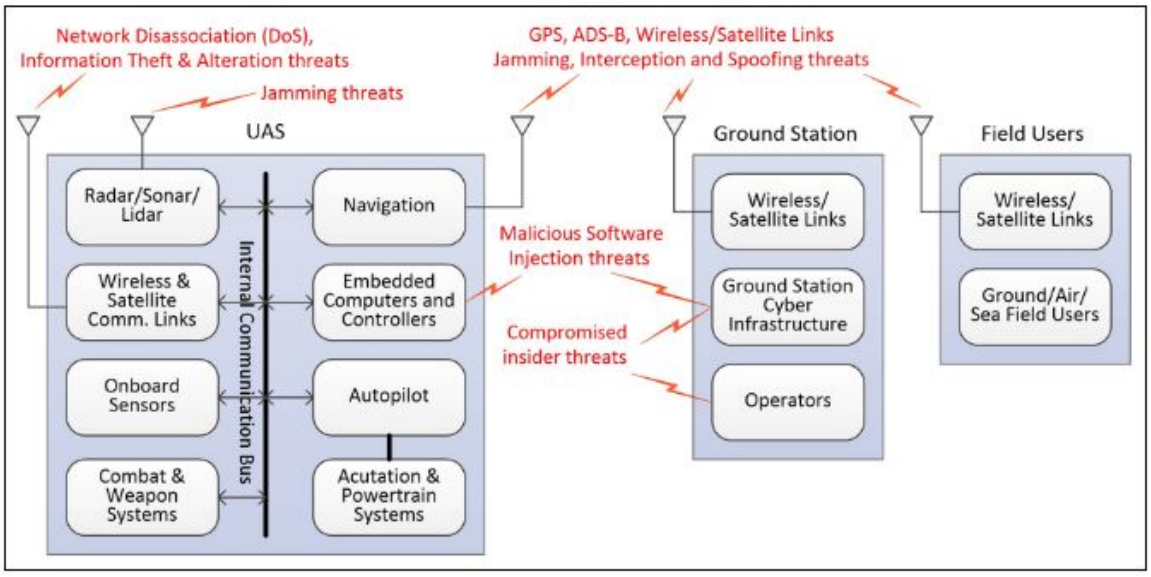

Figure 2: Potential Attack Vectors: Potential vulnerable systems that can be attacked on a UAV [6]

The previous listed vulnerabilities and the attack vectors in Figure 2 aren't all-inclusive but they do show the general trend in UAV security research to date. UAVs have been demonstrated to be vulnerable to simple network attacks that take advantage of insecure communications protocols and poor security practices. UAVs have also been demonstrated to be vulnerable to jamming attacks and GPS spoofing [19]. While these vulnerabilities are extremely important to address, little discussion has been focused on the embedded software on the UAV. Specifically, looking at logic bombs, an implementation of hidden malware, that can appear in UAV software. As with most proprietary software, the source code for many commercial UAVs is not available to the consumer. With users unable to view the code, they are unable to validate that the code executing on their UAV is malware-free.

As UAVs continue to perform more critical societal and military tasks, the threat of exploitation grows. Performing an exploit on a UAV that is responsible for tactical military reconnaissance could put the lives of service men and women in danger. Exploited UAVs that are tasked with monitoring the health of a farmer's crops can potentially ruin the entire harvest costing millions of dollars. Package delivery drones can be hijacked and their payload stolen by criminals. Many exploits that have been performed on UAVs are relatively simple and can be performed by almost any layperson. These threats have been studied but investigation into the detection of logic bombs hidden on a UAV has received little attention. With UAVs performing missions with increasing responsibility and autonomy, it is critically important that the threat of malicious embedded software is also addressed. A methodology to efficiently detect logic bombs is a key component to mitigating this threat.

2.3 Logic Bombs

This section talks in greater depth of logic bomb detection in general and the problems with logic bomb detection in UAV systems. The authors of [21] define a logic bomb as "...malicious application logic that is executed, or triggered, only under certain (often narrow) circumstances". Early iterations of logic bombs were arguably legitimate, designed to render software unusable after license agreements had expired. However, there are far more malicious uses of logic bombs in which the detonation of a logic bomb results in the destruction of useful files or equipment [22]. Many logic bombs act as an activation method for other viruses or worms that may be hidden within the software. When the condition is triggered, the logic bomb deploys its destructive payload which then runs on the host machine and will often times spread to other devices. The effect of the payload can vary greatly depending on the intention of the attack ranging from creating a minute change in a database to rendering a machine unusable. Detection of logic bombs can prove to be very difficult and often relies on passive deterrence such as strong password management and employee privilege monitoring. While passive detection/prevention can deter and prevent some logic bomb use, there is still a need for more involved monitoring in the form of software that is able to analyze source and/or binary code to detect the presence of logic bombs. This section will go into further detail about previous logic bomb use cases to demonstrate their effectiveness and illustrate their potential threat to UAVs, current logic bomb detection methodology, and logic bomb detection in the context of UAVs.

2.3.1 Previous Use Cases

Logic bombs have a long history in the world of corporate espionage and disgruntled employees. Oftentimes, the culprit responsible for the deployment of a logic bomb

on commercial-scale computer systems is a disgruntled employee who feels undervalued or is creating purpose for their position [22]. Perhaps the most recent occurrence was in 2016 when a Siemens contract programmer was caught inserting a logic bomb into a Microsoft Excel spreadsheet [23]. This logic bomb would cause errors in the spreadsheet which would require the company to hire him to fix the piece of software. The culprit was eventually caught when his logic bomb was mistakenly activated while he was out of town and he was forced to relinquish his computer passwords. While the Siemens employees were viewing the code for the Excel program, they noticed the malicious code and the offender was reported to the authorities. The culprit was able to continually receive income for fixing a problem that he had created.

While the aforementioned example of logic bomb insertion is relatively benign, far more insidious logic bombs have been implemented such as a logic bomb that was deployed in South Korea in 2013 [24]. This logic bomb wrote over the master boot record and hard drives of two media companies as well as three banks simultaneously. Within one of the malicious files found on the machines, there was a hex string "4DAD4678", indicating the time and date at which to trigger. This attack rendered various infrastructure machines inoperable for a period of time and also affected ATMs across the country. The simultaneous triggering of the malware across a wide variety of machines makes this type of attack incredibly difficult to contain and to stop. Time and date is just an example of a potential trigger but logic bombs can use any condition as a trigger. In the context of UAVs, there are numerous potential trigger conditions that an adversary can choose to use which allows UAVs with the malware to only be affected when the adversary would like.

These examples are utilized to illustrate the level at which logic bombs have the potential to impact systems while remaining undetected for long periods of time, and where potential logic bomb developers can be found. In many cases, logic bomb

implementation is carried out in a lone wolf type scenario where the hostile actor is choosing to plant a logic bomb due to negative feelings towards a company or entity. However, consider the situation with DJI ground station software containing malware that sent information back to it's home country [4, 5]. The UAVs were to be utilized by the U.S. Department of Homeland Security as well as the United States Army. A foreign nation's corporation has planted malware on devices that are to be used by a foreign government entity. By utilizing logic bombs implanted within commercial software, it wouldn't be difficult for a foreign adversary to gain access to another country's UAV information. Consider a scenario in which the malware is instead a logic bomb and it is embedded within the UAV software. It would be relatively simple from there to have a logic bomb that makes adversary UAVs unusable once the logic bomb has been triggered.

When discussing the implications of a logic bomb implanted within a critical system on a UAV, it is important to consider the mission that the UAV is fulfilling and the scope at which the logic bomb is operating. A logic bomb that is successfully embedded within a critical UAV system has the potential to change any number of system parameters and affect UAV operation in a wide variety of ways. Although there are no publicly documented cases of logic bombs planted on UAV software, it is easy to see how logic bombs are an appealing vector for future attacks. A relatively benign attack may simply make the battery appear to be lower than it realistically is, impacting the mission by reducing longevity and efficiency of sorties. A more overt attack could immediately remove UAV engine functionality resulting in a total loss of the aircraft.

2.3.2 Logic Bomb Detection Methods

Detection of logic bombs is a difficult topic as most malware analysis techniques largely ignore the possibility of the presence of logic bombs and rather focus on finding the insertion of malicious code into running applications [25]. The authors of [25] go on further to state that other software assurance techniques prove to be inadequate to reliably detect logic bombs. Before discussion can be held about why current software assurance techniques are thought to be invalid for logic bomb detection, it must first be discussed how logic bombs end up on systems in the first place.

Logic bombs have the benefit of circumventing traditional malware analysis techniques. They are often inserted into code by developers who have access to the code and intimate knowledge of the system which gives them the ability to hide the malware and setup effective triggers. Insider threat attacks are more formally known as lifecycle attacks. Lifecycle attacks are defined by [26] as "...malicious code inserted surreptitiously by insiders into the source code of the application before it is deployed in the field." As previously mentioned, most malware analysis tools focus on the injection of malicious code into running applications and not the detection of dormant malicious code wherein lies the difficulty of detecting these attacks.

To combat the potential for insider threats and insertion of malicious logic, many corporations have turned to training employees to watch for suspicious behavior in their peers as this may indicate the potential for destructive behavior [22]. Additionally, it is recommended that developers enforce strict policies in regards to password management and limiting individual access to only the source code required to accomplish their job [21]. The aforementioned methods are a passive technique for preventing the insertion of logic bombs but do little in regards to the detection of such malicious code. While these measures are helpful, they are no replacement for an active detection method. Unfortunately, detection of dormant malicious code, or

logic bombs, is extremely difficult unless the use of specific code analysis techniques and tools are utilized. Figure 3 shows various software analysis techniques and why these techniques are largely inadequate at preventing what are known as lifecycle attacks.

Tool category	Reasons why tools/techniques in this category are inadequate in preventing lifecycle attacks
Static analysis	Logic bombs need not rely on presence of programming vulnerabilities such as buffer overflows in target programs.
Dynamic analysis	Logic bombs need not rely on runtime errors such as memory corruption, program hangs and crashes, etc.
Penetration/fuzz testing	Logic bombs, when triggered, need not result in easily observable external events such as system crashes.
Code "diffing"	Logic bombs are inserted during development, when the affected code is anyway in flux.
Model checking	Formal models may not reflect implementation "details" such as hidden logic bombs.
Anomaly detection	Logic bombs may not cause distinct externally observable events such as unusual system call sequences.
Signature detection	Each logic bomb is "custom fitted" to its host application, and need not have a unique signature.
Technical controls	Developers can insert logic bombs in source files using privileges they already possess.
Code inspection	Detailed code inspection is tedious, error prone, and impractical, especially for large applications.
Functional testing	Functional requirements are unlikely to supply the secret input triggers known only to their malicious developers.
Unit testing	Unit tests supplied by developers, who may be the ones to insert the logic bombs, cannot be relied upon.

Figure 3: Code Analysis Tools: Effectiveness of Code Analysis Tools for Logic Bomb Detection [25]

According to [25], previously the only way to detect the presence of a logic bomb is to manually go through the code and examine every branch within the code. Any branch within the code body can contain the trigger condition for the logic bomb. However, the authors of [25] have created software they named "SQA Tool" which uses a mixture of static and dynamic program analysis with white-box testing techniques to search a target piece of software for the presence of logic bombs. Their software ensures that nearly every branch of code is entered at least once and it will attempt to force the conditions present in order to enter any branch that is encountered. The software doesn't automate the entire process however, as there are some branches in which a developer must determine the proper conditions to enter a specific branch of code. While this tool doesn't totally automate the detection of logic bombs, it does cut out significant amounts of developer interaction in order to find the presence of logic bombs.

Another effective tool that has been developed for the detection of logic bombs is called TriggerScope. This tool was designed to detect the presence of logic bombs within Android applications. With the Android mobile platform comprising 78% of all smartphones, a tool needed to be developed that can detect logic bombs lurking within applications submitted to the Google Play Store [27]. This analysis tool combines symbolic execution, path predicate reconstruction and minimization and inter-procedural control-dependency analysis which enables the researcher to detect the trigger conditions for malicious code. Utilizing this tool, researches achieved a 100% detection rate on the malware set used in their research. This tool is a great model that can be implemented for various programming languages to detect malware in a wide range of applications.

The previously mentioned solutions requires some form of the source code from the software in question. This thesis seeks to address the specific situation in which

source code of the device is not available. In addition, the wide range of programming languages, hardware, and configurations utilized by UAVs, increases the challenge with implementing such a tool for logic bomb detection. Development of a tool similar to SQA Tool or TriggerScope is infeasible for UAV logic bomb detection, primarily due to unguaranteed access to source code, but also due to the wide variety of factors previously mentioned.

2.3.3 Logic Bomb Detection in UAVs

The guaranteed way to determine if a logic bomb is present on a system is to test every possible input combination into the system. However, due to the number of potential input combinations into a system, searching for logic bombs through brute force typically isn't possible. In regards to UAVs, input vectors include externally-supplied data such as GPS or command & control data, or internally-derived data such as altitude, attitude, or other sensor data. Furthermore, according to [28], the chance to detect the presence of logic bombs by utilizing a tool that simply provides combinations of inputs into a piece of software is practically zero. Therefore, detecting a logic bomb in a UAV by brute force input testing is, at best, very unlikely.

Difficulties with logic bomb detection are compounded by the fact that many UAV systems will have to be treated as black boxes. Many commercial or government-grade UAV systems have to be treated as black boxes because the release of expensive source code allows their product to be easily replicated. Without access to source code even manually examining all potential branches in the code becomes extremely difficult and time consuming as only the software binaries are available. With the complex software typically found in UAV systems, performing manual analysis on UAV autopilot source code binaries, even with the aid of tools, is currently infeasible. Furthermore, UAV platforms utilize many different programming languages and operate on a wide

variety of hardware devices. Developing tools that cover the multitude of high level programming languages and architectures used in UAVs is challenging, as all devices within the system need to be scanned and validated to ensure there are no logic bombs present on the device.

These factors make logic bomb detection in commercially procured UAVs extremely difficult. This research proposes a methodology that does not require source code and efficiently addresses the problem of input space explosion. Before our solution to this problem is proposed, black box testing methodology must first be discussed and the relationship between black box testing and UAVs without source code must be illustrated.

2.4 Black Box Testing

One of the key aspects of this research is to develop a logic bomb detection technique that doesn't require access to the source code of the UAV in question. Given this condition, the UAV must be treated as a black box where the internal workings of the device are unknown and only the input and output values are gathered. Developing a logic bomb detection technique that doesn't rely on the source code of the device in question is extremely useful as UAVs from various developers can be validated as secure without requiring any privileged access to the device. In order to understand the challenges and the proposed technique with which to find the presence of hidden logic bombs, traditional black box testing methods must be understood.

Traditional software testing techniques fall into one of three categories: black box, white box testing or grey box testing [29]. For purposes of this research, we are only concerned with black box testing techniques as we seek hidden logic bombs in a UAV system while treating the UAV system black box. Black box testing is often referred to as functional testing and can be described as testing a piece of software without

knowing its internal structure, code or implementation [30]. Testers provide inputs to the software and view the corresponding output to ensure that it matches expected output.

There are various advantages and disadvantages to each testing technique and, ideally, all testing techniques would be utilized to maximize source code coverage. Maximizing source code coverage simply means covering as many conditions possible within the code to find as many faults or bugs that may be present. With the proper test cases created, any potential issues can be caught. Utilizing black box testing has many advantages such as its efficiency when being performed on large code bodies, keeping user and developer perspectives separate, not requiring knowledge of the original programming language and quick generation of test cases [29, 31]. With these advantages come some of the following disadvantages: a limited number of test scenarios are performed resulting in limited code coverage and the requirement for explicit entity specifications.

Within the scope of black box testing, there are various testing techniques that can be performed. Shown in Figure 4 is a depiction of some of these techniques.

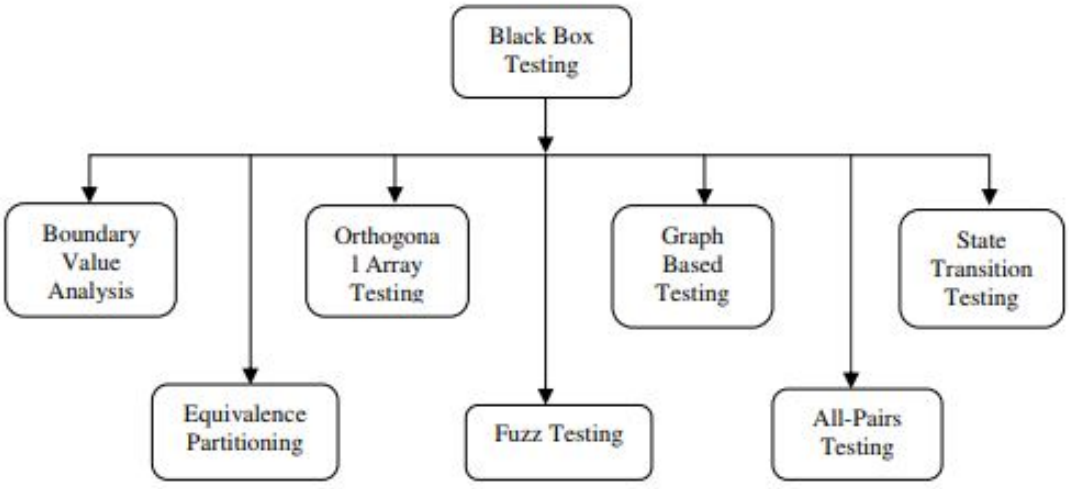

Figure 4: Black Box Testing Techniques: Various Black Box Testing Techniques [31]

This research will not go into depth about the various testing techniques but each has advantages and disadvantages. In this thesis, we will be utilizing a form of equivalence partitioning. Equivalence partitioning is defined by [31] as a "testing method that divides the input data of a software unit into partitions of data from which test cases can be derived."

Figure 5: Equivalence Partitioning: Visualization of Equivalence Partitioning [32]

Figure 5 illustrates how valid and invalid conditions are partitioned into equivalent groups. For example, any value in the range 1-10 is valid so only one test case must be for the values that fall within that range. The technique developed by this research performs equivalence partitioning by utilizing the proposed flight path for the UAV to run various simulation trials to determine appropriate range values for each phase of flight. These equivalence partitions are then utilized to greatly reduce the input space and to determine a boundary range for the output information.

In short, the UAV is treated as a black box and a type of equivalence partitioning is used to generate input cases. This technique utilizes the planned flight path in conjunction with multiple trial runs of the UAV through the flight path to develop a smaller input space. The outputs are then evaluated by examining the flight path that was taken by the UAV to determine if a logic bomb trigger condition was met.

2.5 Ardupilot

The following sections provide background information on the entities utilized in the SLIVer methodology.

Ardupilot is an advanced, open source autopilot software that can be utilized on any vehicle system [33]. This autopilot software is installed in over one million vehicles and boasts advanced data-logging, analysis and simulation tools. The source code for Ardupilot is open source which allows the autopilot to support a wide variety of sensors, companion computers, and communications systems. These features have made this software extremely popular among professionals and amateurs alike. This software is utilized in multiple OEM UAV companies and is utilized for testing by NASA, Intel, and Boeing.

This platform was selected for this research due to the fact that the platform is entirely open source. It has been stated that this research has the stipulation that the UAV in question must be treated as a black box which makes using an open source platform seem contradictory. However, this autopilot code offers a variety of testing, analysis and simulation features that are required to develop the logic bomb detection methodology. In addition, a logic bomb has to be created within the autopilot software that can be triggered within the simulation tool, and using an open-source platform, complied with the logic bomb included, is the most realistic. From the perspective of the methodology development, there is no requirement for access to the internal workings of the autopilot software.

2.5.1 Ardupilot Software in the Loop (SITL)

The most attractive feature that Ardupilot offers for this research is the SITL capability. SITL allows a user to run ArduPilot on their PC directly without any additional hardware [34]. The sensor data input into the simulation software is gen-

erated from a flight dynamics model in a flight simulator. These features give the ArduPilot simulation high fidelity and allow for accurate flight simulations with the actual aircraft code running in the simulation. The SITL software is able to simulate a wide variety of vehicle types such as multi-rotor aircraft, fixed wing aircraft, ground vehicles, underwater vehicles, camera gimbals, antenna trackers and various other sensors. ArduPilot's SITL and its logging features are the centerpiece for demonstrating the logic bomb detection methodology. Figure 6 shows how the ArduPilot software works in conjunction with the simulation software and other peripherals.

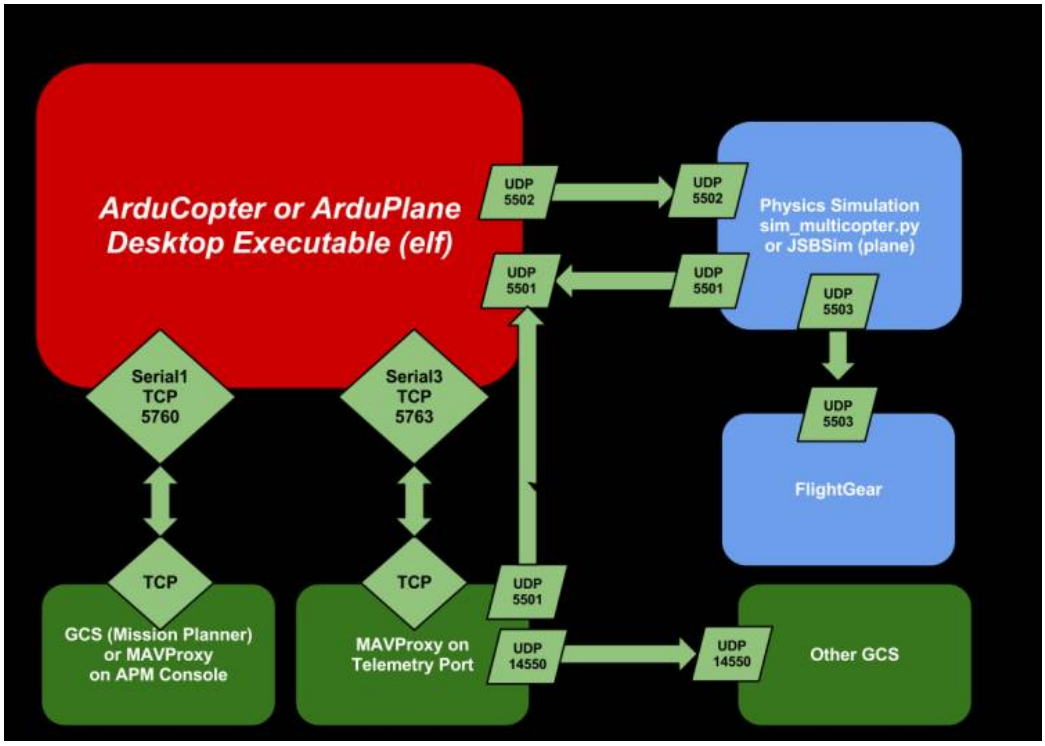

Figure 6: ArduPilot SITL: SITL and Peripheral Interactions [34]

The simulation software runs on the desktop via a compiled executable and receives data from the FlightGear simulation software in addition to the physics simulator. The autopilot software is controlled via MAVProxy through a TCP connection, simulating a real connection to a ground control station. It is also shown that the simulator can interact with other software entities such as Mission Planner which will

be used in this research to create flight plans for the autopilot to execute.

2.5.2 Ardupilot Flight Controller Firmware

For the ArduPlane simulation technique, the SITL simulator is able to simulate different flight control boards, known as Hardware In The Loop (HITL) simulation. Unfortunately, at this time, there is no support to HITL simulation in conjunction with the ArduCopter code base. Given this restriction, a vanilla flight control board is simulated. Despite not being able to simulate the exact type of flight control board that would be present on the UAV with this software, the results obtained from the simulator flights are expected to be very similar to what they would be with the HITL capability included. In the future, having the option to simulate the hardware in conjunction with the software will provide higher fidelity results.

2.5.3 MAVProxy and Mission Planner

The MAVProxy documentation webpage describes MAVProxy as a "fully-functioning GCS for UAV's" [35]. This Ground Control Station software allows the user to interface with their UAV in a simple and efficient manner. MAVProxy has exceptional log collection capabilities which allow users to perform in-depth post-flight analysis of mission runs that will be utilized for this research. MAVProxy includes functionality with Mission Planner software which allows users to create advanced flight paths for the autopilot to automatically run the UAV through the mission. Figure 7 is the MAVProxy software interface running in Windows 7. The software shows a map display with the UAV location as well as a separate console window that continually updated with various UAV flight parameters. Not shown in the figure is the command line window where commands are sent to the autopilot.

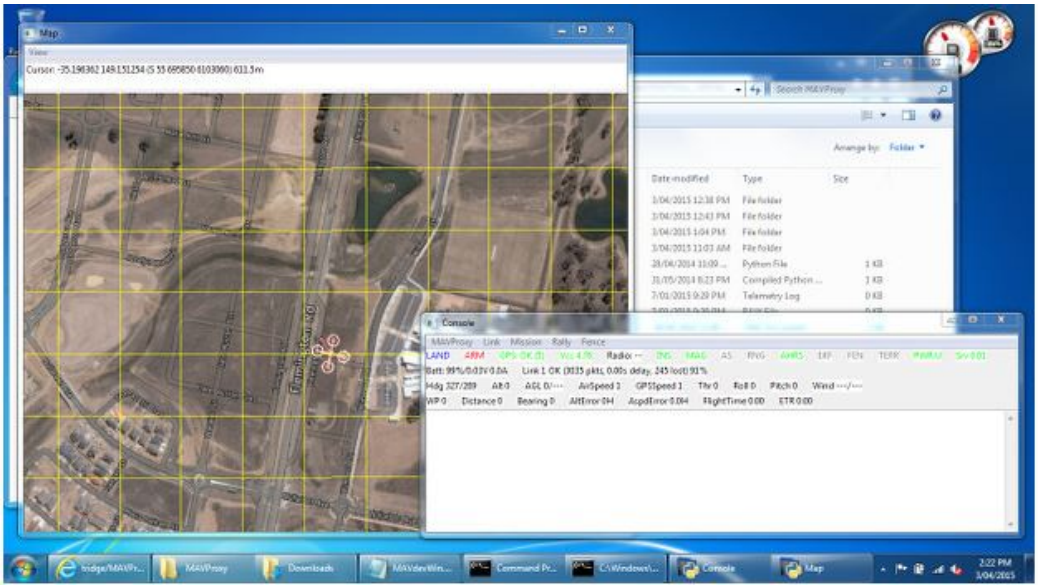

Figure 7: MAVProxy: MAVProxy User Interface [35]

2.5.3.1 Mission Planner

Mission Planner is another ground control software that is compatible with ArduPilot software in the loop functionality. This software can also be utilized to connect with a UAV while it is flying in real time as a traditional ground control software suite with which to interface with the UAV. Mission Planner has a wide variety of features such as point-and-click waypoint entry, mission selection drop down, mission log file download, APM setting configuration, and interface with a PC flight simulator to create a full hardware-in-the-loop UAV simulator [36]. In addition, this tool boasts advanced data analysis tools, log file type conversion, and many other features. Many of these features are utilized for this research but the primary use of this tool was to create flight plans for the UAV missions.

Illustrated in Figure 8 is the mission planning window. Mission plans can be saved, loaded, and edited from this interface, and detailed information is shown about each respective waypoint in the lower section of the window.

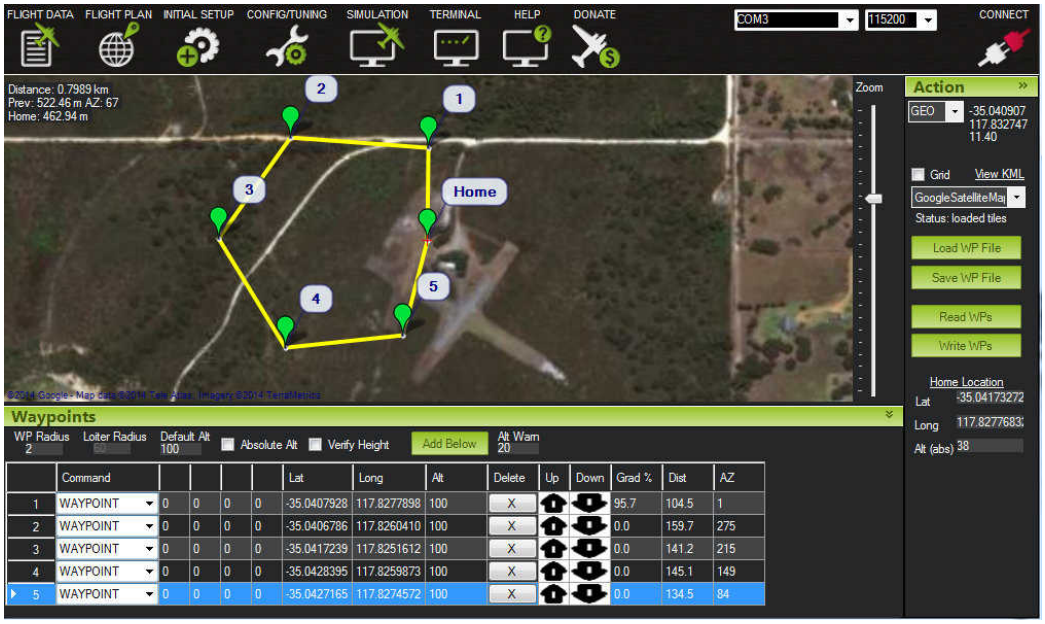

Figure 8: Mission Planner: Mission Planner User Interface [36]

2.6 Summary

UAVs face many glaring security issues. As the UAVs continually perform missions of increasing level of importance, it is essential that these security issues are addressed. While many security issues have been identified by researchers, one potential vulnerability has received little discussion: UAV autopilot logic bombs. With many modern UAVs being procured from multinational corporations, there is little chance that users have access to source code. No access to UAV source code makes searching for the presence of logic bombs even more difficult. For this reason, the SLIVer methodology was developed to detect logic bombs inserted within UAV autopilot code that doesn't require user access to the source code to function.

III. Methodology

3.1 Preamble

The growing use and dependence on UAVs to perform certain tasks demands that these devices are free of malicious code that may inhibit their ability to perform their mission. As previously demonstrated, there are a wide range of vulnerabilities that may be present in UAV systems. These examples show ways in which UAVs have been exploited, however, there haven't been any documented occurrences of logic bombs present within UAV software. With UAV source code sold by international entities or developed by open-source programmers, there is the chance that malicious logic is embedded within source code to fulfill the desires of the rogue programmer.

This research seeks to develop a basic methodology for utilizing UAV simulation software to find logic bombs embedded within UAV software. The trial runs performed in this experiment are carried out treating the UAV in question as though it is a black box and the source code cannot be accessed. Therefore, the only method to determine if a logic bomb has been triggered is by generating the inputs into the simulation and examining outputs from the simulation software.

This chapter is best viewed in two separate parts. The first part discusses the SLIVer logic bomb detection methodology in general terms and the associated concepts that must be described in detail.

The second half of this chapter describes the proof of concept experiment. This portion includes specific system configurations in addition to instructions to configure the system. Additionally, this portion of the chapter covers parameters, factors, and workload considerations for this experiment. These topics are followed by the detailed experiment design to validate the SLIVer methodology.

3.2 Part 1: SLIVer Description

This section introduces SLIVer logic bomb detection methodology in general terms. Figure 9 is a flow chart of the SLIVer methodology. The following sections within the SLIVer Description section are broken down into the specific steps of the methodology with each step discussed in-depth. The proof of concept experiments follow the steps laid out within the flow chart to illustrate the effectiveness and the proper utilization of the SLIVer. All steps on the flow chart are left in general terms with the proof of concept experiments providing specific values for pertinent steps on the flow chart.

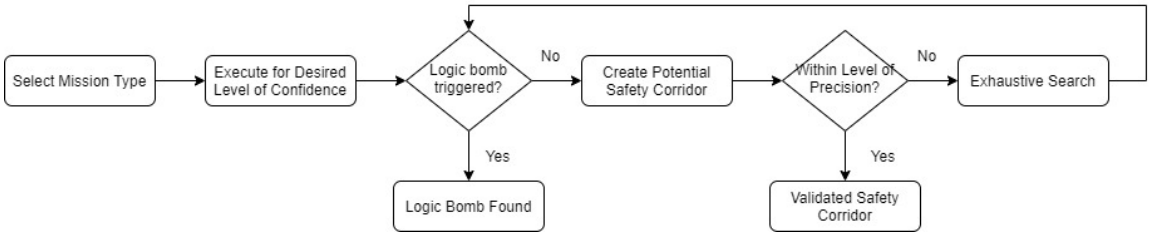

Figure 9: SLIVer Flow Chart

3.2.1 Mission Type Selection

The premise behind SLIVer is that reducing the input space to a UAV's autopilot is essential to detect the presence of logic bombs. As it stands, the current best methodology to detect logic bombs hidden within UAV autopilot code is a brute force approach, or testing every possible combination of inputs to search for a trigger condition. This technique is simply not feasible due to the vast number of inputs that a UAV can experience. One contribution that SLIVer makes is a means to reduce the UAV input space. This critical step is accomplished by utilizing the UAV's flight path as a baseline for the potential input space.

SLIVer gains efficiencies over a brute force search approach by segmentation of the UAV flight path. Segmentation of the flight path is based off of the nature of

flight activity between given waypoints. Flight path segmentation is completed by the analysis script after enough missions runs have been completed to gain a desired level of confidence. The analysis script uses mission selection input to determine how waypoints are characterized. For example, transit waypoints where the UAV is traveling from one point to another are classed differently than survey waypoints. Transit waypoints are subjected to statistical analysis operations to generate a set of inputs that the UAV is likely to experience when following the defined path. Portions of the flight path that require dynamic flight operation or free navigation such as surveillance of an area are subjected to an exhaustive search of the area. Thus, segmentation greatly reduces input space as exhaustive search is only performed on the portions of the flight path where using such a technique is necessary. Using the input reduction techniques described, the upper bound for input space, at worst, is equivalent to the input space of the exhaustive search technique. Figure 10 shows how a mission can be segmented into transit and exhaustive search areas as opposed to performing an exhaustive search over the entire area of operation.

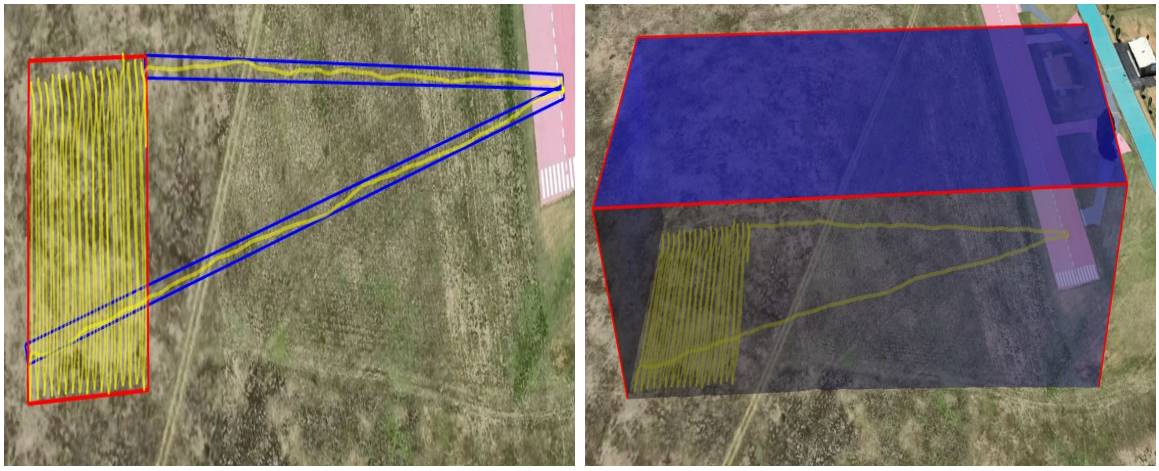

Figure 10: SLIVer Input Space Reduction Compared to Exhaustive Search

Both panels in Figure 10 show the same survey mission with takeoff, transit, survey, and landing portions. Exhaustive search areas are defined by a red border with

statistically defined input spaces defined by the blue border. The yellow line indicates the mission flight path. By utilizing SLIVer's input space reduction techniques, the reduced input space portions are examined only at the given altitude whereas the brute force approach must also account for all altitude levels up to the mission altitude. Through these means, only the input space of relevance to the UAV flight path is examined as opposed to searching the input space of the entire area of operation. Figure 10 clearly illustrates the benefit of utilizing SLIVer's input space reduction technique over a brute force approach.

3.2.2 Execute for Desired Level of Confidence

With the flight path established as the basis for the input space, there is still the problem of determining what the actual inputs are that a UAV is expected to experience throughout the course of the mission. To determine what inputs the UAV is likely to experience on a given flight path, simulation software is utilized to simulate mission runs and gather data logs. By executing a mission one time, it gives the user a general idea of what to expect in regards to what the inputs look like. Additional runs are executed to allow for variations in the flight and so that statistical analysis can be performed on the expected inputs.

To determine the full range of inputs that the UAV is likely to experience for a given mission, the SLIVer methodology creates an area along the entire flight path within which the UAV is expected to operate with a high degree of statistical confidence. A high degree of confidence is defined as the range of inputs along the flight path that the UAV is likely to encounter. In other words, we generate a confidence interval (e.g. 95%) to describe how certain we are that a given input, at a given spot along the flight path, will fall within the range generated by the analysis script. If a predetermined number of missions are completed without a logic bomb trigger event,

the previously described area along the flight path has the potential to become what is known as a "safety corridor".

3.2.3 Determination for Logic Bomb Trigger Event

It is critical to discuss potential logic bomb event detection methods that SLIVer may implement before discussing safety corridor generation as the UAV may encounter a logic bomb trigger event without having to generate a safety corridor. One potential detection method is to observe readouts of the simulation flight data console. In the event of adverse flight behavior, many UAVs issue a warning message to the GCS. Upon activation of logic bombs, the UAV may issue commands to notify the operator of abnormal activity. These built-in messages can be utilized as potential flags for malicious activity. Another potential detection method makes use of post-flight logs. These logs can be inspected for values outside of typical operating conditions for certain parameters of interest, or in the case of log entries that track UAV position and attitude, they can be viewed in mapping software such as Google Earth. Examination of log entries for adverse behavior is a valid detection technique but can be very time consuming if implemented poorly. A final proposed detection technique is by utilizing machine learning algorithms to examine log entries gathered after flights. With a database built of known "good" and "bad" flights, machine learning algorithms could potentially learn how to identify logic bomb trigger events.

3.2.4 Create Potential Safety Corridor

A safety corridor represents the area along the flight path that the UAV is likely to be within (with a desired level of statistical confidence), and due to multiple simulated mission runs having already taken place without incident, is safe for the UAV to operate within.

There are two types of safety corridors that can be generated: large and small safety corridors. A small safety corridor is acceptable as is and can be utilized as is. A safety corridor is defined as a "large" safety corridor when the generated safety corridor is larger than the precision level of the equipment utilized to measure a certain parameter. For example, the proof of concept explored in Section 3.3 assumes a GPS receiver with a precision of 1 meter. If the confidence interval for a location parameter exceeds 1m at a certain point in the flight, this is an example of a large safety corridor which requires an additional step. That additional step is an exhaustive search to ensure that the UAV experiences all theorized inputs for a given mission.

Exhaustive search consists of comprehensively flying through the given large safety corridor area to a predetermined level of precision. This concept is best visualized by imagining the UAV traversing a polygon the size of the large safety corridor and making consecutive passes through the area with the separation between passes being the defined minimum pass distance (e.g. 1 meter, given the previous GPS example).

It is important to address why accounting for sensor precision is critical when developing the safety corridor. Not only is it prudent to operate on the same scale of precision as the hardware devices present within system but for an adversary to develop a reliable attack, the precision of the device must be accounted for. Creating a trigger condition that waits for very precise input values is unlikely to be triggered by a device that is unable to provide the required resolution.

There is one final concept that must be discussed when considering the safety corridor. In the context of this research, a safety corridor purely relates to a physical location in space. However, it is hypothesized that a safety corridor can be developed for any measurable and loggable parameter on the UAV. For example, consider a logic bomb that targets the camera gimbal on the UAV to make it impossible to take photos of a specific location. Test flights can be run for missions with the analysis

script also gathering camera payload log entries. The camera should theoretically always be in the same position at a given location along a flight path to survey an area or photograph a structure. A safety corridor could be constructed by the analysis script in the same manner for the camera gimbal as it is for a safety corridor relating to the UAV's physical location. In this way, SLIVer can extend to other systems on aircraft.

One final consideration for SLIVer implementation is that it is important to consider the fidelity of the simulation software. Any system that a user would like to validate prior to mission execution must be accurately represented within simulation environment in order to gather the required logging data to build safety corridors. Without an accurate and reliable representation of how systems operate during real missions, SLIVer will have decreased levels of success.

3.2.5 Exhaustive Search

The exhaustive search seeks to ensure that the UAV autopilot software experiences in simulation every input along the entire flight path that the UAV can reasonably expect to encounter in its actual mission. As already described for "large" areas of the safety corridor, this is done by performing passes within areas that fall outside of the precision of navigational equipment (or other equipment relating to parameters in question) on the UAV. The same approach is applied to areas in the flight plan where the UAV needs more freedom of navigation, such as performing surveillance in a particular region. The passes are separated by the greatest level of precision of the sensor equipment. Given the previously introduced GPS example, the UAV would perform passes with 1 meter of separation throughout the area in question. If no logic bomb trigger event occurs, the area is certified as part of the safety corridor.

3.2.6 Requirements to Implement SLIVer

While SLIVer was developed with small quad-copters as the primary testing vehicle, there is no reason to believe that SLIVer cannot be utilized in different aerial vehicles. The primary requirements for SLIVer application are (1) a high-fidelity simulation environment that has advanced logging capabilities and (2) mission planning tools. With the aid of these two primary capabilities, it is expected that SLIVer can be scaled to any aerial vehicle.

3.3 Part 2: SLIVer Proof of Concept Experiment

This portion of the chapter covers the proof of concept demonstration of SLIVer. Within this section are the specific details relating to the experimental environment and the configuration settings. Finally, an overview of the proof of concept experiment is discussed in greater detail.

3.4 Goals

A series of goals, listed in sequential order, has been created to facilitate in the demonstration of SLIVer. The remainder of the sections in Chapter III closely mirror the order of the goals.

- Successfully Set Up Simulation Suite

  This first goal of is to create a working test environment for the simulated UAV. Included in the setup is downloading the UAV code base, GCS software setup/configuration, and any peripheral software setup/configuration. The steps performed to accomplish this goal are described in Section 3.4.2.

- Create a Series of Varied Test Missions

Creating the test missions is one of the most critical parts of this research. Picking test missions that allow the UAV to experience a wide range of flight profiles ensures that the UAV is experiencing an environment as close to the real-world as possible.

- Create Analysis Script to Facilitate Log File Analysis

  Before executing test missions, an analysis script is created to generate regions determined to be safe for flight operation. The analysis script parses through the log files generated from mission runs for log file entries of interest. The entries are used to create a region of safe operation along the flight path, known as a safety corridor, for the UAV by performing statistical analysis on the entries across multiple runs.

- Run and Evaluate Test Missions

  Running of the test missions simply consists of running the simulation software with the malicious autopilot build. Evaluating these missions is slightly more complicated as the analysis script must also be completed. This analysis script is used to parse the log files generated from mission runs. The information from the mission runs will fall into two categories: those with a logic bomb trigger event and those without. Runs without logic bombs (minimum of 10) are to be executed through the analysis script to define the area where the UAV is expected to run as normally with a high degree of confidence. In situations where the UAV is found to have triggered a logic bomb, the goal of identifying a logic bomb has been accomplished.

3.4.1 Logic Bomb and Associated Effects

A logic bomb is developed to test the proof of concept for this methodology as there are no documented ArduPilot autopilot logic bombs found within the public domain. The logic bomb created has a few key characteristics: drastic effect upon activation, multiple input parameters for trigger condition, repeatable behavior, and triggers based on UAV location information. A drastic effect upon activation was designed to aid the proof of concept experiment with better detection techniques proposed in the Section 5.4. A trigger condition is a combination of variables required to begin logic bomb code execution. These characteristics serve to guide the creation of a logic bomb that is relatively easy to predictably trigger and detect for the purposes of methodology development.

For this experiment, a logic bomb is placed within the ArduPilot autopilot code. The UAV is to operate normally unless the UAV reaches the predetermined logic bomb trigger condition. Upon reaching the logic bomb trigger condition, the logic bomb interferes with UAV motor causing the UAV to lose control and eventually crash into the ground.

Trigger condition parameters selected for this research are latitude, longitude, altitude, or any combination of the three. The resulting logic bomb satisfies all requirements: easy identification of activation, repeatable, and triggered from multiple location-based parameters.

3.4.2 System

The system for this experiment consists of the ArduPilot simulation space with the use of the MAVProxy software as the ground control station software, and the Mission Planner software that is used to create and load the different flight paths. The system configuration remains the same for the entirety of the experiment besides

the potential introduction of wind values into the environment. Default system configuration settings remain in place and the only parameter changing throughout the course of this experiment is the flight path that the simulated UAV is running.

Leaving the configuration settings the same and only varying the flight path for each different experiment serves as a means to simplify the development of the logic bomb detection technique. If SLIVer proves to effective it can be tested in more diverse environments. To achieve the standard configuration that is being utilized for this experiment, the following steps are provided by ArduPilot for setup on a Windows machine found at [33]. Please note that the machine utilized for experimentation was running on Windows 7 with MAVProxy version 1.5.0, Copter-3.7.0-dev and Python 3.7.3.

3.4.2.1 Cygwin

The first step to getting the ArduPilot SITL setup is to install Cygwin. According to the instructions found on the website, there is an automated process that was developed to install Cygwin in one simple step. That process didn't work properly and it is recommended that the "stepped install" version of the Cygwin steps are utilized. The installation steps for the stepped install for Cygwin can be found at [37]. To complete the installation, the instructions from [37] are followed as instructed.

3.4.2.2 ArduPilot

Ardupilot is acquired from the github page at the following link: https://github.com/ArduPilot/ardupilot. At this location, select the download zip option and save it to a location on the local machine. Unzip the file to the Cygwin home file directory. It is highly recommended that the Cygwin home directory is utilized or there can be various issues with properly running ArduPilot.

3.4.2.3 MAVProxy

Installing MAVProxy is very straightforward. Follow the instructions found at the following website: http://ardupilot.org/dev/docs/sitl-native-on-windows.html. As per the instruction website, install the software from the following link: http://firmware.ardupilot.org/Tools/MAVProxy/MAVProxySetup-latest.exe. Accept the license agreement and install the software with the default instruction options.

3.4.2.4 Mission Planner

Installation instructions for Mission Planner can be found at the following website: http://ardupilot.org/planner/docs/mission-planner-installation.html. The download link for Mission Planner is found at on the previously shown webpage as well the instructions used to install Mission Planner.

3.4.2.5 Mission Planner Configuration

Mission Planner must be configured in order to upload designed mission plans into MAVProxy. Before mission coordinates can be loaded into MAVProxy, it must be connected via UDP using the data rate 115200 to simulate a USB connection. In the top right corner of Mission Planner, simply select UDP from the drop down and then enter the data rate of 115200 and click connect. This should connect Mission Planner and MAVProxy so that mission plans can be uploaded into MAVProxy. Shown in the figure is the connection settings in the top right corner of Mission Planner.

3.4.2.6 Creating Missions in Mission Planner

Creating missions in Mission Planner is straightforward. The first step in this process is to select the flight plan button in the top left corner of the window.

With the flight path selection made, to create different waypoints simply left click

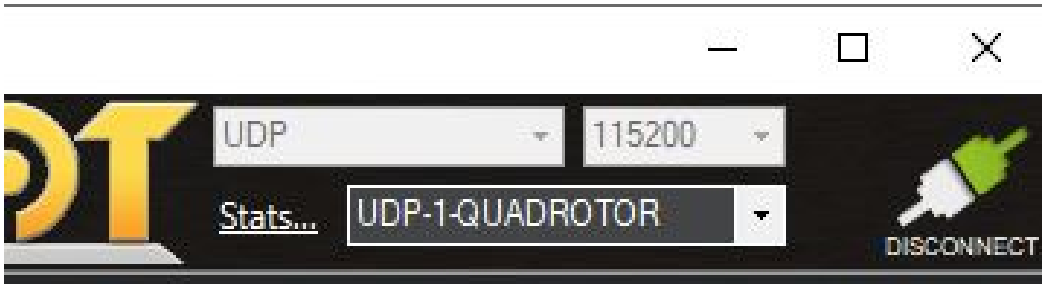

Figure 11: Mission Planner Connection Settings

Figure 12: Mission Planner Flight Path Button

on the map where you would like to create the flight path. Each waypoint that is created connects to the previous waypoint. Additionally, if you would like to add other options such as changing altitude or performing set behaviors, right clicking on the map will give additional options.

The waypoints that are created are shown in a separate window at the bottom of the Mission Planner window. The waypoint entry saves various parameters as shown and these same parameters are saved within the log file that is stored by the UAV.

The created flight path can then be saved by selecting "Save WP File." The saved waypoint file can be read in at a later time by selected "Read WP file" and the waypoint information is written to MAVProxy by selecting "Write WPs." These options can be found on the right hand side of the Mission Planner window.

Figure 13: Waypoint Creation

Waypoints

WP Radius 2 | Loiter Radius 45 | Default Alt 100 | Relative | Verify Height | Add Below | Alt Warn 0 | Spline

	Command	Delay				Lat	Long	Alt	Delete			Grad %	Angle	Dist	AZ
1	WAYPOINT	0	0	0	0	-35.3626310	149.16191...	100	X			32.4	17.9	324.7	283
2	WAYPOINT	0	0	0	0	-35.3593762	149.16148...	100	X			0.0	0.0	364.0	354
3	WAYPOINT	0	0	0	0	-35.3591837	149.16734...	100	X			0.0	0.0	531.7	88
4	WAYPOINT	0	0	0	0	-35.3632960	149.16747...	100	X			0.0	0.0	457.4	179

Figure 14: Mission Planner Waypoint Window

3.4.2.7 Running a Mission

With the proper Cygwin, MAVProxy, Mission Planner configurations, and with the flight path loaded into MAVProxy, running a simulated mission is very easy. Navigate to the /ardupilot/tools/autotest folder in Cygwin. Note that if it was decided to save the Ardupilot code in another file location than the Cygwin root directory, there may be additional errors that will have to be addressed. Once in the proper directory, MAVProxy is opened by the following command: "./sim_ vehicle.py -v Ar-

duCopter –map –console". This command will open the MAVProxy map, command console, and UAV information panel. With MAVProxy running, write the waypoints from Mission Planner. The final step to flying the designated mission is to arm the throttle and set the UAV into "auto" mode.

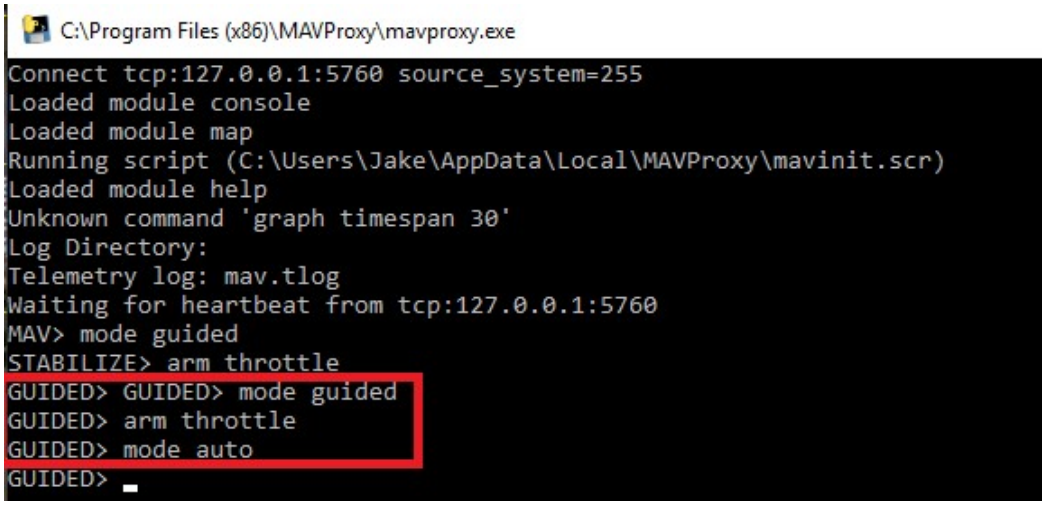

Figure 15: Begin Mission

The UAV will now complete the designed mission without any user intervention. These steps allow us to create various flight paths that can be utilized to not only create a safety corridor but to determine if a logic bomb trigger event occurs.

3.4.3 Evaluation Technique

The evaluation technique for this research is simulation. This technique provides the accuracy of the high-fidelity simulation provided by ArduPilot SITL system while avoiding the risk and complications of using physical hardware devices. Various factors such as wind and weather are circumvented (unless the introduction of such effects is desired) by performing these tests in a purely simulated environment and allow for a much higher degree of control for the experimenter/tester.

At this time, hardware-in-the-loop simulation capabilities are not supported for the ArduCopter code base. However, it is expected that this methodology works

the same with the added feature of hardware simulation. This is due to the fact that the input space remains the same and the information utilized to create safety corridors and determine logic bomb activation are based on the system as a whole. The inclusion of hardware simulation would only serve to increase fidelity of the logic bomb detection methodology.

3.4.4 Parameters

Parameters are divided into two categories: system and workload parameters. The system parameters involve the hardware and software utilized during the course of this experiment. Workload parameters are the characteristics for user requests which typically change between the specific experiment being run [38]. For this research, the system parameters remain consistent between each iteration barring one exception. The exception to this rule is the logic bomb trigger conditions between different mission runs, depending on whether or not logic bomb testing is occurring. The logic bomb that is implemented creates a very adverse effect on UAV performance which typically results in the loss of the simulated aircraft.

Workload parameters for this experiment include environment conditions and the mission plan for that flight. It is theorized that wind affects the performance of the UAV as it increases the time to complete the mission or increases the variation in the flight path when compared to a run that doesn't contain wind. The mission plan changes the performance of the UAV throughout the course of the flight path as different demands are made on the UAV depending on the mission.

Another parameter worth mentioning but isn't likely to be a factor is the logic bomb that is deployed into the autopilot code. The logic bomb effect remains the same throughout the course of SLIVer development and validation. However, the logic bomb has different trigger conditions for each mission to ensure proper and reliable

triggering on each mission.

The flight vehicle is another workload parameter that is being utilized in this experiment. Throughout each iteration of this experiment, the vehicle utilized remains the same. Although multiple different vehicle types are supported by ArduPilot, it isn't necessary to simulate a wide variety of vehicles to demonstrate the detection methodology. The detection methodology developed isn't dependent on the vehicle type and the simulation of multiple vehicles provides little benefit. Additionally, at this time, the ArduPilot SITL simulation tool doesn't support Hardware In the Loop simulation. This makes it impossible to simulate different flight control boards/vehicles.

Location of the mission run is another parameter worth mentioning. Mission locations with large geographic features such as hills or forests have the potential to cause unexpected flight performance as well as acting as obstacles for certain mission types. The mission location remains the same throughout each run of this experiment.

Since this experiment is a demonstration and proof of concept for SLIVer, the list of parameters are relatively short. The small amount of parameters in this experiment are the result of a deliberate effort to simplify the creation of the safety corridor for each mission and to accurately diagnose the presence of a logic bomb. The safety corridor generation is the single most important variable in logic bomb detection as the accuracy of the safety corridor determines the requirement for exhaustive testing on the areas where a corridor is large. As experiments and the methodology develop, more parameters can be added into the experiment to detect more complex logic bombs in environments that are far more dynamic.

3.4.5 Factors

Factors in this experiment are parameters that are varied. As discussed in the previous section, the parameters that are factors in this experiment are environmental conditions, autopilot software, logic bomb trigger condition. The first factor is variable environment conditions. This factor exists in relation to this experiment to add some variation when creating safety corridors but more importantly, to closer represent realistic flight conditions. Without outside factors, it is assumed that each run of the flight path almost always remain the same. Introducing variability into safety corridor generation more accurately simulates multiple runs of the mission plan in the world.

The autopilot software is modified only to introduce the logic bomb into the autopilot code. There are no additional changes to the autopilot software to have the performance of the UAV remain consistent unless the logic bomb trigger condition is reached. It is extremely important to have flight handling characteristics remain the same due to the fact that log files from different runs are utilized to create a confidence interval of the expected UAV location throughout the duration of the flight.

The final factor in this experiment is the logic bomb trigger condition. The logic bomb payload remains the same throughout each mission and only the trigger condition is varied. Changes in the trigger condition are necessary to ensure that the UAV flight path intersects the logic bomb trigger area for each flight path. Additionally, having the trigger location change facilitates in the creation of a robust detection methodology.

3.4.6 Workload

The workload for this thesis is concentrated in developing the flight profiles and safety corridors for each mission. Creating the safety corridor consists of performing

multiple runs of the same mission set with the same code. Developing an accurate safety corridor is critical to accurately diagnosing the presence of logic bombs. This portion of the experiment is the most computationally expensive as completing the mission runs takes time depending on how complicated the mission is. The ArduPilot SITL simulation software is able to speed up runs but the amount of speedup is limited by the hardware that is running the simulation software.

3.4.7 Experimental Design

The experimental design for this thesis closely resembles the goals discussed at the beginning of Chapter III. Simulator setup has already been described previously in Section 3.4. With this goal accomplished, the next step is to create a series of varied test missions.

Before creation of the test missions is discussed, it is important to first discuss the process that is utilized to create both the safety corridor as well as logic bomb trigger determination. When determining the number of runs that must be completed to develop the safety corridor, the variation of GPS locations between runs are examined. The amount of variation between each run is a key factor in determining the runs required to develop a comprehensive safety corridor. A number of runs must be completed with differing conditions as previously mentioned such as wind in the simulation. Wind values that increase the typical variation from the planned profile require more runs to ensure that we can safely determine what the largest expected variation from a mission profile can be. Determining the outside range is critical in developing a logic bomb detection methodology to reduce the number of false positives and to give the most accurate values to test against. 10 runs is the minimum number of runs that are accomplished for each mission to create the safety corridor. For high levels of variation found by the analysis script, more missions are to be completed to

ensure that the best flight profile is generated.

Determination of the logic bomb trigger event is based off of two factors: MAVProxy console output and .kmz file analysis. If 10 runs are successfully completed, an analysis script is utilized to create a safety corridor for UAV operation on the flight. This safety corridor is an area of operation in which the UAV is expected to behave normally with a 95% degree of confidence. If the range of the safety corridor at any point along the flight path is large (greater than 1 meter), further analysis must be completed on this area, known as exhaustive search. The exhaustive search simply means that the area must be completely flown to a resolution of 1 meter. A 1 meter resolution is chosen as it is the greatest level of accuracy typically afforded to small UAVs that run the ArduPilot autopilot code due to GPS sensor limitations. If the range of the safety corridor is less than 1 meter, a good safety corridor has been created. The creation of the test missions are described further in the following sections.

3.4.8 Test Mission Creation

These flight paths are created in the Mission Planner software. For this thesis, 4 missions have been created with each offering a different challenge in logic bomb detection. Detecting a logic bomb trigger for a simple transit mission is easier than detecting a logic bomb that has been triggered in a surveillance mission as the latter flight path has more variability. Detection of logic bombs grows more difficult as the flight paths have more variation.

Before the flight paths are created, it is important to first think about what kind of log data is relevant to determine if the UAV is operating with clean or malicious software. For this thesis, the GPS coordinate log files are the log files of interest but the log files of interest can be changed or added. As the UAV travels between

waypoints, the UAV continually transmits GPS location information to the ground station software, in this case, MAVProxy. The goal when creating the safety corridor is to determine what the usual variation in GPS locations traveled is for each experimental run with the clean UAV build. Knowing what the usual variation for GPS locations from the predefined flight path makes it possible to determine if a deviation larger than normal from the predefined flight path has occurred.

It is also important to consider that environmental factors such as wind in the simulation software can create deviations larger than normal from predefined flight paths. These variations occur in a variety of parameters such as speed, heading, yaw rate, GPS coordinate location, etc. As previously stated, this experiment is going to be closely examining GPS coordinate locations. There is the potential for environmental factors to create large deviations from the predefined flight path. The amount of deviation that occurs from the environmental factors found within the simulator are discussed further in Chapter IV

Another factor to consider is that maneuvers performed during the course of the missions can also increase the deviations in the flight path. Performing a mission such as automated surveying can create deviations larger than a simple transit mission due to the nature of flying that is required for this type of mission and the terrain that is being surveyed. This is important to consider as well because increasing the baseline variation values reduces accuracy of the detection methodology, however, this experiment going to be searching for logic bomb(s) that often cause large variations in GPS coordinate location. It is important to consider that a logic bomb can cause any number of adverse behaviors but for these experiments, we focus on logic bombs that cause adverse flight behavior.

The following 4 missions were created in Mission Planner with their corresponding flight figures shown. These missions were designed to have increasing levels in flight

path variation due to more complicated mission sets/scenarios that the UAVs will be subject to.

3.4.8.1 Transit

This is the simplest flight profile utilized and consists of the UAV simply taking off, flying to a target location, and landing. The purpose of this mission is to simulate an adversary's logic bomb having a trigger condition that lies on a predicted line of travel that a UAV would potentially take. For a real-world situation, this could simulate a military UAV transiting between bases or forward operating locations that are well traveled. In this scenario, the UAV in question would likely be a strategic asset traveling between operating locations with a malicious autopilot build loaded. The malicious logic could aim to cause a loss of vehicle event by shutting down an engine or changing data within the UAV software to cause confusion to the autopilot. More advanced logic bombs could even potentially hijack the aircraft and bring it to an adversary location. For a civilian application, a logic bomb could cause a delivery UAV to be rerouted once it reaches a certain location to an adversary drop location. In short, the effect of the logic bomb and the difficulty to notice an attack may vary greatly.

3.4.8.2 Circle

The circle mission simulates a UAV taking off, flying a circle pattern at a predefined altitude, and returning to the takeoff location to land. Again, this mission can simulate various real-world applications such as surveying a given area, or performing a holding pattern while waiting for further instructions from the ground station. This mission is slightly more complicated than the transit mission with the addition of turns throughout the course of the flight path. It is hypothesized that the addition

of turns will create an added element of variation across the runs of this flight path.

3.4.8.3 Spline Circle

The spline circle mission simulates a UAV taking off, flying a climbing circle pattern, and landing. This mission can simulate a UAV simply climbing to a specified altitude and standing by for further instructions from the ground station, performing reconnaissance for military applications, or performing some kind of monitoring in a specific location. If an adversary knows potential locations in which these operations would likely be conducted, logic bombs with trigger conditions correlating to these GPS locations can be triggered and cause adverse flight behavior. Further room for deviation is introduced from the mission type due to the climbing flight path and a circular pattern. With turning and altitude deviations, it is unlikely that these behaviors are always consistent, adding variability to flight path deviation.

3.4.8.4 Survey

The final mission type that is being utilized is the survey. A survey mission consists of a UAV that has a flight path defined by a polygon. Within the polygon, the UAV performs a flight path that traverses the entire area with a predefined separation distance. For example, within a square, the UAV may complete paths along the length of entire length of the square at a 1 meter separation distance until the entirety of the square has been traversed. This mission type in MAVProxy is utilized to simulate a grid of waypoints during which a camera is utilized to survey the area. The increasing number of waypoints adds an increased potential to deviate from the predefined flight path. This mission type could potentially be utilized for civilian and military applications such as infrastructure monitoring, forest fire monitoring, and intelligence gathering to name a few. This is similar to the circle mission sets

in the sense that an adversary will likely have a good idea of locations of interest to facilitate the creation of a trigger condition for the logic bomb.

3.4.9 Create Effective Logic Bomb and Implement Into Autopilot Software

With the simulation configuration completed and test mission flight paths created, the next step is to create a logic bomb that will be triggered at some point on each mission. The logic bomb(s) utilized in this experiment have simple triggers and have devastating affects to the UAV once triggered such as an engine failure. The goal of the logic bomb in this project is to create a piece of malware that is easily detectable at first so the proof of concept for this methodology can be established.

The logic bombs created in this experiment are located within the ArduPilot autopilot source code. These implanted logic bombs trigger on flight parameter(s) that are stored and constantly updated throughout the course of the UAV flight. The logic bomb triggers can be a specific point, altitude or any other parameter tracked by the autopilot. With the trigger condition reached, the code branches to a new location and affects engine operations which create a noticeable effect on UAV flight path.

3.4.10 Create Analysis Script for Safety Corridor Generation

The purpose of this goal is to create software that parses the log files downloaded from MAVProxy after a mission is executed by the UAV. From the log files, GPS location logs need to be parsed out so a profile is built for what an expected run of each mission looks like. These data points are grouped by their waypoint so that the most accurate profile is developed for each mission set. For example, the GPS transmissions between the starting location and waypoint 1 are grouped together in

a list data structure and then the GPS transmissions from waypoint 1 to waypoint 2 are grouped together in another index in the data structure and so on. With the information within these groupings, an accurate representation of the mission flight profile can be generated.

A python analysis script is created that parses through log file entries across multiple mission runs. This analysis script only picks out the log file entries of interest. In the case of this research, the entries of interest are those that are the GPS location and waypoint log entries. With the relevant log entries parsed out, the analysis script creates a confidence interval across the entire length of the flight path. This confidence interval represents the expected area that the UAV is supposed to operate within, or safety corridor, to the degree of confidence desired by the user or tester.

With the safety corridor generated, it can then be determined if an exhaustive search is required for any area along the length of the flight path. If there is an area that has a confidence interval with a range larger than 1 meter, an exhaustive search must be performed within that area. The distance of 1 meter was chosen based on the technical specifications of common GPS tracking devices found in UAVs.

3.4.11 Run and Evaluate Test Missions

The goal for this portion of the experiment is to run the malicious ArduPilot build and then run the analysis script to gather the relevant information from the flight path. For each mission run with the logic bomb, trigger conditions must be changed but payload of the logic bomb remains the same. Test missions are first run a series of 10 times with the logic bomb present on the autopilot code but with a trigger condition located outside of the UAV operating area. After completion of the 10 runs, the analysis script is run for each mission type. The safety corridor is generated and output by the analysis script, defining safe operating areas for the UAV.

With safety corridors generated, the test missions are then run with the logic bomb trigger condition set to some point along the flight path. At this point, the goal is to determine if the logic bomb trigger can be observed. Upon logic bomb trigger event, it is expected that a loss of power event is flagged in the MAVProxy console, indicating that the quadcopter motors are no longer receiving inputs from the autopilot. This is by design and is a clear indication that the logic bomb has been triggered. The second determination is the .kmz file analysis. A .kmz file is file type that is used to visually display the GPS log entries. Logic bomb trigger events in the .kmz file correspond to a clear loss of control and altitude loss. This event is very distinct and is demonstrated in Chapter IV.

Through these two separate tests, it is demonstrated that the methodology is equipped to both identify a logic bomb trigger event condition as well as create a safety corridor for flight operations if a trigger event is not detected. By this process, UAV users are empowered with a tool that allows them to preflight their UAV missions. Users are able to operate them with a high certainty of normal operation and trust their devices to carry out the critical tasks that UAVs are now being relied upon to carry out. The key development from SLIVer is that it affords users an alternative to brute force testing an entire area of operation for the assurance of safe operation.

IV. Results and Analysis

A series of tests were performed using the previously described software suite to evaluate the effectiveness of SLIVer. Using the research questions from Section 1.3 as guidance for experimental design, a series of experiments were developed as outlined in Chapter III and the results of which are analyzed in Chapter IV. Additionally, this chapter evaluates the limitations of the simulation software through experimentation, such as the introduction of wind and its effect on experimental data.

4.1 Assumptions

During the course of this experiment, a number of assumptions have been made in order to accurately evaluate SLIVer and simplify experimental design. These assumptions are listed below.

1. All flight control boards/UAVs will operate in a nearly identical manner with the modified ArduPilot autopilot code

   At this time, the ArduPilot SITL simulator cannot use the ArduCopter code base to simulate the different possible flight control boards. It is assumed that the autopilot code performs similarly on different flight control boards. Additionally, it isn't possible to simulate different quadcopter chassis at this time. It is apparent that some performance characteristics will vary with different combinations of copter chassis and flight control boards. However, it is assumed that the autopilot code will still function relatively the same when performing the simple maneuvers required during experimental missions.

2. The simulation software accurately represents UAV flight characteristics and location information

When conducting simulation-based testing, it is necessary to assume that the product that is being utilized as a testbed reliably produces results similar to the expected real-world counterpart. Given the credentials and accolades of the ArduPilot product, it is assumed that the simulation software being utilized accurately represents quadcopter flight characteristics to a passable degree.

3. The autopilot contains no other malicious code that would affect performance other than the code modifications introduced in the experiment

   Given that no other documented cases of implanted logic bomb UAV autopilot code has been published in the public domain, the assumption is made that the current autopilot build utilized is free of malicious logic.

4. Log files received from the UAV are accurate and not modified

   The use of log files is essential to properly implement SLIVer. An attack that tampers with log files would require a different approach.

5. UAV has unlimited battery life

   Some of the mission sets, the survey mission in particular, potentially last longer the typical endurance of a small UAV. SLIVer is designed with the hope that this process can be scaled to devices with more functionality and higher performance. Since ArduPilot doesn't have the capability for HITL simulation, it is assumed that the UAV utilized in the experiments isn't constrained by battery capacity. This also serves to focus the methodology on logic bomb detection rather than UAV capabilities as well as to provide more inputs into the UAV in one test flight.

4.2 Logic Bomb/Simulation Representation

The first step to validate SLIVer was to create a logic bomb for the ArduPilot autopilot software. The authors of [8] point out that there are various documented vulnerabilities for UAVs. These attacks include GPS spoofing attacks, standard WiFi-based attacks, jamming attacks and Trojan horse attacks. This research addresses a specific Trojan horse attack that is identified in the "hardware" section of Figure 16. The authors of [8] point out that there is the potential for UAV hardware to have hidden malicious code upon delivery to the customer; a logic bomb.

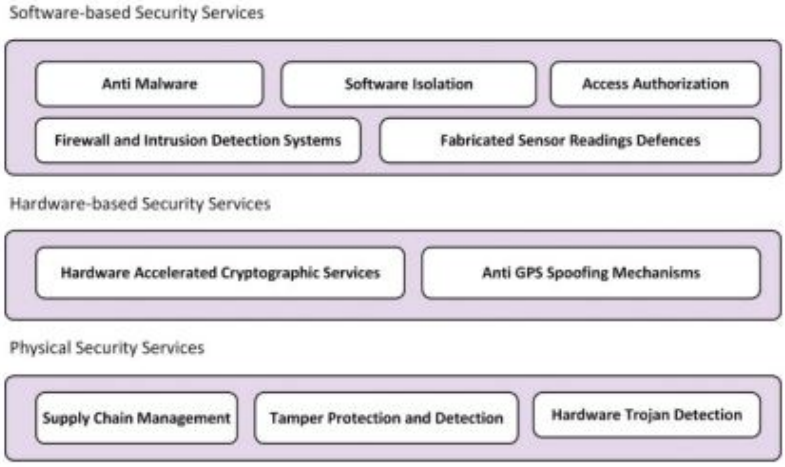

Fig. 4. Proposed security architecture for UAV systems.

Figure 16: UAV Security Architecture [8]

Introduction of such malicious code can occur at various points in the software supply chain. At the time of this research, there is no proposed methodology for validating that UAV autopilot code is free of logic bombs prior to customer use. Additionally, there are no known documented logic bombs that affect ArduPilot's autopilot code which drove us to create our own logic bomb detection methodology.

When designing the logic bomb, there were various sections of code identified as vulnerable to alterations resulting in dramatic effect in flight performance of the UAV. After examination, code injection into the ArduCopter.cpp file was determined to be

the simplest place to introduce the logic bomb. The created logic bomb targets the function that updates information to the quadcopter servo motors. In the absence of the trigger condition, the UAV will operate normally and continually send information to the motors. Upon reaching the trigger condition, the autopilot code no longer sends control information to the motors, resulting in a loss of thrust to all motors and a loss of aircraft event.

The first step was to create a flag condition so the autopilot code could be complied with and without the logic bomb activated. The APM_Config.h file was found to be the best place to perform this task. In line 49 of the file, a new #define entry was added as shown in Figure 17.

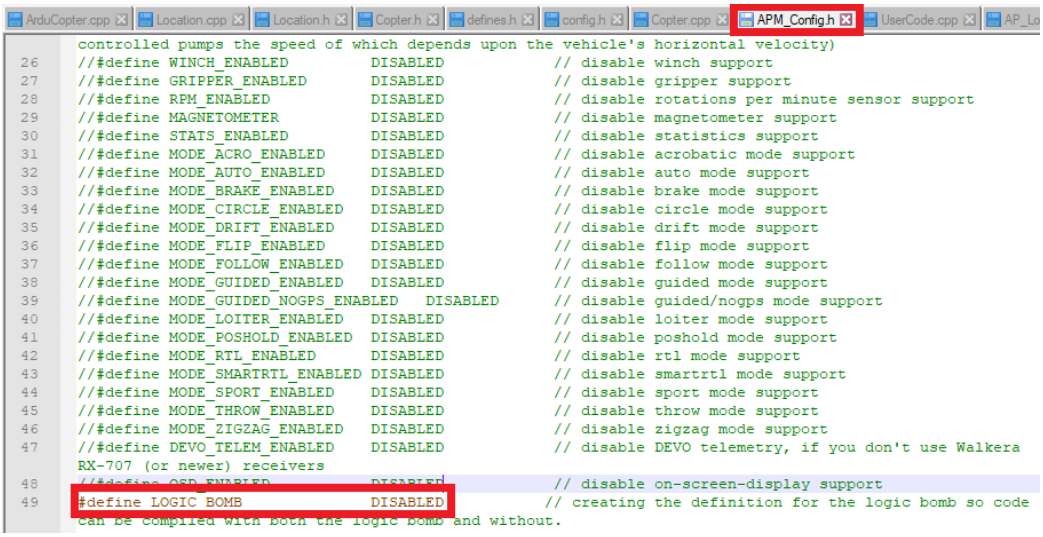

Figure 17: Modification to the APM_Config.h File

This #define statement is “ENABLED” when the logic bomb portion is to be implemented in the desktop executable and “DISABLED” otherwise. The purpose of this functionality is so the same code base can be compiled with and without the logic bomb active for testing purposes.

The next step for the logic bomb implementation was to add the malicious code to the ArduCopter.cpp file. This exploit was loaded into the Main Loop code body which

operates at 400Hz. The function that was targeted was the motors_update() function which sends motor control information to the motor servos. This code modification is shown in Figure 18.

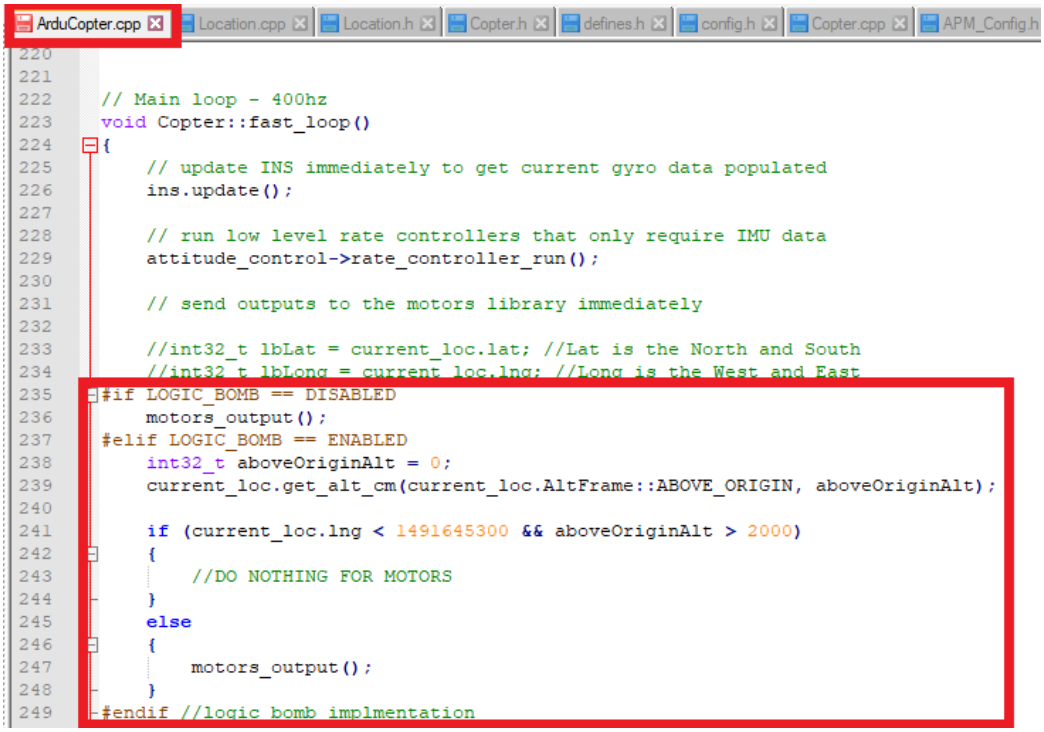

Figure 18: Insertion of Logic Bomb Code

The if statement shown on line 241 lists the trigger conditions for the logic bomb activation. Upon activation, the motors_output() function is no longer called. The value for latitude and longitude are stored within the current_loc object. To gain access to these objects, modifications were made within the Copter.h file. ArduCopter was made a friend class to gain access to the current_loc object parameters. This modification was made in line 196 of the Copter.h file as shown in Figure 19. This condition is based upon the #define condition that was created to ensure that the code operates normally when the LOGIC_BOMB flag is in the DISABLED state.

The latitude and longitude values are stored as 32 bit, fixed length integers. Entering the latitude and longitude coordinates is completed by inputting the decimal

```
180  class Copter : public AP_HAL::HAL::Callbacks {
181  public:
182      friend class GCS_MAVLINK_Copter;
183      friend class GCS_Copter;
184      friend class AP_Rally_Copter;
185      friend class Parameters;
186      friend class ParametersG2;
187      friend class AP_Avoidance_Copter;
188
189  #if ADVANCED_FAILSAFE == ENABLED
190      friend class AP_AdvancedFailsafe_Copter;
191  #endif
192      friend class AP_Arming_Copter;
193      friend class ToyMode;
194      friend class RC_Channel_Copter;
195      friend class RC_Channels_Copter;
196
197  #if LOGIC_BOMB == ENABLED
198      friend class ArduCopter;
199  #endif
200
201      Copter(void);
202
203      // HAL::Callbacks implementation.
204      void setup() override;
205      void loop() override;
206
207  private:
208      static const AP_FWVersion fwver;
```

Figure 19: Addition of Friend Class

degrees format of the location without the decimal point. Altitude is stored in the same object parameter but requires the use of the get_alt_cm() function to get the value. Getting the altitude is achieved by using the following line of code: current_loc.get_alt_cm(current_loc.AltFrame::ABOVE_ORIGIN, aboveOriginAlt). This function gives the coder the option to select if you want the absolute altitude value, above origin altitude, or the relative altitude. For this research, the above origin altitude was utilized and the value is returned in centimeters. This value is then input into the same if statement along with latitude and longitude values to create a

trigger condition that is also based on altitude. Using these parameters and targeting the motors_update() function creates a logic bomb that prevents the motors from operating once a certain trigger condition is reached.

4.3 Analysis Script

Analysis code was developed in Python to perform log parsing and safety corridor generation on the log files gathered from mission runs to aid in finding safe UAV operating areas. Safety corridor generation is accomplished by creating a confidence interval across each and every log entry across multiple runs. At this time, the analysis tool is only concerned with log entries that are relevant to the UAV GPS location and waypoint information as the developed logic bomb affects the attitude, and therefore the position, of the UAV.

4.3.1 Conversion from .bin to .log

The analysis tool accepts log files that are in the .log format while the standard format of the log files from MAVProxy are in the .bin format. The binary log files are converted to the .log format in the Mission Planner application. This step is completed by selecting the flight plan tab at the top and selecting the "Convert .Bin to .Log" option under the "DataFlash Logs" tab at the bottom of the window. The step for the file conversion is illustrated in Figure 20.

Upon selection of the "Convert .Bin to .Log" button, a navigation GUI is opened and the file to be converted is selected. The converted files must then be moved to the same directory as the analysis script.

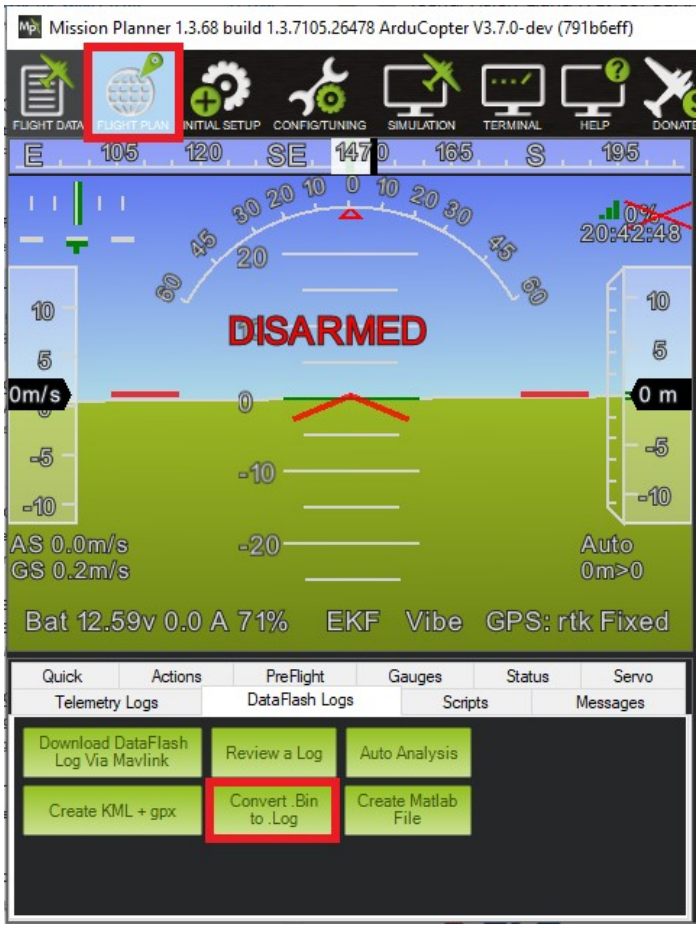

Figure 20: Conversion From Binary to .log Format

4.3.2 Selecting Files

The analysis tool is configured to search for the converted log files in the same directory of the analysis script and follows a specific naming convention. The naming convention is prefixed by the mission type and followed by a number. For example: “transit01”, “transit02”, “circle04”, “spline01”. The naming convention is utilized so that when the mission type is input into the analysis tool, the tool is able to discern which log files are selected for analysis. Figure 21 shows the prompt for the mission selection from the analysis tool.

Note that the tool expects log files to be named in sequential order.

```
C:\Users\Jake\Desktop\AFIT\Thesis\Code>python LogToCSV.py
Select Mission Type 1-4 (Transit, circle, spline circle, Survey).
```

Figure 21: Analysis Tool Mission Select

4.3.3 Parsing Log Entries

The logic bomb that was created for this experiment affects the location of the UAV during flight operations. Due to location information being affected, the GPS and CMD log entries are parsed from the log entry to be used to determine waypoint information and GPS position entries. Figure 22 shows the waypoint log entries in the log file and Figure 23 shows an example GPS log file entry. The entry type is denoted by the first 3 letters present in the entry. For this experiment, we are concerned with GPS and CMD entries. The 11^{th}, 12^{th}, and 13^{th} fields represent latitude, longitude and altitude.

```
1485  PARM, 145086109, AUTOTUNE_AGGR, 0.1
1486  PARM, 145086109, AUTOTUNE_MIN_D, 0.001
1487  PARM, 145086109, TUNE_MIN, 0
1488  PARM, 145086109, TUNE_MAX, 0
1489  MSG, 145086109, ArduCopter V3.7.0-dev (791b6eff)
1490  MSG, 145086109, DESKTOP-AEAD2KQ
1492  CMD, 145086109, 3, 0, 16, 0, 0, 0, 0, -35.3632615, 149.1652307, 584.01, 0
1493  CMD, 145086109, 3, 1, 22, 0, 0, 0, 0, -35.3633047, 149.1652607, 40, 3
1494  CMD, 145086109, 3, 2, 21, 0, 0, 0, 1, -35.3621498, 149.1650838, 0, 3
1496  MSG, 145086109, Frame: QUAD
1497  MODE, 145086109, Guided, 4, 2
1498  ORGN, 145086109, 0, -35.363261, 149.1652299, 584.01
1499  ORGN, 145086109, 1, -35.3632615, 149.1652307, 584.01
```

Figure 22: Waypoint Log Entries

The log entries are then programmatically grouped together into two list data structures. These two lists are then utilized throughout the course of the program to further group the entries by run, waypoints and then each waypoint's individual log entry. The end result is one large list that stores the contents of each run, organized by waypoint which easily allows for the creation of a confidence interval for every

```
1605  VIBE, 145316017, 0.005570653, 0.005737753, 0.005477352, 0, 0, 0
1606  CTRL, 145316017, 0.002975348, 2.171743E-05, 0.002821677, 2.033542E-05, 0.0001595216
1607  IMU, 145316017, 0.002147356, 0.002048112, 0.002189239, 0.0002379741, -0.002389336, -9.818218, 0, 0, 0, 1, 1, 999, 999
1608  IMU2, 145316017, 0.002003385, 0.002272756, 0.001992446, -0.001253549, -0.0001583903, -9.817053, 0, 0, 0, 1, 1, 760, 799
1609  IMU, 145356001, 0.002060595, 0.002013626, 0.002056205, -0.001848655, -0.002448582, -9.818207, 0, 0, 0, 1, 1, 999, 999
1610  IMU2, 145356001, 0.002127372, 0.002050972, 0.002054566, -0.003855804, -0.0006364082, -9.817352, 0, 0, 0, 1, 1, 760, 799
1611  MAV, 145368496, 0, 175, 384, 0
1612  IMU, 145395985, 0.002207521, 0.002120242, 0.002003799, -0.001816574, -6.636987E-05, -9.817823, 0, 0, 0, 1, 1, 999, 999
                                                                                                                    0, 0, 0, 1, 1, 760, 799
1614  GPS, 145415977, 6, 157786400, 2079, 10, 1.21, -35.363261, 149.1652299, 584.09, 0, 0, 0, 1
1616  BAT, 145415977, 12.58798, 12.58798, 0, 0, 0, 0, 0
1617  MAG, 145415977, 215, 73, -538, 5, 13, -18, 0, 0, 0, 1, 145415977
1618  MAG2, 145415977, 215, 73, -538, 5, 13, -18, 0, 0, 0, 1, 145415977
1619  MAG3, 145415977, 215, 73, -538, 5, 13, -18, 0, 0, 0, 1, 145415977
1620  BARO, 145415977, 0, 94502.8, 34.92, 0, 145415, 0, 25.53199
```

Figure 23: GPS Log Entry

log entry across multiple runs. Figure 24 shows a simple program flow diagram that represents the analysis tool data organization.

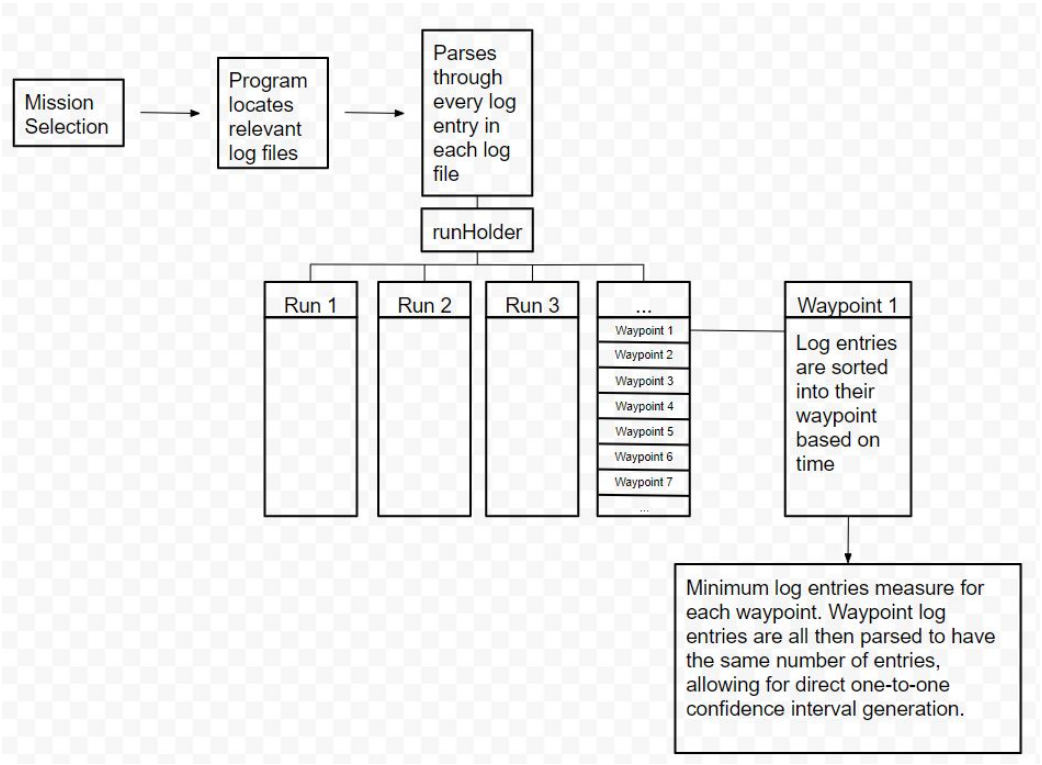

Figure 24: Analysis Tool Data Organization

The data is organized into one large list named “runHolder”. This list is then passed to the ConfidenceInterval.py helper class which performs the confidence interval generation across the log entries.

4.3.4 Confidence Interval Generation

ConfidenceInterval.py has the ability to generate confidence intervals where the sample size is both greater than and less than 30. For this experiment, the maximum sample size utilized (runs) was 10 and a two-tailed 95% confidence interval was generated. The confidence interval was generated using the following equation:

$$95\% \text{ confidence interval} = \bar{x} \pm t(\rho/\sqrt{n}) \text{ where n = the number of entries} \quad (1)$$

The log entry files are parsed so that, for each run, the number of entries between two specific waypoints is the same. For example, the transit from waypoint 1 to waypoint 2 should have the same number of entries for each run prior to finding the confidence intervals on the position metrics. This is accomplished by simply removing, at most, the final two entries for a run for a given waypoint. Nearly all runs had the same number of entries for each waypoint $\pm$ 1-2 entries. As a result, it was decided to simply remove one or two entries to create a confidence interval for every log entry to increase resolution of the UAV flight profile and to develop a finer confidence interval.

4.4 ArduPilot SITL Settings

There are two primary parameters that were experimented with while performing the test flight paths: simulation speedup and wind (an experimental factor). Simulation speedup was utilized during the Survey Mission type as the run time without the aid of the speedup was approximately 40 minutes. With the aid of the speedup feature, the resulting run time was approximately 10 minutes, despite setting the simulation speed to 10 times wall clock speed. This was due to the hardware limitations on the machine utilized to run the experiments. The ArduPilot website doesn't list a maximum possible speedup time so it is unknown how far the speedup parameter can

be tuned, but theoretically the system utilized for simulation purposes is the limiting factor in terms of speedup potential.

Results were compared from speedup runs versus wall clock timed runs. This experiment was performed to assure that the results obtained from speeding up the simulation are consistent with wall clock time simulations. To validate the consistency, the analysis software was executed across multiple runs of each simulation speed test. It was found that the confidence interval generation for both sets of parameters was consistent and the flight paths were nearly identical. As expected, the simulation speedup had no effect on the results obtained from the simulation software. Speedup with a a value of 10 (10 times wall clock speed) was utilized for the Survey mission experimental runs.

Wind was investigated as an experimental factor as it can be set in the simulation environment. It was hypothesized that this parameter would add variation in the flight path and generate a wider confidence interval to more closely emulate real-world conditions. With a wider confidence interval, the exhaustive search within the confidence interval path would have to be examined as discussed in the Methodology. However, it was found that there was almost no variation in the flight path that was taken as a result of the wind parameter. To test this, multiple runs were performed at wall clock time with various wind speeds. The wind speeds utilized were from 0-5 meters per second of wind speed from the 180 degree direction. There was almost no variation in the flight path taken. Figure 25 shows the different flight paths overlaid in Google earth using the .kmz files generated from Mission Planner after the runs. The resulting color in the flight path occurs from the combination of the different flight path runs overlapping one another.

As shown, there is no discernible variation in the flight path as a result of the wind parameter. The total average range on the confidence interval created across each

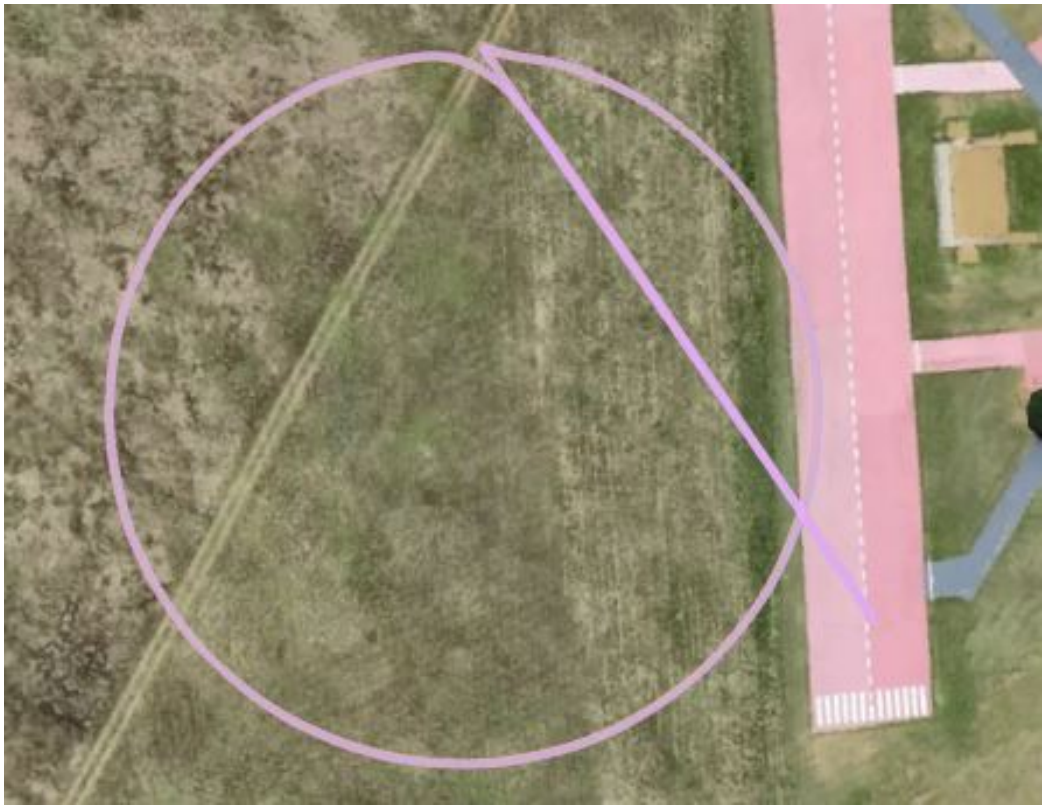

Figure 25: Flight Paths at Various Wind Speeds Overlaid

long entry across 6 runs with varying wind speed results in approximately 0.5 meters. The average range recorded with the simulated wind effect makes little change to the flight path and produces similar values to runs that have the wind parameter disabled. Ideally, the simulator would have increased variation across runs to better simulate the variability of real-life conditions.

4.5 Transit

The transit mission profile consists of takeoff, transit, and landing portions. This flight path can simulate a number of different missions types such as the observation of critical infrastructure entities, or moving a piece of critical hardware from one location to another. This mission profile is the simplest of mission profiles and was simulated with the logic bomb trigger condition along the flight path and outside of

the flight profile area. Figure 26 shows the transit mission flight profile without the logic bomb trigger condition within the flight profile. Figure 27 shows the flight path with the logic bomb trigger condition area indicated by the red box.

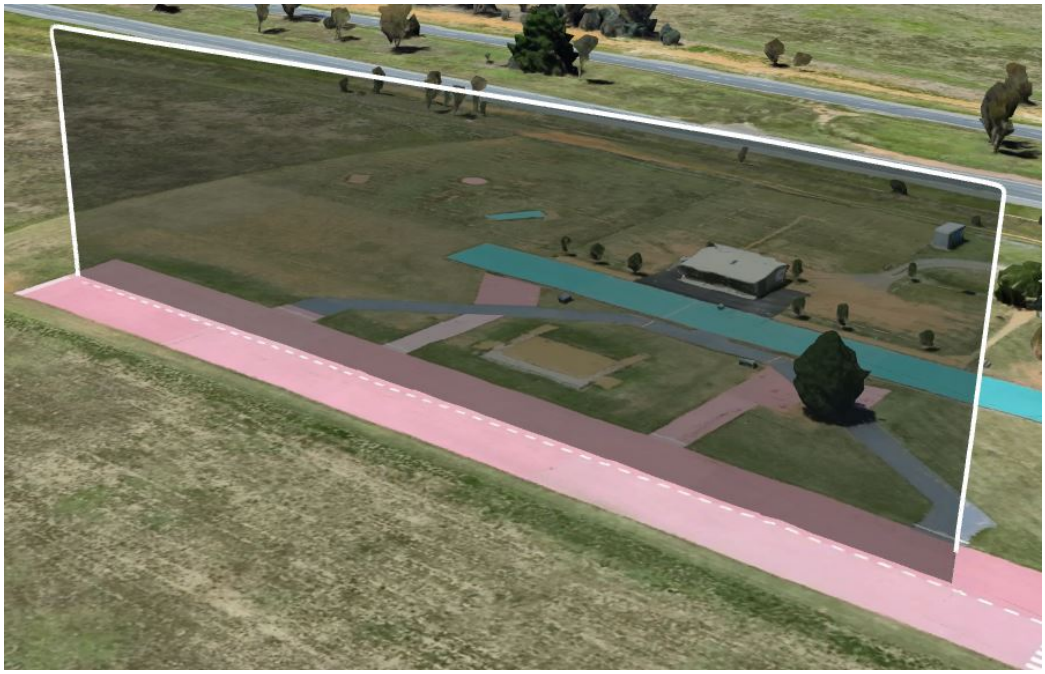

Figure 26: Transit Mission Without Logic Bomb

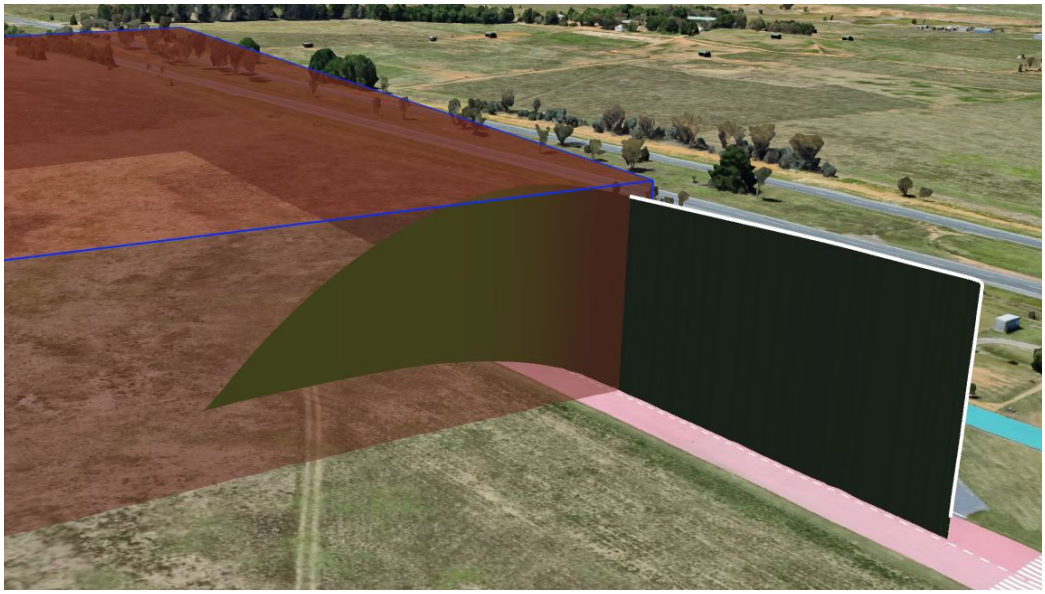

Figure 27: Transit Mission With Logic Bomb

These images are all generated by utilizing the Mission Planner .kmz file conversion tool. The .kmz file is then uploaded into the Google Earth Pro application where the flight path is shown and the red box for the logic bomb trigger zone is created.

Ten flights (runs) were performed with the logic bomb trigger area outside of the designated flight path for this mission profile. This was done to have ten full runs with which to create a confidence interval for each log entry to represent expected flight behavior to a high degree of confidence. Another run is completed with the logic bomb trigger condition within the flight path to demonstrate the effectiveness of the logic bomb. With the flight profile known, it can be determined whether exhaustive search within the confidence interval is required or not. This topic will be discussed in greater depth later in this chapter.

Figure 28 shows the sample output from the analysis script showing the confidence interval creation for the safety corridors for this mission set across 10 runs.

```
The confidence interval for the altitude for entry 542 is 595.1864423915615 through 595.1975576084385
The confidence interval for the latitude for this entry 542 is -35.36356565518928 through -35.36356762481073
The confidence interval for the longitude for this entry 542 is 149.16527120567875 through 149.16527609432123
The distance between the edges of the confidence interval for entry 542 is 0.4945875765244713 meters.

The confidence interval for the altitude for entry 543 is 595.1982446130664 through 595.2077553869337
The confidence interval for the latitude for this entry 543 is -35.363569542091895 through -35.3635713579081
The confidence interval for the longitude for this entry 543 is 149.1652614483118 through 149.16526643168817
The distance between the edges of the confidence interval for entry 543 is 0.495084186619331 meters.

The confidence interval for the altitude for entry 544 is 595.2112866666668 through 595.2187133333333
The confidence interval for the latitude for this entry 544 is -35.363573195694244 through -35.36357492430575
The confidence interval for the longitude for this entry 544 is 149.16525161956142 through 149.16525662043856
The distance between the edges of the confidence interval for entry 544 is 0.4926683053982037 meters.

The confidence interval for the altitude for entry 545 is 595.2238013333334 through 595.2341986666667
The confidence interval for the latitude for this entry 545 is -35.36357668003885 through -35.36357833996115
The confidence interval for the longitude for this entry 545 is 149.16524167144084 through 149.16524672855914
The distance between the edges of the confidence interval for entry 545 is 0.4944659057599791 meters.
```

Figure 28: Sample Program Output

The flight path figures clearly indicate when a logic bomb trigger condition has been reached. As shown in the image, when the UAV reaches the indicated trigger condition zone, the flight path deviates from the planned course and results in a loss of aircraft event.

Trigger event detection at this point consists of messages received from the MAVProxy

tool and examination of the .kmz file output. A command is issued upon logic bomb activation that a loss of power event has been detected. Additionally, readouts from MAVProxy indicate that the logic bomb has been triggered and examination of the .kmz file clearly indicate that the logic bomb has been triggered. This particular logic bomb triggers a loss of thrust message in the MAVProxy command window, indicating that the logic bomb has been triggered. By viewing the .kmz file in Google Earth, it is easy to identify where the logic bomb was triggered as the flight path of the UAV indicates a loss of control and a crash into the ground. At this time, programmatically detecting the logic bomb trigger event isn't implemented. It would be simple to create an algorithm that detects the trigger event for a logic bomb with an obvious and dramatic effect such as this. Simple implementation of detection algorithms for logic bombs with drastic effects would very likely miss more subtle events. For this reason, a programmatic solution hasn't been implemented and long term, robust programmatic detection methodology is proposed in Section 5.4.

After executing the analysis script on the 10 test mission runs, it was found that the range on the generated confidence interval was less than 0.5 meters, indicating that the UAV flight path across multiple runs was nearly identical. This results from the ArduPilot SITL simulator. Efforts were made to create more variation for each mission by introducing other parameters into the simulator but, as discussed, these efforts didn't add more variance to the flight path. A simulator that implemented more advanced weather effects would be needed to cause greater variation between runs. Because the confidence interval is less than the minimum resolution of 1 meter, the methodology does not require a second step of exhaustively testing the safety corridor.

4.6 Circle

The sequence of events for the circle mission are similar to that of the transit mission with the flight beginning with takeoff, a brief transit period to circle's edge, circle flight path, and finally landing. Figures 29 and 30 show this sequence of events from different angles to clearly illustrate the flight profile. Note that Figure 29 is the mission plan execution from the top-down view to clearly illustrate the different phases of flight while Figure 30 right shows the flight path from the side view.

Figure 29: Circle Mission Top Down

Figure 31 shows the UAV and the logic bomb trigger condition is indicated by the red box. When the UAV reaches the logic bomb trigger condition, there is a clear deviation from the designated flight path shown from Figure 30. The UAV is unable to recover from the logic bomb activation and quickly crashes into the ground as indicated in Figure 31. This test demonstrates that the logic bomb predictably

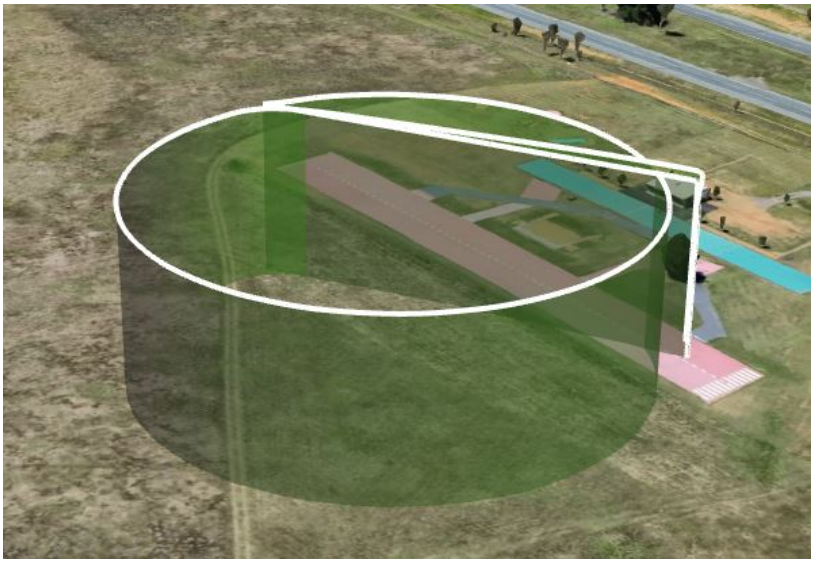

Figure 30: Circle Mission Side View

works on different missions sets with a newly configured trigger condition.

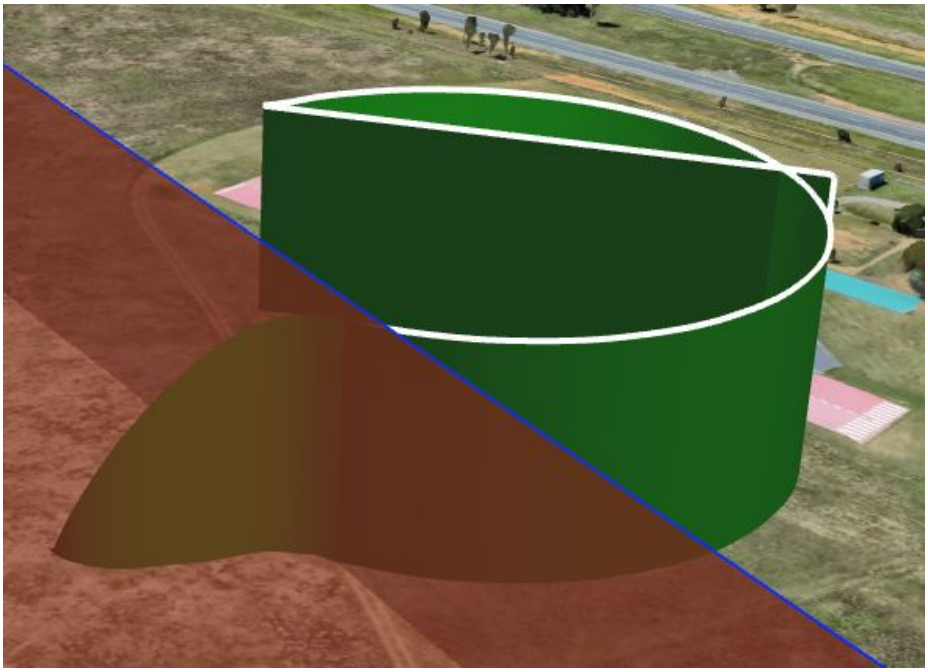

Figure 31: Circle Mission Logic Bomb

The circular mission area has a diameter of $100m$ and area of approximately $7.9km^2$. To reduce complexity, it is assumed to be at a single altitude. Exhaustively

searching then at a $1m$ resolution would require a simulated flight distance of approximately $7,854m$. By utilizing the flight path, the simulated flight distance is reduced to $314m$ per run (the circumference of the circle), with the UAV passing though all points of relevance on this mission. Using 10 runs to generate the confidence interval increases total simulated flight distance to $3,140m$, a reduction of 60% from the total mission area. The small variation observed between runs would have allowed fewer runs to generate an acceptably small confidence interval in this case, but that may not always be true.

4.7 Spline Circle

The spline circle mission is very similar to the circle mission except that multiple laps are taken around the circle. Instead of assuming one altitude as in the last use case, it is increased at a prescribed rate of 5 meters per lap up to an altitude of 25 meters. An added benefit of including this mission is having the ability to include altitude as a third parameter in the logic bomb trigger. As illustrated in the experimental results, the UAV operates normally for multiple laps around the circle until a combined longitude, latitude, and altitude are reached.

Figures 32 and 33 show the mission runs both with and without the logic bomb trigger condition along the flight path. Figure 32 only shows the flight path taken as indicated by the blue line while the yellow line indicates the GPS positional entries on the ground level.

Unlike the circle mission that was only analyzed at one altitude, spline circle comparison accounts for 3 dimensions. The volume of the mission area extending from the ground to max altitude of 25m is $19.6km^3$. Exhaustively testing this full volume at a 1m lat/long resolution and 5m altitude resolution (for fair comparison) results in a simulated flight distance of $39.3km$. Each run developing a confidence

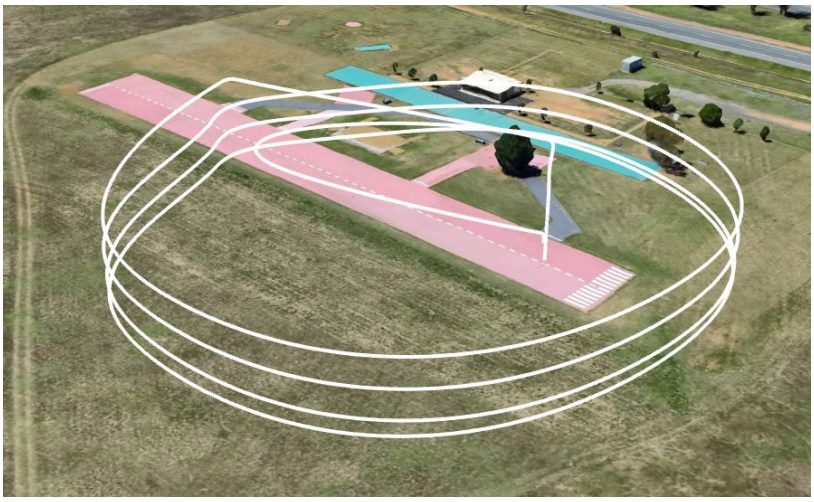

Figure 32: Spline Circle Mission

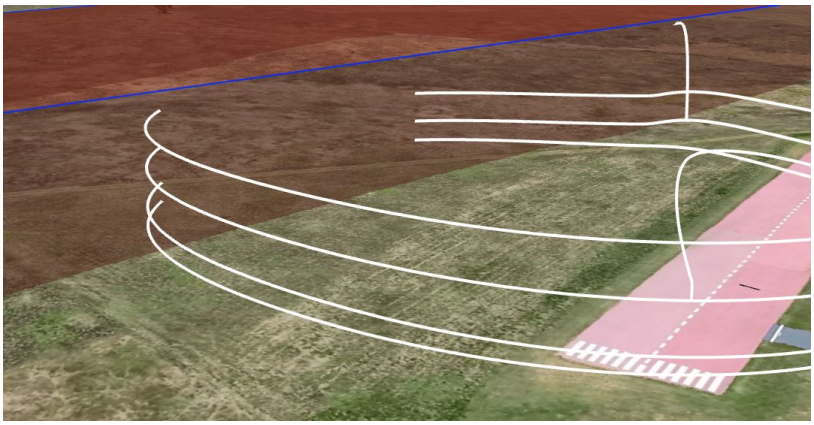

Figure 33: Spline Circle Mission With Logic Bomb

interval requires only $1.57m$, or a total of roughly $15.7km$ for 10 runs. As in the last case, this reduces the simulated flight time significantly by 60%.

4.8 Free Area Survey

The survey mission is relatively close to a brute force detection methodology approach but can still provide a significant benefit over simply brute forcing an entire area of operation when used in conjunction with the reduction techniques described in the other cases. A safety corridor is generated only during the transit, takeoff,

and landing portions of the mission. When the UAV reaches its intended "free" search area, the flight path calls for a sweeping pattern, with distances of 1 meter between sweep, of a very specific area of operation at a given altitude. Utilizing this technique, all theoretical coordinate information that could potentially activate a logic bomb within that area will be input, allowing for dynamic flying within an area during the real world mission. Figure 34 shows the survey flight path, indicated by the yellow lines. This figure was captured in the mission planner window to better illustrate the flight path.

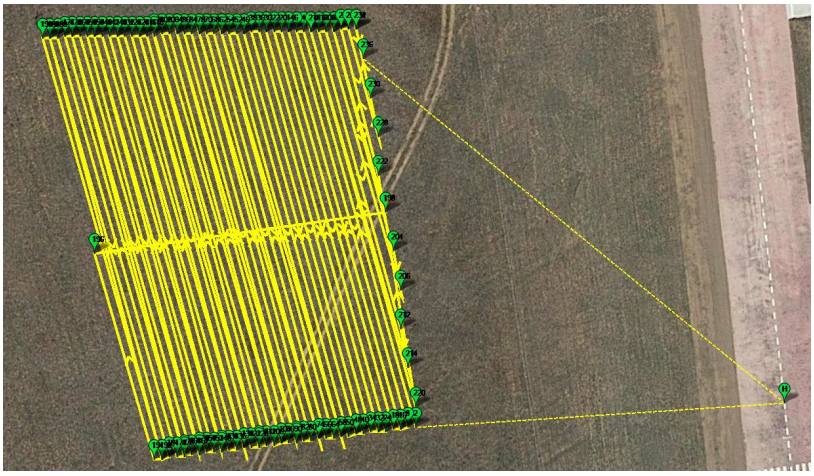

Figure 34: Overview of Survey Mission in Mission Planner Software

Shown in Figure 35 is the survey mission path with with the logic bomb set to have the trigger condition intercept the survey mission plan and the resulting UAV flight path as the logic bomb is activated.

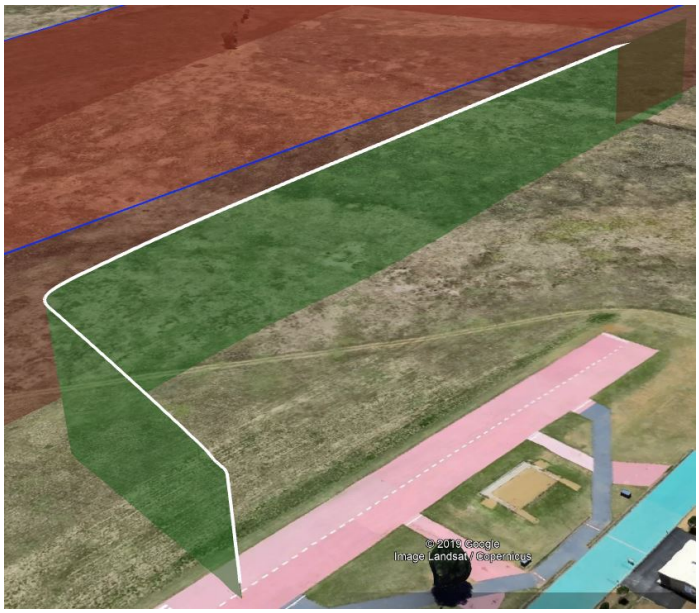

Figure 35: Survey Mission With Logic Bomb

The survey area of operation in this flight path is $188,000m^3$. An area of this size leaves a huge number of inputs that must be validated. However, when compared to what the total operation area is including take-off and transit, the ability to define the total input space to a very small location is extremely beneficial. For example, instead of having to account for all inputs within a given country or township, the technique allows for the input space to be defined only to a given surveillance area. The benefit of performing this test is that it allows planners to operate a UAV dynamically in a given area without fear of potential malicious logic waiting to be activated while still drastically reducing the search space of brute forcing a large area of operation.

4.9 Exhaustive Search

The final portion of this methodology is the exhaustive search. The exhaustive search technique comes into play when a "large" safety corridor, or a safety corridor with a range greater than 1 meter, is generated for a given portion of the flight path. When this situation occurs, the operating area for a UAV during a given segment

of the mission is too large to be validated from simply running the mission plan. The flight path cannot be validated because the assumption that GPS navigational equipment on UAVs is accurate to 1 meter, meaning that UAV logic bomb trigger conditions will be based on a 1 meter increment. To validate the flight path, an exhaustive search pattern is conducted within the segments where the confidence interval is large. This technique validates all potential inputs that the UAV would experience during a given mission.

The exhaustive search pattern is very similar to that of the free area survey conducted in the previous experimental test. Paths with 1 meter of separation are created within the segments of the mission where the safety corridor is large. This technique covers all area within the within the safety corridor within the accuracy range of navigational equipment on the UAV. Implementing targeted use of this technique still greatly reduces the search space relative to the UAV area of operation while validating that the UAV will behave normally during a given mission set.

To illustrate the importance of segmenting the flight and only using the exhaustive search when necessary, assume a 1 kilometer by 1 kilometer area with an operating ceiling of 25 meters. In this scenario, the UAV is assigned to traverse from one end of the are to the other end of the area on a transit mission. Figure 36 shows the simulated mission bordered by the blue line with red filling to indicated the area. The white line indicates the 1 kilometer transit path.

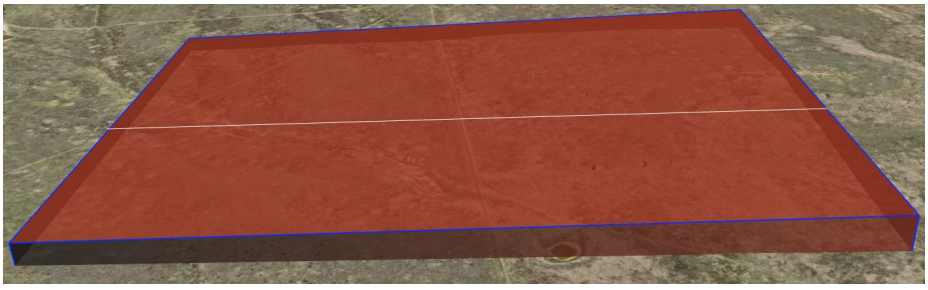

Figure 36: Simulated Mission Area

Using exhaustive search, the volume of the mission area is the input space, totaling to $2.5^7 m^3$. To perform exhaustive search within this region, all potential points within the region must be input to a resolution of 1 meter. This is accomplished by the UAV traversing through the entire area at a resolution of 1 meter with 25 passes at different altitudes. In other words, the entire red area from Figure 36 must be traversed to a resolution of 1 meter. The resulting flight path equates to the volume of the region or 2.5^7 meters. On the other hand, performing a linear search by utilizing a flight path with takeoff, transit and landing phases, with a distance of $1km$, the required distance to be searched is $1050km$, with the $50km$ coming from the takeoff and landing portions. The takeoff and landing portions are accounted for as the UAV performs takeoff and landing in a singular spot for a distance of 25 meters (the mission altitude) but these flight portions are accounted for as inputs into the system. Performing the exhaustive search is necessary at times but is to be performed only when a defined safety corridor must be validated.

The exhaustive search technique hasn't yet been implemented because the deviations in flight paths over the course of multiple runs don't warrant the use of this technique. In all observed test missions, the safety corridors that are generated are smaller than 1 meter. Performing exhaustive search sequences on areas of this size is likely to simply duplicate inputs and reduce the efficiency of the detection methodology. More importantly however, is the fact that navigational equipment present on many small UAVs is unable to accurately determine position at such high fidelity, making it unlikely for adversaries to create triggers so precise. With the current state of the simulation software, it is unnecessary to perform the exhaustive search at this time until simulation tools create an environment where mission runs have variance levels similar to that of the real-world environment.

4.10 Takeaway

SLIVer greatly reduces the potential input space when searching for logic bombs within UAV autopilot code. Using the flight path as a baseline for the input search space allows for a drastic reduction in the potential inputs experienced throughout the course of a mission. Additionally, the analysis software provides the ability to generate a safety corridor encompassing coordinates that the UAV is likely to travel through. Using this safety corridor, more refined searches can be made along the flight path, almost guaranteeing that the UAV autopilot software will not experience different inputs during actual mission execution from simulated mission runs. By receiving all potential inputs throughout the course of simulated missions and observing UAV behavior, planners have the ability to ensure reliable UAV operation on a wide range of missions.

Perhaps the biggest strength of this methodology is that no access to UAV autopilot source code is required. Without access to the source code, the previous option to validating UAV autopilot code is brute force input testing. This methodology provides a means to search for logic bombs on a given flight plan through simulation-based testing and not only validates the flight path, but also the other potential areas the UAV traverses by safety corridor generation.

V. Conclusions

5.1 Overview

The purpose of this Chapter is to summarize the research that has been conducted regarding logic bomb detection in UAVs and the subsequent creation of SLIVer. Through the course of the research, multiple UAV simulation, planning, and command-and-control tools are utilized in addition to the development of logic bomb code, and log file analysis code. The chapter ends by discussing future works projects that would be of great benefit if added to this methodology. These future work projects each add a degree of robustness to the framework proposed to create an encompassing logic bomb detection methodology for UAV autopilot code.

5.2 Summary

The research conducted has the overall goal of detecting the presence of logic bombs within UAV autopilot code. This research identifies the unique challenge of identifying logic bombs within a body of code, further amplified by the assumed condition that there is zero access to the UAV autopilot code. The zero access to source code condition levied upon this research is essential as it more accurately mirrors real-world detection scenarios due to the fact that it is unlikely that a user or planner has access to proprietary source code. This work also creates the first logic bomb for UAV autopilot code. While the logic bomb is very simple, the development of such code was necessary to accurately create the condition that this research seeks to identify. Finally, this research creates a log file analysis script that is utilized to parse the log files gathered from UAV missions to create exhaustive search space regions if necessary. SLIVer is the culmination of these creations and discoveries.

Identification of logic bombs in UAV systems is difficult due to the countless

number of potential inputs that can be associated with a large variety of parameters. The key development from this research is the development of a methodology that circumvents this issue. This methodology utilizes UAV flight profiles for a given mission to narrow the potential input space for logic bomb conditions. While running unverified autopilot code through simulation tool, the potential inputs are limited to just the values and combination of values that would be reasonably expected to be encountered throughout the course of a particular flight path. This method creates a manageable search space for logic bomb trigger conditions that can then be identified through the simulator and from log file examination. Additionally, this methodology proposes a technique to exhaustively search areas that have been identified by analysis code to have large degree of variance. With the aid of the simulation speedup feature, the flight path is able to be validated as "safe" in a much shorter amount of time than the actual time required to perform the mission. Furthermore, the methodology that has been proposed has no reliance on access to the technical details of the UAV or any UAV code.

Another key aspect of this research is the development of the first-known logic bomb that specifically targets UAV autopilot code. While the logic bomb can be considered simple, it demonstrates that UAVs are susceptible to malicious code that relies on a positional trigger condition. As demonstrated in the background section, this type of attack is a viable threat as various United States Government departments have been targeted and exploited from commercially procured GCS software with embedded malicious logic. Therefore, it is imperative that a methodology such as this is utilized to validate UAVs prior to flying missions of critical importance.

An analysis script was developed to parse the UAV log file entries into a manageable format to perform analysis on positional information. While the script is only examining latitude, longitude, and altitude values at this time, it can be fur-

ther configured to examine various different parameter data collected by the UAV throughout the course of the flight. The analysis script goes through each positional log file and creates a confidence interval on each parameter across each entry for multiple runs. This process creates a confidence interval within which the UAV is expected to fly within each and every run of the given mission. With this confidence interval, if the range is less than 1 meter, no exhaustive search of this area needs to be conducted due to the accuracy level of GPS devices compatible with small UAVs. For confidence intervals that span larger than 1 meter, an exhaustive search will need to be performed with a resolution of 1 meter to ensure that all reasonably expected input combinations are experienced by the UAV. Through this technique, the flight path can be validated and it can be determined to a high degree of confidence that the UAV will operate reliably throughout the course of the mission. This technique greatly reduces the amount of inputs of the mission profile to just the locations that the UAV will traverse through on the course of the mission, or in the worst case, exhaustive searches on portions where a high degree of variance is experienced across runs.

5.3 Research Contributions

This work makes multiple contributions towards the goal of detecting logic bombs in UAV autopilot software and to the security of UAVs. A novel methodology is proposed that reduces the potential input space of values that need to be tested to detect the possible input combinations for a logic bomb trigger, circumventing the primary hurdle of logic bomb detection methodologies; search space. Primarily, this research illustrates how the ArduPilot SITL software can be repurposed to create a methodology to validate flight paths and, with the aid of analysis code, detect a logic bomb trigger event. While it is unlikely that this suite of tools would be utilized

for commercial or government entity-grade UAVs, this research shows a clear path forward in developing a methodology for detecting logic bombs within UAV software.

This research also created a logic bomb specifically designed to target UAV autopilot code. At this time, we have been unable to find any other research or example of a logic bomb that targets UAV autopilot code. While logic bombs that would be implemented in a real-world scenario may be more advanced, the logic bomb developed demonstrates a proof of concept and has provided a starting point from which more advanced logic bombs can be created to further test the methodology. The method by which the logic bomb utilized in this research was created can easily be repurposed to look at any number of parameters that are tracked by the UAV during the course of the flight. By creating a logic bomb and proving its effectiveness, the threat of such an attack is validated as an area where further attention is required by users and developers.

Creation of a log file analysis script was accomplished during the course of this research. This script is a key component of this methodology and serves as a starting point towards the development of a robust logic bomb detection methodology for UAV autopilot code. With the addition of more parameters to be examined as well as more advanced analysis techniques on the log files such as machine learning, the analysis script has the potential to detect both nuanced and obvious logic bomb trigger conditions upon completion of simulation runs.

5.4 Future Work

The biggest addition to this research would be to increase the number of parameters that are examined by the analysis scripts code. For example, camera parameters could be measured throughout the course of flight paths to search for strange activity in the camera as the UAV progresses through the mission or the possibility that the

camera could be used to trigger a logic bomb. Increasing the number of parameters that are examined by the analysis software introduces the possibility of finding a wider variety of logic bombs. It is unlikely that all logic bombs deployed on UAV autopilot code would target only the UAV position/flight attitude. SLIVer can be applied to other parameters to find a wider variety of logic bombs and/or malware.

Another area for future work would be to develop more logic bombs for the ArduPilot autopilot software. Using additional exploits and performing more runs would create a database for normal UAV operation and compromised UAV operation. Utilizing these libraries of data, machine learning algorithms could potentially be implemented to automate and increase sensitivity of logic bomb trigger event detection. The development and implementation of machine learning algorithms for logic bomb detection would greatly increase the robustness and the validity of this methodology for real-world critical infrastructure applications due to the enhanced ability to detect a wider range of logic bomb trigger events.

Another potential future addition to SLIVer is the development of a exhaustive search algorithm within the Mission Planner software. This exhaustive search algorithm would add additional waypoints along the flight path to perform the exhaustive search on areas where confidence intervals are greater than 1 meter (or any predefined level of precision). Automating this part of the methodology greatly increases robustness of the methodology in addition to reducing the time to manually create flight paths that perform the exhaustive search.

One final future work project that would provide great benefit to this effort is an automated rerouting algorithm that creates a new flight path for the mission once a logic bomb trigger event is detected. This solution would allow compromised devices to operate despite having a logic bomb lurking within the code. While rerouting the mission profile isn't ideal, it allows for continued operation if the situation calls for

it.

It is envisioned that the addition each of these future works contributions would give this methodology the potential to be extremely robust at detecting logic bombs before they are activated in real-world operations while also providing flight path solutions. Utilization of SLIVer with the additional future works ideas would create a full suite of anti-logic bomb tools that can be utilized before mission runs to validate the flight path and ensure optimal operation throughout. The addition of these future projects would result in a methodology that can be utilized by UAV users to validate their mission plan before takeoff and provide far greatly security for their devices.

5.5 Conclusion

As UAVs continue to become more important parts of society, the requirement for UAV autopilot code validation is essential. Discussed in Chapter II are examples of how UAV code sold by corporations has already been found to have malicious code hidden within the software. Therefore, it is essential that users and planners have the tools necessary to validate their aircraft. It is unlikely that users will ever be guaranteed access to proprietary source code, as such, there is a distinct requirement for a logic bomb detection methodology that doesn't require the source code to function. SLIVer requires no source code by greatly reducing the potential input space that a UAV is likely to encounter on a given mission providing a far better solution that the current best logic bomb detection technique for UAV autopilot code, brute force. The SLIVer techniques gives users and planners a capability they previously weren't afforded: a reliable logic bomb detection and flight path validation technique.

Appendix A. Analysis Script Code

1.1 analysis.py

```
#! python3
__author__="Jake Magness"
__date__="4 June 2019"

from WaypointParameter import WaypointParameter
from ConfidenceInterval import ConfidenceInterval
from enum import Enum
import os, os.path
import fnmatch
import statistics



TIMELOC = 1
LATITUDE = 7
LONGITUDE = 8
ALTITUDE = 9
BASELINE = False #Variable to flag whether or not we are gathering data for baseline
flight path or not.

WPs, GPs, ways = [], [], []
waypointGrouper = [[]] # Stores the gps locations of each waypoint.
nameSuffix = 1
namePrefix = ""
runHolder = [] # Holds a run list that contains waypoint information
```

```
minMaxes = []


def missionType():
userInput = input("Select Mission Type 1-4 (Transit, circle, spline circle, Survey).
if userInput == "1":
return "transit"
elif userInput == "2":
return "circle"
elif userInput == "3":
return "splineCircle"
elif userInput == "4":
return "survey"
elif userInput == "9":
return "newestLog.log"
else:
return None


def keywordParse(namePrefix, nameSuffix):
relevantDataWPs, relevantDataGPS, waypoints = [], [], []
firstTime = None
#Open and parse the file
try:
with open(namePrefix + str(nameSuffix).zfill(2) + ".log") as file:
#with open('newestLog.log') as file:
line = file.read().splitlines()
line = [line[x] for x in range(1, len(line), 1)]
```

```
#Scan for the key words that we need
for x in range(len(line)):
tmp = line[x][0] + line[x][1] + line[x][2]
if tmp == 'GPS':
relevantDataGPS.append(line[x])
if tmp == 'CMD':
tmpParse = line[x].split(",")
if firstTime == None:
firstTime = tmpParse[TIMELOC].strip() #This gets the value for the time.
waypoints.append(line[x]) # Add the first value to the waypoints array
elif tmpParse[TIMELOC].strip() == firstTime: #Append to the waypoints array
waypoints.append(line[x])
else:
relevantDataWPs.append(line[x])
except:
return None, None, None

return relevantDataWPs, relevantDataGPS, waypoints

def writeToFile(relevantDataWPs, relevantDataGPS, waypoints, namePrefix, nameSuffix):

with open ('short' + namePrefix + str(nameSuffix).zfill(2) + '.log', 'w+') as file:
#with open ('newShortLog.log', 'w+') as file:
for i in range(len(relevantDataGPS)):
file.write(relevantDataGPS[i] + "\n")
```

```
tmp = relevantDataGPS[i].split(",")
relevantDataGPS[i] = tmp
file.write("Below are the waypoints. \n");

for i in range(len(waypoints)):
file.write(waypoints[i] + "\n")
tmp = waypoints[i].split(",")
waypoints[i] = tmp
file.write("Below are the waypoint locations reached by UAV" + "\n")
#Waypoint information written into log as follows:
#These waypoints are sent to the device all at once, indicated by the time stamp.
They are then sent again once reached, as noted by time
#Follow the following format: MSG Type, Time, # Total waypoints, Waypoint #, Unknown,
Unknown, Unknown, Unknown, Unknown, Lat, Long, Alt, Unknown.
for i in range(len(relevantDataWPs)):
file.write(relevantDataWPs[i] + "\n")
tmp = relevantDataWPs[i].split(",")
relevantDataWPs[i] = tmp
file.write("Below are the GPS coords paired in respective waypoints" + "\n")

def minMaxUpdate(relevantDataWPs, relevantDataGPS, waypoints, namePrefix,
        nameSuffix):
#This function needs log files that have completed all the waypoints
or an error is thrown.
waypointTracker = 0
waypointLocations = [[] for i in range(len(waypoints))] #Needs to be this size to
```

```
contain all WP information.

for x in range(len(relevantDataGPS)):
if waypointTracker < len(waypointLocations)-1:
if int(relevantDataGPS[x][TIMELOC].strip()) <
int(relevantDataWPs[waypointTracker][TIMELOC].strip()):
waypointLocations[waypointTracker].append(relevantDataGPS[x])
else:
waypointTracker += 1
waypointLocations[waypointTracker].append(relevantDataGPS[x])
else:
waypointLocations[waypointTracker].append(relevantDataGPS[x])

return waypointLocations

def createBaselineFlightpath(namePrefix):
numberOfFiles = len(fnmatch.filter(os.listdir(), namePrefix + '*' + ".log"))
#This creates a file string like file01.txt. This will have to be changed based on
formatting but we know how to zero pad here now.
#print(namePrefix + str(nameSuffix).zfill(2) + ".txt")
#print(len(fnmatch.filter(os.listdir(), namePrefix + '??' + ".log")))
if numberOfFiles < 2:
return False
else:
return numberOfFiles
```

```
def findMinMax(wpLocs):
# This function is utilized to characterize a tested flight profile.
This will gather the minimum and maximum values from a flight path
and it stores them in the parameterHolder object as a WaypointParameter
object. The WaypointParameter object simply stores the mins and
max values for identified parameters.
minAlt = float(99999.0)
maxAlt = float(-99999.0)
minLat = float(99999.0)
maxLat = float(-99999.0)
minLng = float(99999.0)
maxLng = float(-99999.0)
minMaxes = []
average = 0
altSum = 0
latSum = 0
lngSum = 0
#wpLocs[RUN][WAYPOINT][LOG ENTRY][PARAMETER]
for x in range(len(wpLocs)):
parameterHolder = []
for i in range(len(wpLocs[x])):
for j in range(len(wpLocs[x][i])):
altSum += float(wpLocs[x][i][j][ALTITUDE].strip())
latSum += float(wpLocs[x][i][j][LATITUDE].strip())
lngSum += float(wpLocs[x][i][j][LONGITUDE].strip())
```

```
if(float(wpLocs[x][i][j][ALTITUDE].strip()) < minAlt):
minAlt = float(wpLocs[x][i][j][ALTITUDE].strip())
if(float(wpLocs[x][i][j][ALTITUDE].strip()) > maxAlt):
maxAlt = float(wpLocs[x][i][j][ALTITUDE].strip())
if(float(wpLocs[x][i][j][LATITUDE].strip()) < minLat):
minLat = float(wpLocs[x][i][j][LATITUDE].strip())
if(float(wpLocs[x][i][j][LATITUDE].strip()) > maxLat):
maxLat = float(wpLocs[x][i][j][LATITUDE].strip())
if(float(wpLocs[x][i][j][LONGITUDE].strip()) < minLng):
minLng = float(wpLocs[x][i][j][LONGITUDE].strip())
if(float(wpLocs[x][i][j][LONGITUDE].strip()) > maxLng):
maxLng = float(wpLocs[x][i][j][LONGITUDE].strip())
parameterHolder.append(WaypointParameter(minAlt, maxAlt, minLat, maxLat,
minLng, maxLng))
parameterHolder[i].updateAverage(altSum/(j+1), ALTITUDE)
parameterHolder[i].updateAverage(latSum/(j+1), LATITUDE)
parameterHolder[i].updateAverage(lngSum/(j+1), LONGITUDE)
altSum, latSum, lngSum = 0, 0, 0
# Reset values for new waypoint. Need to send out the waypointParameters and
send them back out of the function.
minAlt = float(99999.0)
maxAlt = float(-99999.0)
minLat = float(99999.0)
maxLat = float(-99999.0)
minLng = float(99999.0)
maxLng = float(-99999.0)
```

```
minMaxes.append(parameterHolder)
# for i in range (len(minMaxes[4])):
# print("Waypoint: ", i)
# minMaxes[4][i].printFunction()

def findCI(wpLocs):
#From the wpLocs file, we gather just the alts, lats and longs.
We perform mean and std deviation on these parameters.
The results of these calculations are then passed into the a confidenceInterval
object. The confidence interval object stores data about a specific
confidence interval for one waypoint. These waypoint confidence intervals
are stored in the confInt object and this object can be compared to a baseline
flight profile. The baseline flight profile will have the minimums and
maximums compared to the confidence interval that has been created.
sampleAlts = [[] for i in range(len(wpLocs[0]))]
sampleLats = [[] for i in range(len(wpLocs[0]))]
sampleLngs = [[] for i in range(len(wpLocs[0]))]
confInt = []
for x in range(len(wpLocs)):
for i in range(len(wpLocs[x])):
for j in range(len(wpLocs[x][i])):
sampleAlts[i].append(float(wpLocs[x][i][j][ALTITUDE].strip()))
sampleLats[i].append(float(wpLocs[x][i][j][LATITUDE].strip()))
sampleLngs[i].append(float(wpLocs[x][i][j][LONGITUDE].strip()))
```

```
for i in range(len(sampleAlts)):
altSampleMean = statistics.mean(sampleAlts[i])
altSampleStdev = statistics.stdev(sampleAlts[i])
latSampleMean = statistics.mean(sampleLats[i])
latSampleStdev = statistics.stdev(sampleLats[i])
lngSampleMean = statistics.mean(sampleLngs[i])
lngSampleStdev = statistics.stdev(sampleLngs[i])
print("Waypoint ", i)
confInt.append(ConfidenceInterval([altSampleMean, latSampleMean, lngSampleMean],
[altSampleStdev, latSampleStdev, lngSampleStdev], len(sampleAlts[i])+1))

def tFindCI(wpLocs):

confInt = [[] for i in range(len(wpLocs[0]))] #Array that is 7 long
for each waypoint.

for i in range(1, len(wpLocs[0])): # Leaving out takeoff for now.
for j in range(len(wpLocs[0][i])):
altitude = []
latitude = []
longitude = []
for x in range(len(wpLocs)):
altitude.append(float(wpLocs[x][i][j][ALTITUDE]))
latitude.append(float(wpLocs[x][i][j][LATITUDE]))
longitude.append(float(wpLocs[x][i][j][LONGITUDE]))
#print("Run: ", x, "Waypoint: ", i, "Log: ", j, "Long: ", wpLocs[x][i][j][LONGITUDE])
```

```
altSampleMean = statistics.mean(altitude)
altSampleStdev = statistics.stdev(altitude)
latSampleMean = statistics.mean(latitude)
latSampleStdev = statistics.stdev(latitude)
lngSampleMean = statistics.mean(longitude)
lngSampleStdev = statistics.stdev(longitude)
confInt[i].append(ConfidenceInterval([altSampleMean, latSampleMean,
lngSampleMean], [altSampleStdev, latSampleStdev, lngSampleStdev], x+1))


def compareFlightProfiles(baselineCI, testProfile):
if (len(baselineCI) != len(testPofile)):
print("Flight profiles don't contain the same number of waypoints.")
else:
for i in range(len(baselineCI)):
if(testProfile[i].getAltMin() < baselineCI[i].getAltLower):
return False

def entryCounter(wpLocs):
entryMins = []
for i in range(1, len(wpLocs[0])): # This ignores takeoff sequence for entry mins
and gathers number of entries in each waypoint.
entryMins.append(len(wpLocs[0][i]))

for i in range(1, len(wpLocs)): # We ignore the first run here because we are setting
the default mins to the first run.
```

```
for x in range(1, len(wpLocs[i])): # Since entry mins has the first waypoint
after takeoff indexed at 0, we start there and have to leave off edge
print("Run: ", i, "Waypoint: ", x, "Length: ", len(wpLocs[i][x]))

if (entryMins[x-1]) > (len(wpLocs[i][x])):
entryMins[x-1] = len(wpLocs[i][x])
return entryMins

def entryPrune(wpLocs):
mins = entryCounter(wpLocs)
for i in range(len(wpLocs)):
for x in range(1, len(wpLocs[i])):
#if (len(wpLocs[i][x]) > mins[x-1]):
while len(wpLocs[i][x]) > mins[x-1]:
#print("Length: ", len(wpLocs[i][x]), "Mins: ", mins[x-1])
print("Run:", i, "Waypoint:", x, "Time:", wpLocs[i][x][-1][TIMELOC],
"Deleted info:", wpLocs[i][x][-1][ALTITUDE],
wpLocs[i][x][-1][LATITUDE], wpLocs[i][x][-1][LONGITUDE])
del wpLocs[i][x][-1] # This deletes the last element, extending
outside of this funciton.


def entryCheck(wpLocs):
for i in range(len(wpLocs[0])):
for x in range(len(wpLocs)):
print("Length run", x, ": ", len(wpLocs[x][i]))
```

```
def main():
namePrefix = missionType()
nameSuffix = 1

# Calls baseline flight path
baselineFlightPath = createBaselineFlightpath(namePrefix)
if baselineFlightPath == False:
print("Not enough files to create an accurate baseline flight path.")
else:
for x in range (int(baselineFlightPath)):
WPs, GPs, ways = keywordParse(namePrefix, nameSuffix)
if (WPs == None):
print("The log file name: " + namePrefix + str(nameSuffix).zfill(2) + " is not named
else:
writeToFile(WPs, GPs, ways, namePrefix, nameSuffix)
waypointGrouper = minMaxUpdate(WPs, GPs, ways, namePrefix, nameSuffix)
if(waypointGrouper != None):
runHolder.append(waypointGrouper)
nameSuffix += 1
#minMaxes = findMinMax(runHolder) # This function won't really be used for the
baseline flight path but for the test flight path.
#findCI(runHolder)
entryPrune(runHolder)
entryCheck(runHolder)
tFindCI(runHolder)
```

```
if __name__ == "__main__":
    """ This is executed when run from the command line """
    main()
```

1.2 ConfidenceInterval.py

```
#! python3
__author__="Jake Magness"
__date__="28 October 2019"
import statistics
import math
from math import radians, cos, sin, asin, sqrt

count  = 0
average = 0

class ConfidenceInterval:

def __init__(self, mean, stdev, samples):
self.altLower = 0
self.altUpper = 0
self.latLower = 0
self.latUpper = 0
self.lngLower = 0
self.lngUpper = 0
self.tCalculateInterval(mean, stdev, samples)
```

```
def getAltLower(self):
return self.altLower

def getAltUpper(self):
return self.altUpper

def getLatLower(self):
return self.latLower

def getLatUpper(self):
return self.latUpper

def getLngLower(self):
return self.lngLower

def getLngUpper(self):
return self.lngUpper

def printFunction(self):
global count
global average

print("The confidence interval for the altitude for entry", count, "is",
self.altLower, "through", self.altUpper)
```

```
print("The confidence interval for the latitude for this entry", count, "is",
self.latLower, "through", self.latUpper)
print("The confidence interval for the longitude for this entry", count, "is",
self.lngLower, "through", self.lngUpper)
print("The distance between the edges of the confidence interval for entry",
count, "is", self.findDistance(self.latLower, self.lngLower, self.latUpper,
self.lngUpper), "meters.")
print("")
count = count + 1
average = (average + self.findDistance(self.latLower, self.lngLower,
self.latUpper, self.lngUpper))
print("average: ", average/count)

def calculateInterval(self, mean, stdev, samples):
z = 1.96

if mean[0] > 0:
self.altLower = mean[0] - (z * (stdev[0]/math.sqrt(samples)))
self.altUpper = mean[0] + (z * (stdev[0]/math.sqrt(samples)))
else:
self.altLower = mean[0] + (z * (stdev[0]/math.sqrt(samples)))
self.altUpper = mean[0] - (z * (stdev[0]/math.sqrt(samples)))
if mean[1] > 0:
self.latLower = mean[1] - (z * (stdev[1]/math.sqrt(samples)))
self.latUpper = mean[1] + (z * (stdev[1]/math.sqrt(samples)))
else:
```

```
self.latLower = mean[1] + (z * (stdev[1]/math.sqrt(samples)))
self.latUpper = mean[1] - (z * (stdev[1]/math.sqrt(samples)))
if mean[2] > 0:
self.lngLower = mean[2] - (z * (stdev[2]/math.sqrt(samples)))
self.lngUpper = mean[2] + (z * (stdev[2]/math.sqrt(samples)))
else:
self.lngLower = mean[2] + (z * (stdev[2]/math.sqrt(samples)))
self.lngUpper = mean[2] - (z * (stdev[2]/math.sqrt(samples)))
count = self.printFunction()

def tCalculateInterval(self, mean, stdev, samples):
tValues95 = [12.706, 4.303, 3.182, 2.776, 2.571, 2.447, 2.365, 2.306, 2.262, 2.228,
2.201, 2.179, 2.160, 2.145, 2.131, 2.120, 2.110, 2.101, 2.093, 2.086, 2.080, 2.074,
2.069, 2.064, 2.060, 2.056, 2.052, 2.048, 2.045, 2.042]

if mean[0] > 0:
self.altLower = mean[0] - (tValues95[samples-1] * (stdev[0]/math.sqrt(samples)))
self.altUpper = mean[0] + (tValues95[samples-1] * (stdev[0]/math.sqrt(samples)))
else:
self.altLower = mean[0] + (tValues95[samples-1] * (stdev[0]/math.sqrt(samples)))
self.altUpper = mean[0] - (tValues95[samples-1] * (stdev[0]/math.sqrt(samples)))
if mean[1] > 0:
self.latLower = mean[1] - (tValues95[samples-1] * (stdev[1]/math.sqrt(samples)))
self.latUpper = mean[1] + (tValues95[samples-1] * (stdev[1]/math.sqrt(samples)))
else:
self.latLower = mean[1] + (tValues95[samples-1] * (stdev[1]/math.sqrt(samples)))
```

```
self.latUpper = mean[1] - (tValues95[samples-1] * (stdev[1]/math.sqrt(samples)))
if mean[2] > 0:
self.lngLower = mean[2] - (tValues95[samples-1] * (stdev[2]/math.sqrt(samples)))
self.lngUpper = mean[2] + (tValues95[samples-1] * (stdev[2]/math.sqrt(samples)))
else:
self.lngLower = mean[2] + (tValues95[samples-1] * (stdev[2]/math.sqrt(samples)))
self.lngUpper = mean[2] - (tValues95[samples-1] * (stdev[2]/math.sqrt(samples)))
self.printFunction()


def findDistance(self, lat1, long1, lat2, long2):
#Found at stackoverflow.com/questions/4913349/haversine-formula-in-python-
bearing-and-distnace-between-two-gps-points
radius = 6372.8 #Radius of earth in kilometers

lat = radians(lat2 - lat1)
long = radians(long2 - long1)
lat1 = radians(lat1)
lat2 = radians(lat2)

a = sin(lat/2)**2 + cos(lat1)*cos(lat2)*sin(long/2)**2
c = 2*asin(sqrt(a))

return radius*c*1000 #Multiply by 1000 to get this value into meters.
```

Bibliography

1. *Studying Analysing The Current Activities in The Field of UAV*, 2007.

2. C. Barrado, R. Messeguer, J. Lopez, E. Pastor, E. Santamaria, and P. Royo. Wildfire monitoring using a mixed air-ground mobile network. *IEEE Pervasive Computing*, 9(4):24–32, October 2010.

3. Andrew Liptak. The faa says the commercial drone market could triple in size by 2023. 4 May 2019.

4. Paul Mozur. Drone Maker D.J.I. May Be Sending Data to China, U.S. Officials Say. *New York Times*, 2017.

5. Smith. Leaked DHS memo accuses drone maker DJI of spying for China. *CSO*, 2017.

6. Doina Bein Bharat B Madan, Manoj Banik. Securing unmanned autonomous systems from cyber threats. *Journal of Defense Modeling and Simulation: Applications, Methodology, Technology*, 16(2), 2019.

7. Raghavendra Sriram Dariusz Mikulski Frank L Lewis Chaitanya Rani, Hamidreza Modares. Securing unmanned autonomous systems against cyber-physical attacks. *Journal of Defense Modeling and Simulation: Applications, Methodology, Technology*, 13(3), 2016.

8. Kamesh Namuduri and Damien Sauveron. *Safety, Security, and Privacy Aspects in UAV Networks*, page 160176. Cambridge University Press, 2017.

9. Vijay K. Devabhaktuni Mansoor Alam Ahmad Y. Javaid, Weiqing Sun. Cyber Security Threat Analysis and Modeling of an Unmanned Aerial Vehicle System.

IEEE International Conference on Technologies for Homeland Security, pages 585–590, 2012. ISBN: 9781467327084.

10. Ricardo de O. Schmidt Nils Miro Rodday and Aiko Pras. Exploring Security Vulnerabilities of Unmanned Aerial Vehicles. *IEEE/IFIP Network Operations and Management Symposium*, pages 993–994, 2016. ISBN: 9781509002238.

11. M. Hooper, Y. Tian, R. Zhou, B. Cao, A. P. Lauf, L. Watkins, W. H. Robinson, and W. Alexis. Securing commercial wifi-based uavs from common security attacks. In *MILCOM 2016 - 2016 IEEE Military Communications Conference*, pages 1213–1218, Nov 2016.

12. Jeffrey M. Sullivan. Evolution or revolution. *IEEE Technology and Society Magazine*, Fall 2006:43–49, 2006. ISBN:.

13. J. Villasenor. drones and the future of domestic aviation [point of view]. *Proceedings of the IEEE*, 102(3):235–238, March 2014.

14. A. Giyenko and Y. I. Cho. Intelligent uav in smart cities using iot. In *2016 16th International Conference on Control, Automation and Systems (ICCAS)*, pages 207–210, Oct 2016.

15. Divya Joshi. Exploring the latest drone technology for commercial, industrial and military drone uses, 2017. [Online; accessed August 1, 2019].

16. Jack Brown. Types of military drones: The best technology available today, 2017. [Online; accessed August 1, 2019].

17. HQ USAF. Usaf unmanned aircraft systems flight plan 2009-2047, 2009. [Online; accessed July 31, 2019].

18. Pleban J Band R Creutzburg R. Hacking and securing the AR.Drone 2.0 quadcopter: Investigations for improving the security of a toy. *Mobile Devices and Multimedia: Enabling Technologies, Algorithms, and Applications*, 9030:90300L, 2014. DOI: 10.1117/12.2044868.

19. K. Hartmann and K. Giles. Uav exploitation: A new domain for cyber power. In *2016 8th International Conference on Cyber Conflict (CyCon)*, pages 205–221, May 2016.

20. Noah Shachtman. Exclusive: Computer virus hits u.s. drone fleet, 7 October 2011. [Online; accessed August 5, 2019].

21. Ravi Kandala. What are logic bombs and how to detect them, 29 May 2013. [Online; accessed August 5, 2019].

22. Kasey Wehrum. Technology: When it workers attack, 1 April 2009. [Online; accessed August 5, 2019].

23. Julie Bort. A contract programmer faces 10 years in jail for inserting a 'logic bomb' into a spreadsheet that caused the company to keep rehiring him, 23 July 2019. [Online; accessed August 5, 2019].

24. Kim Zetter. Logic bomb set off south korea cyberattack, 3 March 2013. [Online; accessed August 5, 2019].

25. Hira Agrawal, J Alberi, L Bahler, J Micallef, A Virodov, M Magenheimer, S Snyder, V Debroy, and E Wong. Detecting hidden logic bombs in critical infrastructure software. *7th International Conference on Information Warfare and Security, ICIW 2012*, pages 1–11, 01 2012.

26. H. Agrawal, J. Alberi, L. Bahler, W. Conner, J. Micallef, A. Virodov, and R. S. Shane. Preventing insider malware threats using program analysis techniques.

In *2010 - MILCOM 2010 MILITARY COMMUNICATIONS CONFERENCE*, pages 936–941, Oct 2010.

27. Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. Triggerscope: Towards detecting logic bombs in android applications. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 377–396, May 2016.

28. Hira Agrawal, James Alberi, Lisa Bahler, William Conner, Josephine Micallef, Alexandr Virodov, and R. Snyder Shane. Preventing insider malware threats using program analysis techniques. *2010 - Milcom 2010 Military Communications Conference*, 2010.

29. Farmeena Khan Mohd. Ehmer Khan. A comparative study of white box, black box and grey box testing techniques. *International Journal of Advanced Computer Science and Applications*, pages 12–15, 2012.

30. Jagruthi Dondeti Srinivas Nidhra. Black box and white box testing techniques - a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, pages 29–50, 06 2012.

31. Mohd. Ehmer Khan. Different approaches to black box testing technique for finding errors. *International Journal of Software Engineering & Applications (IJSEA)*, pages 12–15, 10 2011.

32. guru99.com. Boundary value analysis & equivalence partitioning with examples. [Online; accessed August 15, 2019].

33. ArduPilot. Ardupilot about. [Online; accessed August 11, 2019].

34. ArduPilot. Ardupilot sitl. [Online; accessed August 11, 2019].

35. MAVProxy. Mavproxy. [Online; accessed August 12, 2019].

36. MissionPlanner. Missionplanner. [Online; accessed August 12, 2019].

37. ArduPilot. Setting up the waf build environment on windows using cygwin. [Online; accessed August 11, 2019].

38. R. Jain. The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling., 1991.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
26 Mar 2020	Master's Thesis	Sept 2018 — Mar 2020

4. TITLE AND SUBTITLE

SLIVer: Simulation-Based Logic Bomb Identification/Verification

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Magness, Jake M, 2nd Lt

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-MS-20-M-039

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory, AFMC
Attn: Maj Benjamin Bruckman
2250 Avionics Circle
Wright-Patterson AFB, OH 45433-7765
benjamin.bruckman1@us.af.mil DSN: 713-4252

10. SPONSOR/MONITOR'S ACRONYM(S)

AFRL/RYWA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This research introduces SLIVer, a Simulation-based Logic Bomb Identification/Verification methodology, for finding logic bombs hidden within Unmanned Aerial Vehicle (UAV) autopilot code without having access to the device source code. Effectiveness is demonstrated by executing a series of test missions within a high-fidelity software-in-the-loop (SITL) simulator. In the event that a logic bomb is not detected, this methodology defines safe operating areas for UAVs to ensure to a high degree of confidence the UAV operates normally on the defined flight plan.

15. SUBJECT TERMS

Unmanned Aerial Vehicle, Logic Bomb, Logic Bomb Detection, ArduPilot, Mission Planner, MAVProxy, Software-In-The-Loop, UAV, UAV Security, black box

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Lt Col Patrick J. Sweeney, AFIT/ENG
U	U	U	UU	118	19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636, x4757; patrick.sweeney@afit.edu