NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

FEASIBILITY OF AN EXTENDED-DURATION AERIAL PLATFORM USING AUTONOMOUS MULTI-ROTOR VEHICLE SWAPPING AND BATTERY MANAGEMENT

by

Alexander G. Williams

December 2017

Thesis Advisor: Oleg A. Yakimenko
Second Reader: Brian S. Bingham

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** December 2017	**3. REPORT TYPE AND DATES COVERED** Master's thesis	
4. TITLE AND SUBTITLE FEASIBILITY OF AN EXTENDED-DURATION AERIAL PLATFORM USING AUTONOMOUS MULTI-ROTOR VEHICLE SWAPPING AND BATTERY MANAGEMENT			**5. FUNDING NUMBERS**
6. AUTHOR(S) Alexander G. Williams			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB number ____N/A____.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

Although the U.S. military relies increasingly on autonomous unmanned systems, such systems cannot surveil for long periods of time. For better intelligence collection and communication, an extended-duration aerial platform is required. This thesis focuses on the development and evaluation of a multi-rotor persistent system to provide a longer-duration system using vehicle swapping and intelligent battery management. A proof-of-concept system was built using three quadcopters, a single wireless network router and a laptop to execute code. The system monitored vehicle battery life; when the limit was exceeded, the next vehicle was launched and swapped in its place autonomously. This cycle continued as long as fresh batteries were available. The system provided 54 minutes of platform coverage, more than five times the duration of the single quadcopter. Testing found the system to be feasible and suggests how autonomous capabilities can be extended with persistent platforms. The system is easily scalable for increased survivability and coverage. Battery life and recharging capability proved to be key limitations of the system. However, if the rate at which fully charged batteries are available exceeds the rate at which they are expended, the system can operate until all individual quadcopters mechanically fail.

14. SUBJECT TERMS multi-rotor UAS, persistent coverage, battery management			**15. NUMBER OF PAGES** 93
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

FEASIBILITY OF AN EXTENDED-DURATION AERIAL PLATFORM USING AUTONOMOUS MULTI-ROTOR VEHICLE SWAPPING AND BATTERY MANAGEMENT

Alexander G. Williams
Lieutenant Commander, United States Navy
B.S., North Carolina State University, 2006

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN SYSTEMS ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December 2017

Approved by: Oleg A. Yakimenko
Thesis Advisor

Brian S. Bingham
Second Reader

Ronald E. Giachetti
Chair, Department of Systems Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Although the U.S. military relies increasingly on autonomous unmanned systems, such systems cannot surveil for long periods of time. For better intelligence collection and communication, an extended-duration aerial platform is required. This thesis focuses on the development and evaluation of a multi-rotor persistent system to provide a longer-duration system using vehicle swapping and intelligent battery management. A proof-of-concept system was built using three quadcopters, a single wireless network router and a laptop to execute code. The system monitored vehicle battery life; when the limit was exceeded, the next vehicle was launched and swapped in its place autonomously. This cycle continued as long as fresh batteries were available. The system provided 54 minutes of platform coverage, more than five times the duration of the single quadcopter. Testing found the system to be feasible and suggests how autonomous capabilities can be extended with persistent platforms. The system is easily scalable for increased survivability and coverage. Battery life and recharging capability proved to be key limitations of the system. However, if the rate at which fully charged batteries are available exceeds the rate at which they are expended, the system can operate until all individual quadcopters mechanically fail.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

3DR	3D Robotics, a commercial unmanned aerial vehicle manufacturer
CLI	Command Line Tool
COI	critical operational issue
COTS	commercial off-the-shelf
DARPA	Defense Advanced Research Projects Agency
DOD	Department of Defense
GCS	ground control station
GPS	Global Positioning System
IR	infrared
ISR	intelligence, surveillance, and reconnaissance
MANPADS	man-portable air-defense system
MOE	measure of effectiveness
MOP	measure of performance
NPS	Naval Postgraduate School
RCS	radar cross-section
SSH	Service Set Shell
SSID	Service Set Identifier
SUAS	small unmanned aerial system
UAS	unmanned aerial system
UV	ultraviolet

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

Unmanned systems continue to be at the forefront of development for the U.S. military and the defense industrial base. Unmanned systems operate more and more with greater autonomy. A key aspect of enhancing autonomy is providing persistent systems (Defense Science Board 2012). Fuel limits the operational time of many systems characterized as persistent. This thesis focuses on the development and evaluation of a multi-rotor persistent system to provide a longer-duration system using vehicle swapping and intelligent battery management.

The system developed consisted of three commercially available 3DR Solo quadcopters, shown in Figure 1, a wireless network router, and a laptop to execute Python code. System operation was autonomous, swapping vehicles, as shown in Figure 1, to maintain an airborne vehicle in the loiter area based on battery health. There are three vehicles in this system. The first vehicle, shown in red, is replaced by the second vehicle, shown in blue, while the third vehicle, shown in black, remains on the ground next in line for tasking.

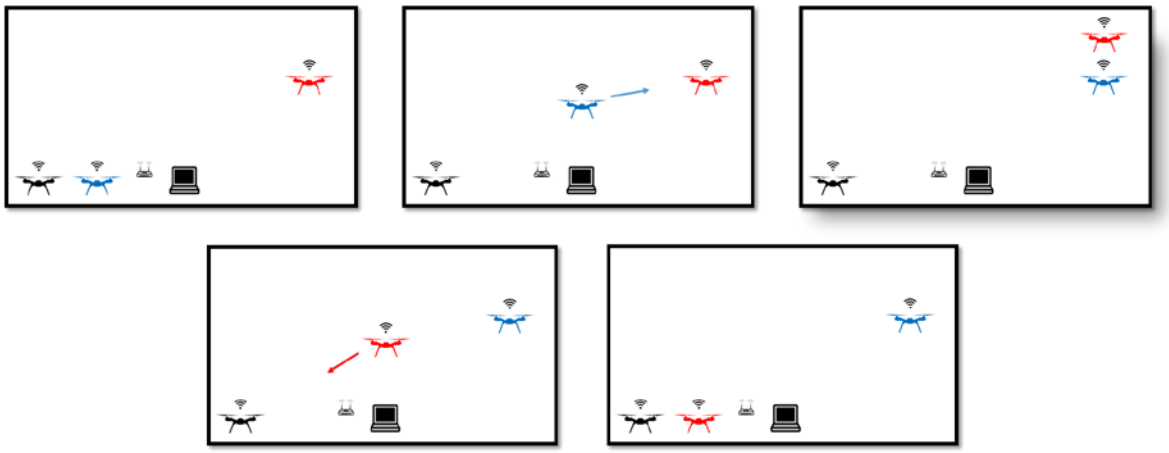


Figure 1. Operational Concept Diagram

Based on analysis and assessment of the field test results, the persistent system is feasible. The system conducted four vehicle swaps and maintained a vehicle in the loiter

area for 54 minutes, more than five times the air time of a single vehicle, as shown in Figure 2. The loiter time accomplished by the system is more than five times the average loiter time of a single vehicle.

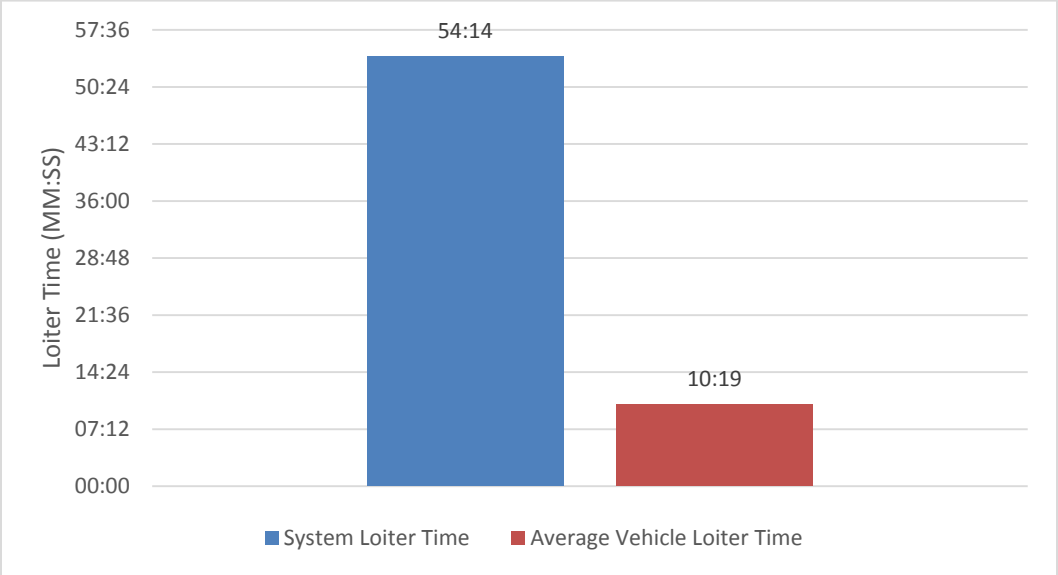


Figure 2. Loiter Time Comparison

The field testing was conducted with limited numbers of batteries. The life of the system is limited because the rate batteries are expended exceeds the time required to charge batteries. From analysis, with nine batteries and chargers per vehicle, the system could conceivably continue to operate for much longer durations.

Vehicle swapping using intelligent battery management is a feasible solution for providing persistent systems for much longer time durations. The system is easily scalable for added robustness or increased coverage. With further development, this system can become a fully deployed technology available to enhance our military capabilities.

Reference

Defense Science Board. 2012. *The Role of Autonomy in DOD Systems.* Washington, DC: Office of the Under Secretary of Defense for Acquisition, Technology and Logistics. https://www.acq.osd.mil/dsb/reports/2010s/AutonomyReport.pdf.

ACKNOWLEDGMENTS

This thesis has been quite the process and I could not have done it without the loving support of my wonderful wife, Paula, and daughter, Emilia. Thank you for giving me the free time to travel and conduct testing in Paso Robles and for the longer nights and weekends in putting it all down on paper. I love you always!

To my thesis advisor, Dr. Yakimenko, thank you for taking on my project and never questioning where it was going. You are truly an asset to the university, and I greatly appreciate our discussions and how simple you made thesis travel and material acquisition. You made this system tangible.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. THE ROLE OF UNMANNED SYSTEMS

Commercial and government entities have allocated large amounts of research and development resources to unmanned systems. According to a study conducted by Bard College, the Department of Defense (DOD) allocated approximately $4.457 billion to unmanned systems in the proposed fiscal year 2017 budget (Gettinger 2016). Commercial development and use of unmanned systems for filming movies, delivering packages, conducting engineering evaluations on difficult-to-reach equipment, and for use in hobbyist racing are just some of the applications driving companies to spend billions in advancing the technologies.

Commercial unmanned systems, equipped with multiple sensors and cameras, are widely available from companies such as DJI. Commercial entities and hobbyists primarily use these systems to conduct aerial photography and engineering evaluations. Many commercial systems have been used for DOD applications. Modifying a commercial unmanned system is typically cheaper than developing new government-based systems. However, many of these systems have cyber vulnerabilities that exist without full knowledge of the user, as seen in the U.S. Army's restriction on use of DJI systems imposed in August 2017 (Scott 2017).

The Defense Advanced Research Projects Agency (DARPA) has been a leader in guiding the discovery and advancement of many of every day technologies. Dr. Drozeski, program manager in the Tactical Technology Office of DARPA, recognized the need for better capabilities within the military, stating,

> Effective 21st-century warfare requires the ability to conduct airborne intelligence, surveillance, and reconnaissance (ISR) and strike mobile targets anywhere, around the clock. Current technologies, however, have their limitations. Helicopters are relatively limited in their distance and flight time. Fixed-wing manned and unmanned aircraft can fly farther and longer but require either aircraft carriers or large, fixed land bases with runways often longer than a mile. Moreover, establishing these bases or deploying carriers requires substantial financial, diplomatic, and security commitments that are incompatible with rapid response. (2017, 1)

The military has implemented unmanned systems for ISR and strike missions primarily in Iraq and Afghanistan. According to Dr. Kaminski of the Defense Science Board (2012), fielded unmanned systems are improving defense operations, but autonomy technology remains underutilized. Further, the value of these systems is not in replacing humans, but rather assisting humans in providing persistent capabilities.

Unmanned systems are widely used by military and civilians to provide quick services without risking human life. Many of the systems currently in use have limited flight times due to battery limitations. From the 2012 Defense Science Board's report on the role of autonomy in Department of Defense (DOD) systems,

> The true value of these [unmanned] systems is not to provide a direct human replacement, but rather to extend and complement human capability in a number of ways. These systems extend human reach by providing potentially unlimited persistent capabilities without degradation due to fatigue or lack of attention. Unmanned systems offer the warfighter more options and flexibility to access hazardous environments, work at small scales, or react at speeds and scales beyond human capability. (DOD 2012, 1)

Many unmanned systems have been developed since 2012, yet persistent systems are still an idea for the future.

A system is necessary to maintain the mobility of aerial platforms while providing a solution for limited battery life. Tethered systems offer extended-duration flight, but are limited in range. Little has been done to solve the problem of persistent platforms without placing great restrictions on the platform in use. Commercially, unmanned systems labelled as "persistent" require the vehicle to return to a base station for battery charging or battery swap. During this period of maintenance, the aerial platform is lost. In military applications, this presents a period of gapped collection or vulnerability to the operating unit. Manufacturers suggest that multiple independent unmanned systems may work together, but offer no guidance to tie the systems together seamlessly.

Many military units utilize unmanned systems for intelligence, surveillance, and reconnaissance (ISR) collection. Ground units often use commercial off-the-shelf (COTS) quadcopter and hex-rotor unmanned aerial systems (UAS) for situational

awareness and immediate visual feedback over an affected area. If these systems are to accomplish the vision provided by the Defense Science Board and aid operators, rather than further task them, autonomous persistent platforms are required.

B. THE DESIRE FOR PERSISTENT PLATFORMS

Greater autonomy drives advancement in unmanned systems. Operators no longer have to control every movement of the system and can focus on high level tasking and overall mission accomplishment. Often degraded by human fatigue or lack of attention, current systems do not provide truly persistent capabilities to users (Defense Science Board 2012). DARPA uses the term "persistent" to describe some systems, but fuel still limits the system's operational capabilities. Similarly, many commercial companies advertise persistent systems, but require the vehicle to return for extended periods of time to recharge. For military applications, persistent systems must overcome fuel constraints.

1. Current State of the Art and Technology

The majority of multi-rotor systems in use today are COTS. They provide users with a platform, usually a setup for taking pictures and recording videos from perspectives that had required much more costly helicopters or camera rigs. Multi-rotor systems are limited by their battery life. Many systems, such as the 3DR Solo shown in Figure 1, can be operated for up to 25 minutes.

Figure 1. Single 3DR Solo Multi-rotor System. Source: Holland (2015).

Tethered systems made up of a single multi-rotor system are typically COTS vehicles with an affixed tether which can provide power. The advantage of this system is that it extends the operating duration of the platform. While batteries are no longer a concern, the life of the vehicle's mechanical components limit the life of the system. Also, while tethered, the system is much less mobile and requires a power source on the ground which may further restrict system mobility.

2. Desired State

The intent of the system described in this thesis is to provide an aerial platform to affix ISR, communication systems, or whatever the user sees fit. The system operates for an extended duration using multiple vehicles. Vehicles swap in the loiter position autonomously based on battery monitoring and management through the ground control station (GCS). The GCS exists primarily to monitor the system health and real-time activity.

The near-term solution does not focus on the use of the platform. Its primary purpose is to provide an aerial platform. The solution also does not focus on extending individual battery life or automatically swapping batteries, but rather looks at what is possible with current technology. This is the primary purpose of this thesis.

The system is intended to be used as an early indication, extra sensor, or an additional defensive weapon for a military unit. The system will be launched by a user to provide a platform for seamless sensors, or cameras, through autonomous vehicle swapping and streaming data transfer. As a secondary mission, the system can be tasked to investigate other areas and will automatically launch and task the next multi-rotor system to conduct the primary loiter mission near the unit. The best weapon to counter an unmanned system is an unmanned system. The future system utilizes computer vision to counter adversary unmanned system nearby at the desire of the unit commander.

3. Research Application

This thesis focuses on determining the feasibility and battery limitations in providing a persistent multi-rotor aerial platform using battery health management and vehicle swapping. The lack of persistent capabilities in available systems presents an opportunity for development of future autonomous technologies. Using commercially available quadcopters to minimize cost and provide a proof of concept, this thesis evaluates the capabilities and limitations of multiple vehicle persistent systems. Advancing persistent technologies would allow the military longer on station times while minimizing any time gaps in coverage. The U.S. military relies heavily on its sensors to detect and counter threats. Affixing sensors to persistent systems would allow operators to focus on the big picture without missing important information. When a ship is in port or a military unit is stationary, the majority of sensors power down and safety is dependent on security personnel. Cuts to military staffing and the increased speed of weaponry, greatly reduces reaction time. The number of personnel, capacity of human memory in identifying threats, and personnel fatigue limit security patrols; whereas, an unmanned vehicle can autonomously patrol at higher speed and frequency which could complement or replace security personnel (Seng et al. 2015). Persistent unmanned

systems could advance our military into the twenty-second century, but still require extensive development.

C. TECHNOLOGY GAPS IN PERSISTENT PLATFORMS

The Tern system is one of the Defense Advanced Research Projects Agency (DARPA) solutions in development that recognizes the limitations of helicopters and manned aircraft for use with the U.S. Navy. The objective of Tern is to provide a long endurance unmanned aerial system for ISR and strike missions deployed, operated, and recovered from small ships (DARPA 2017). Though it will provide longer on-station times than traditional aircraft, the need for fuel limits the system. From conceptual images provided by DARPA and Northrop Grumman, shown in Figure 2, the vehicle appears similar in size to a shipboard manned helicopter. Based on the vehicle size and design, the system likely cannot swap vehicles or conduct refueling autonomously in order to keep the system on station. Moving vehicles of this size to launch positions would require manpower and personnel time for execution.

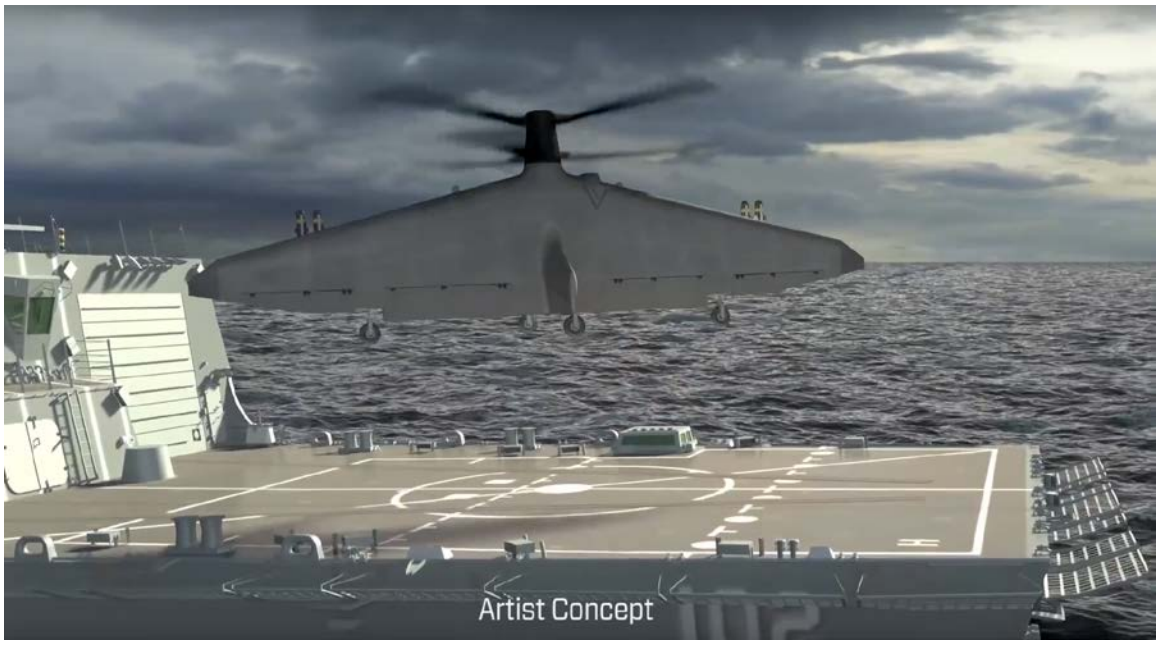


Figure 2. Tern Artist Concept. Source: Northrop Grumman (2016).

Commercial companies such as Hoverfly and H3Dynamics offer versions of persistent coverage capabilities. Hoverfly offers a multi-rotor platform that is tethered to a ground station. The ground station provides power to the vehicle through the affixed cable, which limits the vehicle's mobility. Hoverfly vehicles cannot travel far from the ground station. In a military environment, the loss of the single vehicle would mean the loss of the system's full capabilities. By contrast, H3Dynamics produces a product called Dronebox, which does not require a tether. The Dronebox system features autonomous flight and has a command center capable of charging multi-rotor vehicle batteries. However, the life of the vehicle's battery remains a limitation. Even with multiple Dronebox systems, the company does not offer a way to mesh them together to work as a single autonomous system.

D. THESIS OBJECTIVES AND RESEARCH APPROACH

Unmanned systems provide increased capability to modern-day militaries. Autonomous unmanned systems act as force multipliers, providing unit commanders immediate capabilities without requiring increased training or manpower. Development of an extended-duration aerial platform is crucial in improving the military's ability to collect intelligence, provide early warning to deployed units, and protect forces on the ground. Figure 3 shows video surveillance coverage for a bottom mounted camera on the platform.

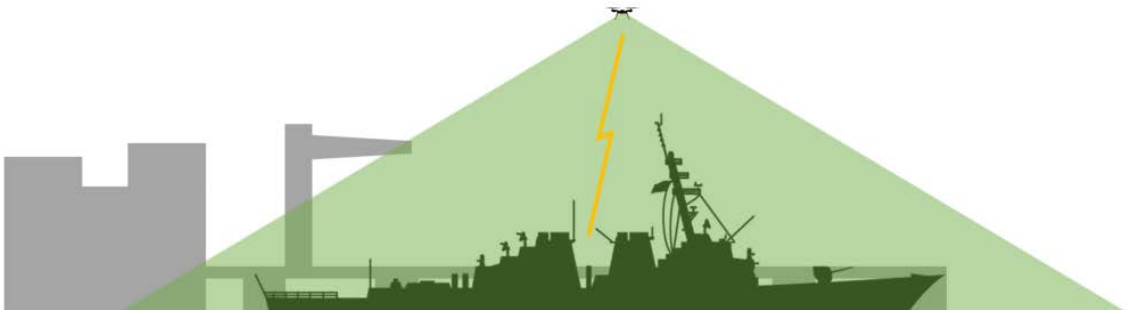


Figure 3. High-Level Operational Concept Graphic

The objective of this thesis is to determine whether a system of quadcopters can (1) provide a persistent aerial platform, and (2) operate autonomously utilizing battery management techniques, while they (3) explore the limitations of an extended-duration airborne system. The system is comprised of COTS quadcopter systems, a single router, a laptop, and a computer program. Field experiments helped to develop and evaluate the system as well as determine its capabilities and limitations. To determine the suitability of this system for military use, a survivability analysis was also conducted.

While UAS can consist of an array of vehicles, this thesis focuses on multi-rotor vehicles. A multi-rotor vehicle is an aerial vehicle with at least two rotors. The multi-rotor vehicle used for development and testing is the commercially available 3DR Solo quadcopter. The user conducts battery changes manually, but in the future system, autonomous swapping occurs.

The system developed is not ready for the field, but rather a prototype of a system to show proof of concept. The system utilizes COTS quadcopters to examine whether a low-cost solution is possible. Providing a more secure, mission specific system will require additional research and development.

E. THESIS ORGANIZATION

To address the objectives formulated in the previous section, this thesis is organized as follows.

Chapter II describes the development and configuration of the test platform. This chapter provides the methodology for the experiment. The aerial vehicle is introduced along with the necessary vehicle configuration, network configuration, and software configuration to realize the near term desired system. The scenario details the field experiment and evaluation criteria for the system.

Chapter III highlights the results of field testing and analysis. In this chapter, field testing appears along with the results. Field testing covers how the system performed as a whole and the results of battery evaluations conducted. Battery evaluation became a major concern for the system moving toward a fielded system.

Chapter IV focuses on the survivability of the system for use in military applications. To determine how this system might operate in the field in its future desired state, this chapter contains the survivability assessment. The assessment provides a susceptibility and vulnerability analysis as well as an evaluation of improvement technologies.

Chapter V concludes the thesis and suggests future work recommendations.

THIS PAGE INTENTIONALLY LEFT BLANK

II. DEVELOPMENT OF THE TEST PLATFORM

This chapter provides details on the system used and describes the process necessary to prepare for field testing of the proof-of-concept multi-rotor persistent platform. A sequence of events is shown in Figure 4. The system concept was established by December 2016. From December to February, the investigator explored available multi-rotor vehicles to determine which were best suited to work in a persistent system. Software research began in January and continued until June when the vehicles were delivered to the Naval Postgraduate School. From June through July, the aerial vehicles and network were configured and software was developed and tested in increments. Field testing ended in August 2017.

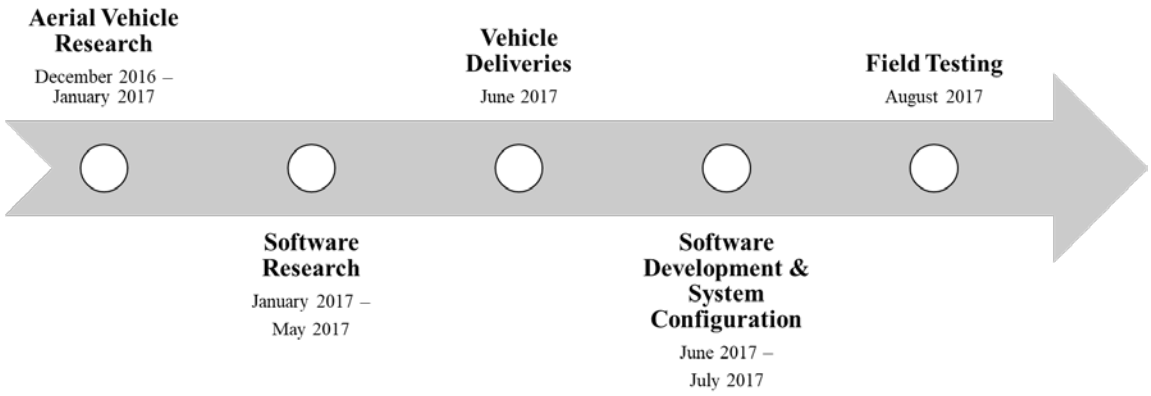


Figure 4. Research and Development Timeline

A. SYSTEM OVERVIEW

The system conducts vehicle swapping based on the battery state of the airborne vehicle. When the user starts the system, the first vehicle launches to the desired position and loiters. The system monitors the battery health of the airborne vehicle, and when the battery level falls below the user set threshold, the next vehicle launches. The first vehicle rises to avoid collision while continuing to provide the platform to the user. The replacement vehicle flies to the desired position and loiters in the area previously occupied by the first vehicle. Once the replacement vehicle reaches the desired position,

the first vehicle returns to the launch platform for battery swap or charging to be ready for its next use. This system cycles through vehicles until all batteries are exhausted or the user ends the mission. Additional safeguards are built in to skip vehicles that encounter faults or errors, such as vehicles with depleted batteries or ones that experience issues with reaching desired altitude on launch.

B. HARDWARE SETUP

There are two basic parts of the system: the hardware and the software. The hardware consisted of the aerial vehicles and wireless network, each of which, when properly configured, communicated with the laptop. The laptop was not discussed in this thesis as it is was primarily used to develop and execute the program code, which could be accomplished by most low-level computers.

1. Aerial Vehicle

Many commercially available vehicles have software development kits that allow the user to program and control through software running external from the controller and vehicle. The vehicle chosen, the 3DR Solo, shown in Figure 5, offered a low-cost requirement with readily available parts.

Figure 5. 3DR Solo Controller and Vehicle

The 3DR Solo specifications, shown in Table 1, were competitive with other commercially available quadrotors. While conducting research on the development kits available, 3DR Solo also proved to have a wealth of software development documentation. Additionally, former 3DR employees had developed a collaborative document that detailed the steps that they had used to operate multiple 3DR Solos as a "swarm." For the purposes of field testing, the equipment included three 3DR Solos, one wireless router operating at 2.4 GHz, and a laptop capable of running the Python programming language.

Currently, the system is capable of operating with additional 3DR Solos, so long as the vehicles are configured properly and their associated internet protocols and ports are written into the software code used to control the system.

Table 1. 3DR Solo Vehicle Specifications. Adapted from 3DR (2015).

Component	Description
Dimensions	10 inches tall, 18 inches motor-to-motor
Weight	3.3 lbs. / 3.9 lbs. with GoPro and Solo Gimbal
Range	0.5 miles
Max speed	55 mph
Flight time	20–25 minutes
Flight battery	Lithium polymer 5200 mAh 14.8 VDC
Battery charge time	1.5 hours
Communications	Secure WiFi network
Frequency	2.4 GHz
Flight battery	Lithium polymer 5200 mAh 14.8 VDC
Flight time	20–25 minutes
Motors	880 kV
Autopilot	Pixhawk 2

2. Vehicle Configuration

From the manufacturer, each vehicle came set up to connect with its included controller. Using the network protocol Secure Socket Shell (SSH), each vehicle and controller was remotely reconfigured. To operate the system properly, port identifiers for each vehicle were assigned along with the network Service Set Identifier (SSID) information necessary to connect to a single wireless network.

3. Network Configuration

The goal is to operate three vehicles from a single terminal running Python code. The system operates autonomously, monitoring vehicle battery health and swapping vehicles, when necessary. To do this, the vehicles must all operate on a common network.

The three vehicles and one laptop are connected through a single wireless access point operating at 2.4GHz. The wireless access point allowed the ability to access both the internet and any of the vehicles on the network. The internet could update the vehicles to the latest manufacturer's firmware but otherwise was not necessary for the purpose of developing or testing the system.

The network architecture, shown in Figure 6, is a block diagram showing the system entities and connections. The 3DR Solos connect to the laptop through the wireless network router. Internet connection is optional through the router. The network shows solid lines to indicate wireless connections, and a dotted line to show an optional connection to the internet. The three colored blocks represent individual 3DR Solo controllers and associated vehicles. The network diagram illustrates the critical path and dependencies for connectivity.

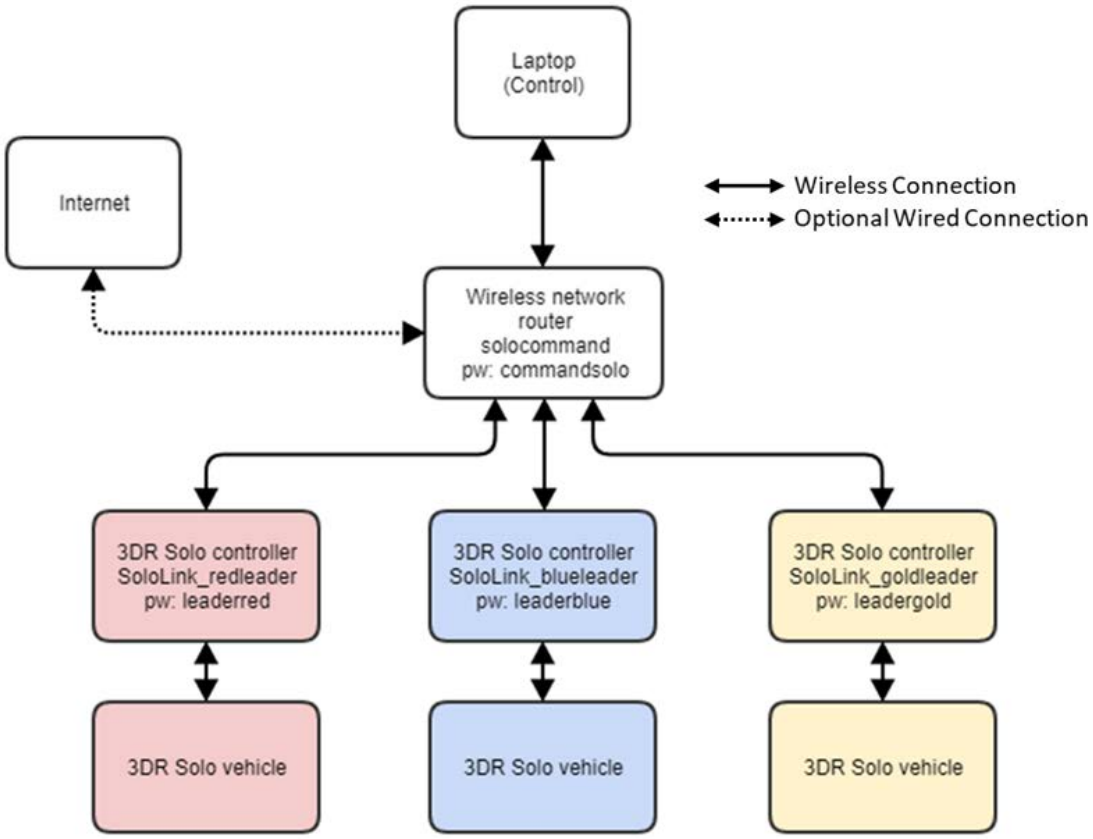


Figure 6. System Network Architecture

C. SOFTWARE ARCHITECTURE

Python, a programming language designed for ease of use and readability, was the primary means of developing autonomous system function. Using the documentation—provided as part of DroneKit-Python—an online software development kit, and multiple smaller field tests the vehicles were configured to conduct flight operations initiated by the Python script. Short flight tests were conducted in May and June 2017 at Camp Roberts to ensure the function of sections of the code before the field test.

The code for the system followed a six-step development cycle:

1. Collecting Battery Data

The first challenge was to connect to a single 3DR Solo and ensure that real time battery information could be accessed by the system. This is the crux of the entire project and without this capability, the system would not properly operate. The code reads the battery status of a single vehicle while connected and activates the next quadrotor when the battery's health falls below the desired threshold.

2. Multiple Vehicle Connectivity

The second challenge was to verify that all of the vehicles connected to the network and provided real time system health information to the Python script. The code ensured that the vehicles could report all information back to the ground station—the laptop—without losing information from another vehicle.

3. Vehicle Launch

The next challenge was launching vehicles on command. The 3DR Solo code provided in DroneKit-Python experienced issues that caused the vehicle not to launch properly. Often, the vehicle would hover less than a meter above the ground, but the code would ignore the state of the vehicle. A number of implemented checks warranted that the vehicle launched successfully before proceeding. Additional measures guaranteed that if a vehicle remained in the launch state, it could not continue until the vehicle reached the desired launch altitude or a replacement vehicle launched to that altitude in its place. This prevented the code from progressing before the vehicle was ready to respond.

4. Vehicle Flight

Flight control was relatively simple compared to the vehicle launch or connectivity challenges. For the test, the intended flight path was to transit to a specified latitude and longitude to loiter using the flight controller and Global Positioning System (GPS) onboard.

5. Vehicle Swapping

The final development for the vehicle was to code in automated vehicle swapping, which raised the original vehicle to a higher altitude before sending the replacement vehicle. The replacement vehicle moved to occupy the position once held by the original vehicle. Once the replacement vehicle moved into the loiter position, the original vehicle returned to the launch platform.

6. Data Logging

To analyze the success of the system in the field test, the code included data collection scripts. The code collected vehicle data and appended an external file to the laptop. Excessive data collection led to multiple system crashes and subsequent loss of collected data. As a remedy, fewer data points were collected but the data focused on vehicle transition points within the test. Unfortunately, numerous flight tests encountered this issue and data was lost for those flights.

THIS PAGE INTENTIONALLY LEFT BLANK

III. TEST AND EVALUATION OF THE PROPOSED SOLUTION

This chapter describes the procedures to evaluate the proposed solution. It starts with a brief description of the test and evaluation environment, continued by a test scenario, scope, and objectives description. The results of field testing are presented and an analysis of system limitations and improvement opportunities are evaluated. The chapter ends with a discussion of system scalability.

A. TEST ENVIRONMENT

1. Camp Roberts

The Naval Postgraduate School is located in close proximity to the Monterey regional airport. As such, flight operations with government furnished equipment is extremely limited. McMillan Field, at Camp Roberts, California, offers dedicated airspace and provides the power and connectivity resources necessary to conduct field testing. All flight tests and field testing were conducted at McMillan Field.

2. Scenario

To evaluate the feasibility of the system, the loiter scenario was developed. During the scenario, the system flies approximately 100 meters away from the launch point to an altitude of 100 meters and provides an aerial platform that loiters in this position until cancelled by the user or available batteries are exhausted. This simulates a real world environment where this system operates above a ship in port or ground base providing a nearby airborne platform for sensors. Each of the 3DR Solos include its gimbals and cameras, the GoPro 4, to test with a payload.

Success for this system is defined as multiple vehicle swaps when the user defined 30% battery threshold level is met. The system shall provide aerial coverage for a duration that exceeds the capability of a single 3DR Solo.

3. Scope

The development and testing focused on the vehicle as a platform. The main focus of field experimentation determined whether the system would autonomously swap vehicles when the battery level reached a defined limit. Camera and video production development and testing were not included but should be developed for future systems.

4. Objectives

The primary critical operation issue (COI) addressed in this system is mobility. In order to achieve mobility capabilities, the measure of effectiveness (MOE) tested in this thesis is endurance. Each measure of performance (MOP) evaluated is shown in Figure 7.

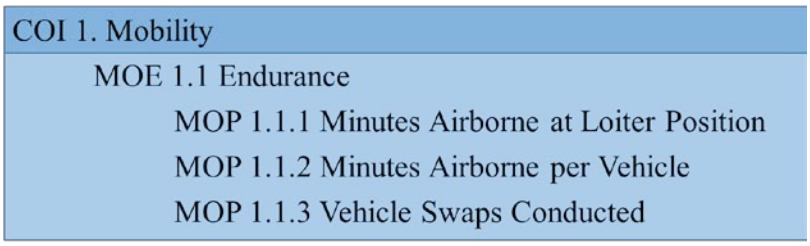


Figure 7. System Mobility MOE and MOP

Field testing at Camp Roberts on August 7–8, 2017, sought to determine whether the system would swap vehicles without user intervention, triggered solely on battery percentage remaining. When the battery percentage of the flying vehicle falls below the 30%, the test value set, the Python code initiates the swapping process.

B. TEST RESULTS

Figure 8 shows a summary of field testing results. Table C-1 (see Appendix C) provides complete details. Three vehicles were tested. Vehicles one and two executed multiple flights before expending all available batteries.

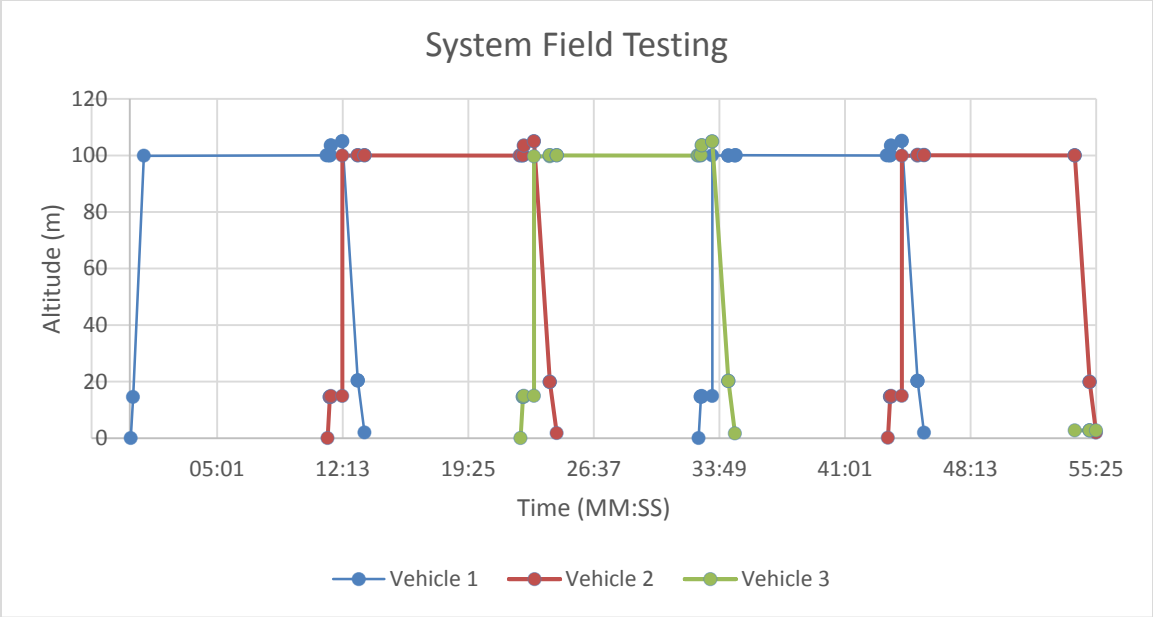


Figure 8. System Field Testing Results

The results show that the system successfully operated multiple vehicles in the intended loitering position and swapped vehicles autonomously. The testing concluded after five fully-charged 3DR Solo batteries fell below the 30% battery threshold. A timeline of field testing is summarized in Table 2.

Table 2. System Action Timeline

Time (MM:SS)	Event	Vehicle On Station	
00:03	Vehicle 1 Launch		
00:48	Vehicle 1 Reached Loiter Point	Vehicle 1 On Station	
11:17	Vehicle 1 Battery Limit Exceeded (29%)		
11:20	Vehicle 2 Launch		
11:28	Vehicle 1 Raise Altitude For Vehicle Swap		
12:11	Vehicle 2 Reached Loiter Point	Vehicle 2 On Station	SWAP
12:11	Vehicle 1 Return to Platform		
13:28	Vehicle 1 Land		
22:22	Vehicle 2 Battery Limit Exceeded (29%)		
22:25	Vehicle 3 Launch		
22:33	Vehicle 2 Raise Altitude For Vehicle Swap		
23:11	Vehicle 3 Reached Loiter Point	Vehicle 3 On Station	SWAP
23:11	Vehicle 2 Return to Platform		
24:29	Vehicle 2 Land		
32:34	Vehicle 3 Battery Limit Exceeded (29%)		
32:37	Vehicle 1 Launch		
32:45	Vehicle 3 Raise Altitude For Vehicle Swap		
33:24	Vehicle 1 Reached Loiter Point	Vehicle 1 On Station	SWAP
33:24	Vehicle 3 Return to Platform		
34:42	Vehicle 3 Land		
43:24	Vehicle 1 Battery Limit Exceeded (29%)		
43:28	Vehicle 2 Launch		
43:36	Vehicle 1 Raise Altitude For Vehicle Swap		
44:16	Vehicle 2 Reached Loiter Point	Vehicle 2 On Station	SWAP
44:16	Vehicle 1 Return to Platform		
45:33	Vehicle 1 Land		
54:11	Vehicle 2 Battery Limit Exceeded (29%)		
54:11	No Replacement Available		
55:03	Vehicle 2 Return to Platform		
55:25	Vehicle 2 Land		

Four vehicle swaps occurred during field testing. Vehicles remained at the loiter position for approximately 54 minutes using vehicle swapping and manual battery changes. When a vehicle returned to the launch platform with an expended battery, the vehicle was turned off, equipped with a fully charged battery in place of the old one, and the vehicle was restarted. The vehicle reconnected with the network and the Python script would reconnect to the vehicle as the script progressed.

From the results, the system is much more capable than using a single 3DR Solo. The loiter time is the period that the vehicle occupies the desired position intended for the platform. When compared to the average individual vehicle loiter time of 10 minutes and 19 seconds, the system provided coverage in the loiter area for more than five times the duration at 54 minutes and 14 seconds, as shown in Figure 9.

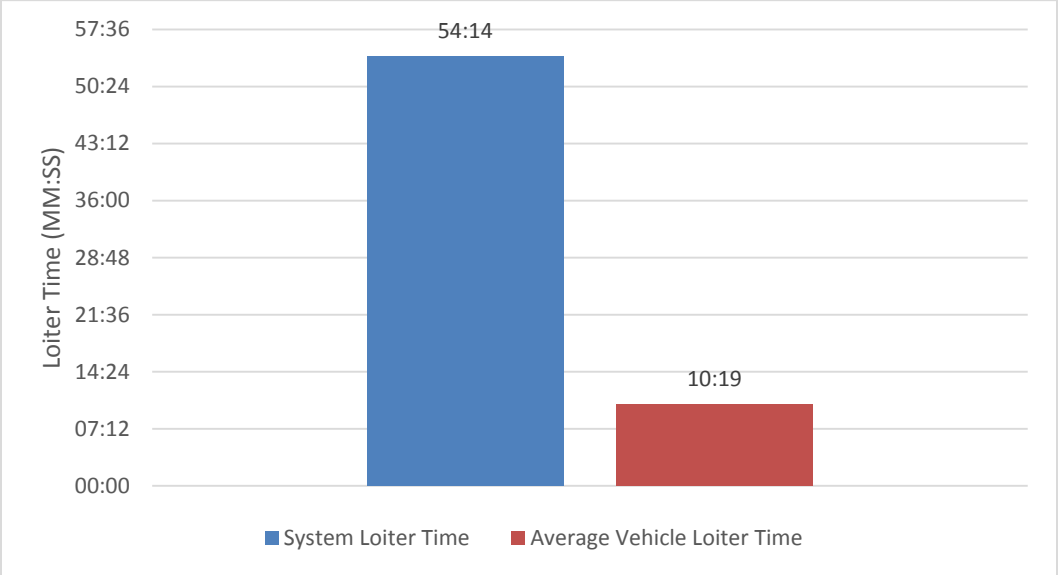


Figure 9. Loiter Time Comparison

One alarming finding was that the battery life of the 3DR Solos expended battery approximately twice as fast as the manufacturer's specifications. Instead of 20–25 minutes, the vehicles could only safely operate for 12–14 minutes, as shown in Table 3.

Table 3. Field Testing Individual Vehicle Flight Time

	Vehicle Flight Time (MM:SS)
Sortie 1	13:24
Sortie 2	13:09
Sortie 3	12:18
Sortie 4	12:55
Sortie 5	11:56
Manufacturer Specification	20:00-25:00

With additional batteries, this system could conceivably continue to operate for much longer durations. The primary limitation is the number of batteries available at full charge. Unfortunately a battery discharges at 12–14 minutes and charging takes approximately 1.5 hours. This does not keep up with the rate at which the battery drains when in use by a vehicle.

$$Number\ of\ \ Batteries\ Required = \left\lceil 1 + \frac{Time\ to\ Charge\ a\ Battery}{Time\ to\ Expend\ a\ Battery} \right\rceil \tag{1}$$

$$Number\ of\ \ Batteries\ Required = \left\lceil 1 + \frac{90\ \ minutes}{12\ \ minutes} \right\rceil = \lceil 8.5 \rceil = 9\ \ Batteries$$

For the system to keep up with the rate at which the batteries are expended, nine batteries and chargers are necessary. This assumes a maximum charge time of 90 minutes for a battery, as provided by the manufacturer, and assumes that the time to expend a battery does not decrease. Additional battery studies are necessary to determine the feasibility of extending the operational duration of the system.

C. SCALABILITY

The system developed and tested in this thesis is a proof of concept. Employing even more than three vehicles would assure true survivability and extend the capability of the system. Additional aerial vehicles should follow the same configuration guidance included in Appendix A and include the network address of the new vehicles in the program code. The minimum number of batteries and chargers required remains nine

whether the system has three vehicles or more, as seen in Figure 10. In Figure 10, the top diagram represents battery usage by the three vehicle configuration system whereas the bottom diagram shows the four vehicle configuration system, both with nine batteries. Both diagrams show the time required to discharge battery 1, shown in orange, and charge, shown in green.

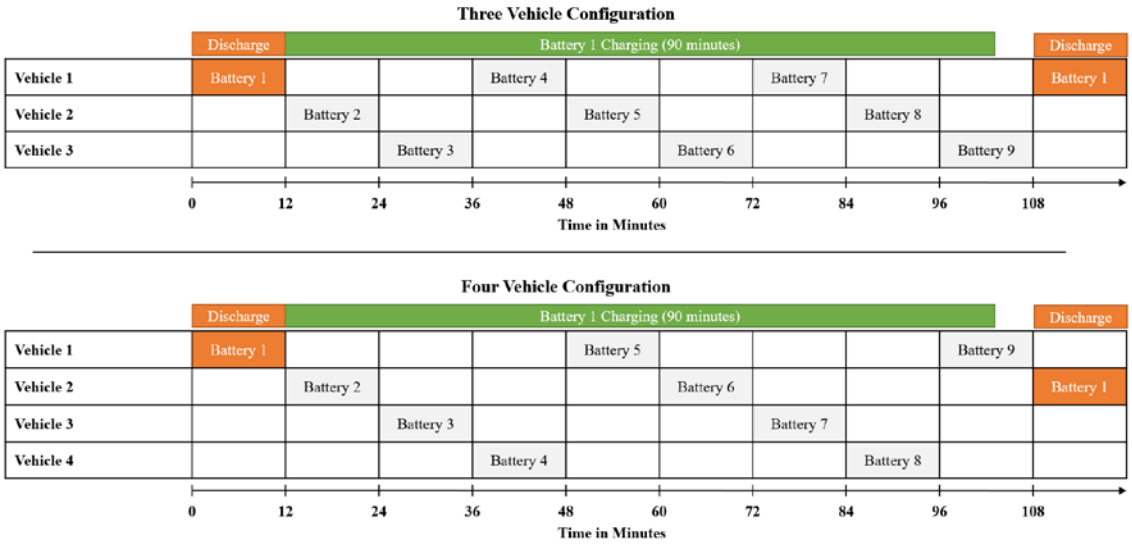


Figure 10. Nine-Battery Discharge Plan

To cover larger areas, employ multiple interconnected systems with the same fundamental setup and configuration. This system scales well with mission requirements.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. SUITABILITY OF THE DEVELOPED SYSTEM

In the system's desired future state, the system is supposed to be highly survivable. To address the issue, this chapter presents an analysis of the prospective system's threats, key limitations, and opportunities with respect to susceptibility and vulnerability.

A. MISSION THREAT ANALYSIS

1. The Mission

Managed by a single computer running the Python script, the system maintains a single quadcopter over an area and conducts vehicle swaps autonomously to manage the limited battery life of individual quadcopters. The system concept diagram, Figure 11 illustrates this process. There are three vehicles in this system. The first vehicle, shown in red, is replaced by the second vehicle, shown in blue, while the third vehicle, shown in black, remains on the ground next in line for tasking. Utilizing autonomous battery management, the system provides extended coverage for less cost than larger aerial vehicles with much quicker response times.

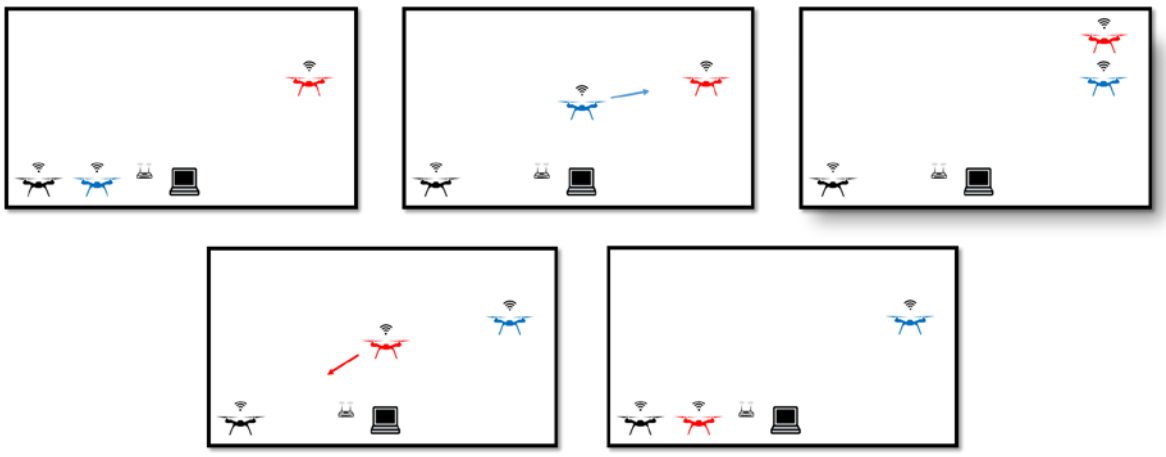


Figure 11. Operational Concept Diagram

The system conducts vehicle swapping based on the battery state of the airborne vehicle. The first vehicle launches to the desired position and loiters. The system monitors the battery health of the airborne vehicle and when the battery level falls below the user set threshold, the next vehicle launches. The first vehicle rises to avoid collision while continuing to provide the platform to the user. The replacement vehicle flies to the desired position and loiters in the area previously occupied by the first vehicle. Once the replacement vehicle reaches the desired position, the first vehicle returns to the launch platform for battery swap or charging to be ready for its next use. This system cycles through vehicles until all batteries are exhausted or the user ends the mission.

2. Theater of Operation

The system is intended for use in a stable, fortified environment. When used on a ship, the ground station and launch platform exist onboard. The system may be exposed to hostile actions and extreme environmental conditions, but for the most part will operate in a relatively low-risk setting. Onboard a ship, in order for adversaries to access the ground station or other quadcopters used in the system, they would need to attack the ship itself. This makes the system less exposed to direct threats.

B. MISSION-THREAT ENCOUNTER ANALYSIS AND GEOMETRIC DESCRIPTION

Mission-threat encounter analysis and geometric description is accomplished by examining expected threats, their characteristics and likelihood, and the system's critical components and kill modes.

1. Expected Threats

Given the theater of operation, Table 4 characterizes and describes the most likely threats. Given the operational environment and cost of the threats, the most likely threat is from small arms.

Table 4. Expected Threats

Threat	Range	Likelihood
Small arms	600 m	High
Portable and relatively inexpensive At long range, inaccurate		
Electronic attack	Power dependent	Medium
Most effective Requires technology, power, and proximity to attack		
Man-portable air-defense systems (MANPADS)	6 km	Low
Very capable against target emitting Infrared (IR) signatures		
Radar guided missile	70 km	Low
Expensive and proven		

In March 2017, BBC reported that a U.S. ally conducted a strike on a small unmanned aerial system (UAS) using a patriot missile successfully at the cost of $3M (Baraniuk 2017). The engagement demonstrated the system's ability to engage the small target, even though according to Justin Bronk, a researcher at the Royal United Services Institute, the Patriot system may struggle to target a small quadcopter effectively (2017). The radar guided missile system costs millions, making it economically infeasible for use against an individual quadcopter in the small unmanned aerial system (SUAS) that costs only a few hundred dollars. Similarly, a MANPAD costs thousands of dollars and the financial cost associated for its deployment would be overkill for an inexpensive quadcopter.

2. Critical Components and Kill Modes

Table 5 describes the SUAS critical components and their associated kill modes.

Table 5. SUAS Critical Components and Kill Modes

CRITICAL COMPONENT	KILL MODE(S)
Power System	
Battery	Battery connection severed/damaged Battery damaged Battery removed Battery depleted
Propulsion	
Propeller	Damage to control surfaces Loss of propeller
Motor	Motor failure
Flight Control System	
Navigation	Loss of satellite signal Connection failure
Flight Control	Disruption of control signal path Loss of control power Mechanical damage Overheating
Ground Station	Loss of connection
Payload	
Sensors/Cameras	Sensor/camera damaged Connection failure

Quadcopters make up the system, which has many critical components without redundant parts. By killing a single component, the quadcopter will likely fail, damaging the system. However, the small size of the system makes the vehicle difficult to engage with a weapon, which makes it less susceptible to attack.

C. SUSCEPTIBILITY ANALYSIS

The likelihood that the system is hit by a weapon or threat is referred to as susceptibility, DOD recognizes six concepts: tactics; threat warning; signature reduction; noise jamming and deceiving; expendables; and threat suppression (Ball 2003). This SUAS actively employs the first three. By understanding the aspects which make the system susceptible, DOD can conduct system risk mitigation or reduction.

1. Tactics, Flight Performance

As an automated system, to ensure success, this SUAS does not require training and proficiency as do most manned systems. The tactics for decreasing susceptibility primarily focus on minimizing isolated exposure of the SUAS and with redundant systems.

The SUAS is made up of multiple quadcopters, which offers a tactical advantage to ensure immediate replacement. If one of the quadcopters is lost, a replacement will be launched immediately to provide continuous coverage. The aerial platform exists toward the center of an operating unit, which will likely minimize direct enemy exposure. The SUAS may act as a decoy while protecting ground or naval units that can return fire. These tactics reduce system susceptibility.

2. Threat Warning

Using a radar warning or missile launch and approach warning system, military commanders can receive advanced notice of imminent threats. These systems provide information on the location and type of threat.

Utilizing cameras and additional sensors, the SUAS can detect incoming threats using computer vision, the process for acquiring and processing digital images to gather information. It may then take evasive action removing an easy kill opportunity for the enemy and allowing the system to be less susceptible. It would also provide the ground or naval units an immediate notification of a nearby threat, as seen in Figure 12.

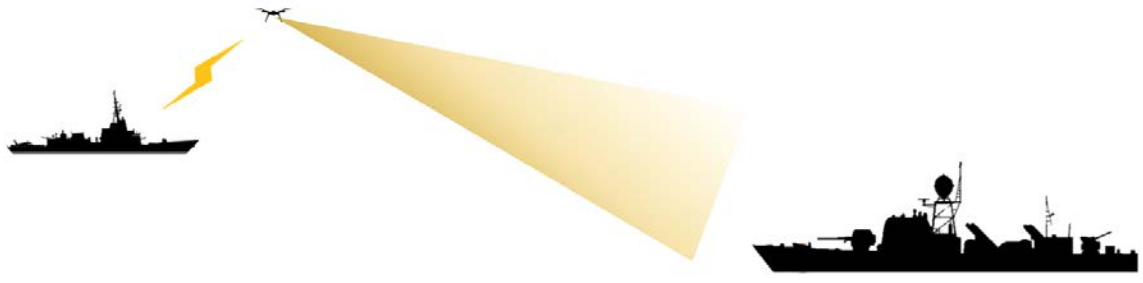


Figure 12. Threat Warning

3. Signature Reduction

Signature reduction is the practice of reducing the detectability of the vehicle below a weapon's sensor threshold. This means minimizing signature in radar, infrared (IR), visual, magnetic, acoustic, and ultraviolet (UV), as seen in Figure 13 from Adams' Susceptibility Reduction lecture (2017). Because radar typically provides the greatest detection ranges, minimizing radar cross-section (RCS) has been a common practice in military design.

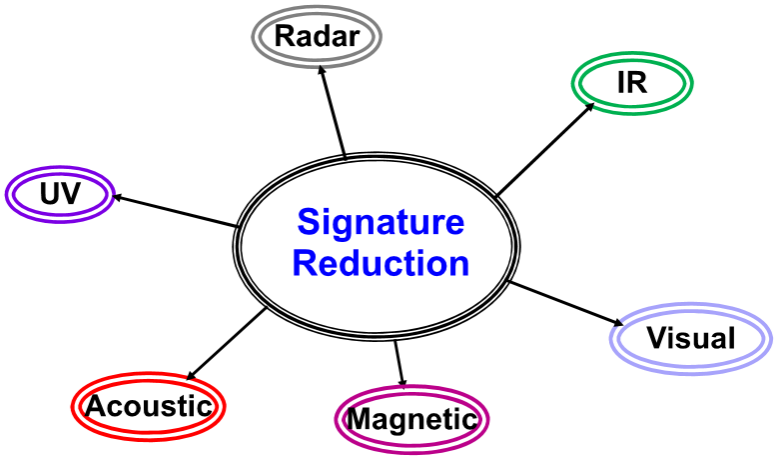
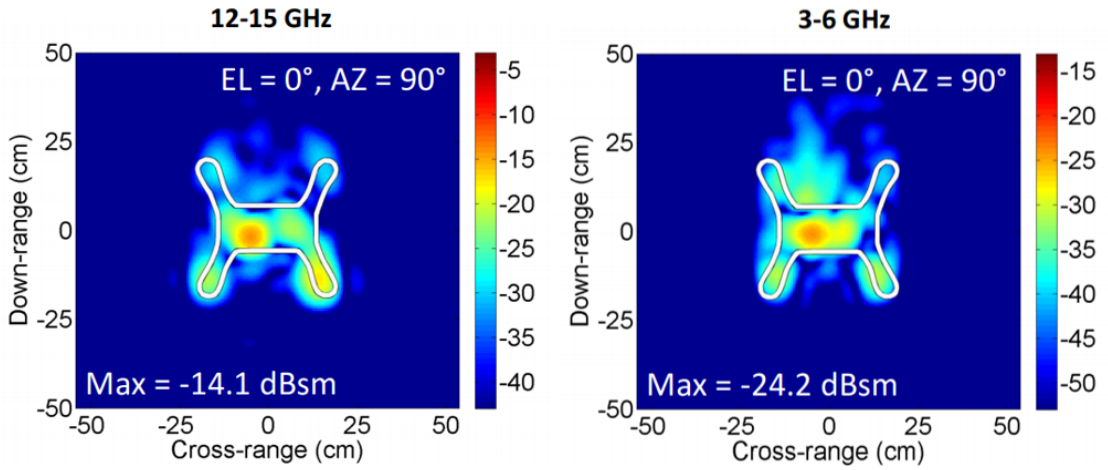


Figure 13. Signature Reduction. Source: Adams (2017).

Small aerial systems have insignificant RCS and may be lost in air clutter or misidentified as another object. The SUAS has a small RCS because of its size and lack of traditional gasoline or diesel fuel engine. It will also operate with larger units, making the SUAS RCS less visible to radar operators. The University of Texas at Austin completed an evaluation of the 3DR Solo with respect to radar cross-section and found it to have a very small RCS (Li and Ling 2016). The actual RCS value, shown in Figure 14, will vary depending on the radar and environmental conditions, but radar operators observe the value as relatively small, according to Li and Ling's report Radar Signatures of Small Consumer Drones (2016).



The white outlines indicate the overall size and location of the 3DR Solo.

Figure 14. Radar Cross-Section of 3DR Solo. Source: Li and Ling (2016).

D. VULNERABILITY ANALYSIS

The likelihood that the system will survive an engagement, vulnerability, is defined by the DOD in six different concepts: component shielding; component location; component elimination or replacement; component redundancy; passive damage suppression; and active damage suppression (Ball 2003). To keep cost low and maximize operational characteristics of the system, the SUAS actively employs redundancy.

The SUAS consists of at least three quadcopters with commercially available replacement parts, should repair be necessary. This provides the system with redundancy, though there may be a delay in the time between the loss of one unit and stationing of the second unit.

E. SURVIVABILITY ENHANCEMENT TRADE STUDY

As typical with all systems, adding enhancements with operational functions of the system entails major trade-offs. Besides the additional cost of developing the system further, susceptibility and vulnerability improvement opportunities determine the trade-offs of enhancing the SUAS.

1. Susceptibility Improvement Opportunities

Additional signature reduction. The SUAS could be outfitted with stealth technologies. By changing the shape, curves, edges, and outer material, the system may achieve greater stealth advantage. The trade-off comes at the cost of additional weight on the system.

Noise jamming and deceiving. The SUAS may employ this feature in the future. Electronic attack and protection would help to decrease the susceptibility of the system. The trade-off is weight and power consumption that come along with any electronic system addition.

Expendables. The SUAS may employ this feature in the future. Expendables may help the system to avoid impact from MANPADS or missiles. However, this comes at the cost of weight on the system and additional power consumption. The MANPAD/missile threat is assessed as minimal to the system as a whole, given the costly nature of employing these weapons.

Threat suppression. The SUAS was not designed as an offensive system, but should the desire exist, the system could be outfitted as an enemy-seeking weapon. In that, it may carry a warhead. The vehicle could be tasked to seek out enemy fortifications or high value units for destruction, eliminating or suppressing the threat. The system could accommodate this by launching an additional system to take over the persistent platform mission. This enhancement comes at the cost of weight and additional power consumption, as seen with the other susceptibility reduction features. Another trade-off would be a loss of an individual system and redundancy. In using the system for offensive capabilities, the system is removed from the persistent platform mission. Additional quadcopters would be necessary to maintain both the offensive and defensive mission outlined in threat suppression.

2. Vulnerability Improvement Opportunities

Component shielding. The SUAS may be equipped with resistant materials such as composite armor which should lower the vulnerability of the system to small arms fire. With the advancement of composite lightweight armor, this is a likely feasible

enhancement for the SUAS. Any trade-off would come in increased unit cost and possible weight addition.

Component redundancy. The SUAS has redundancy through multiple quadcopters. Individual quadcopters could have component redundancy, allowing the quadcopter to take damage and lose components, while carrying on the mission. This comes at the cost of additional weight.

Passive/active damage suppression. The SUAS could be equipped with damage suppression systems. The main concern is additional power draw and weight.

3. Survivability Enhancement Impact

The majority of the enhancements' trade-off cost includes increased weight or power draw, leading to decreased time of flight. With the low cost of individual quadcopters and the redundant nature of the system, survivability enhancements are not financially logical.

F. KILL TREE

1. Individual Kill Tree

The kill tree of an individual quadcopter and of the system of quadcopters are shown in Figure 15. The left portion of this figure shows a single quadcopter's kill tree which is included in the system of three quadcopter's overall system kill tree. A single quadcopter looks vulnerable. However, the system kill tree of the SUAS is a more robust, redundant system.

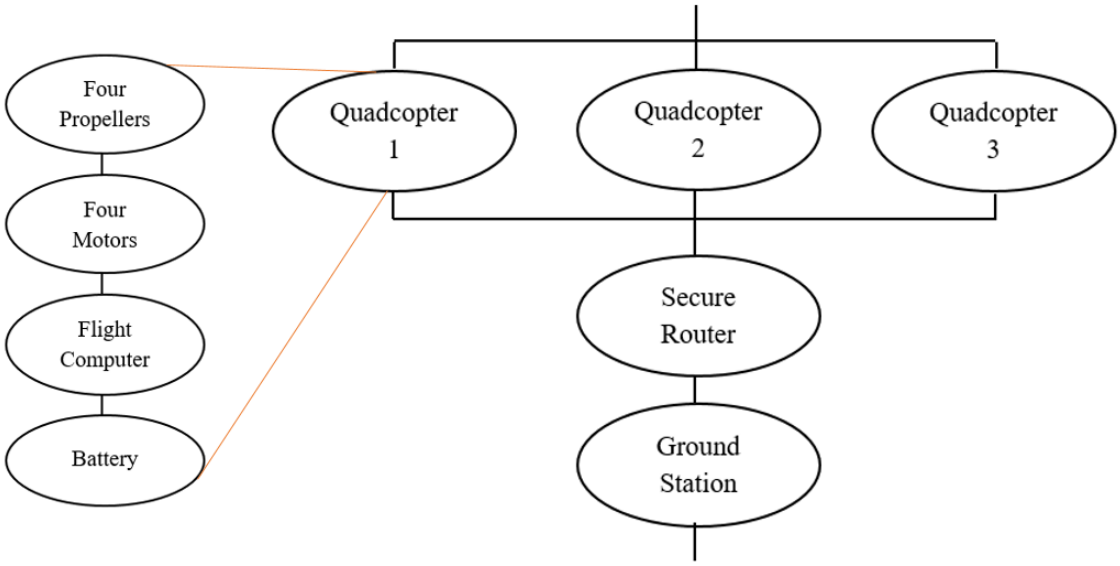


Figure 15. Individual Quadcopter Kill Tree versus SUAS Kill Tree

2. System Kill Tree

A kinetic kill of individual quadcopters would not disable the system unless all three of the quadcopters were lost, as seen in Figure 16. The loss of two quadcopters illustrates a partial kill. The system remains functional with a third quadcopter available for tasking. The ground control station would be located within a ship or ground unit and therefore be protected. The logical means of attacking the SUAS system is either with an overwhelming weapon that destroys the area, a surgical strike on the ground station, which may not be readily identifiable to the enemy, or the most likely scenario, a jamming system that floods the operating frequency (2.4 GHz) into the area. This may effectively disable the quadcopters and cause a mission kill.

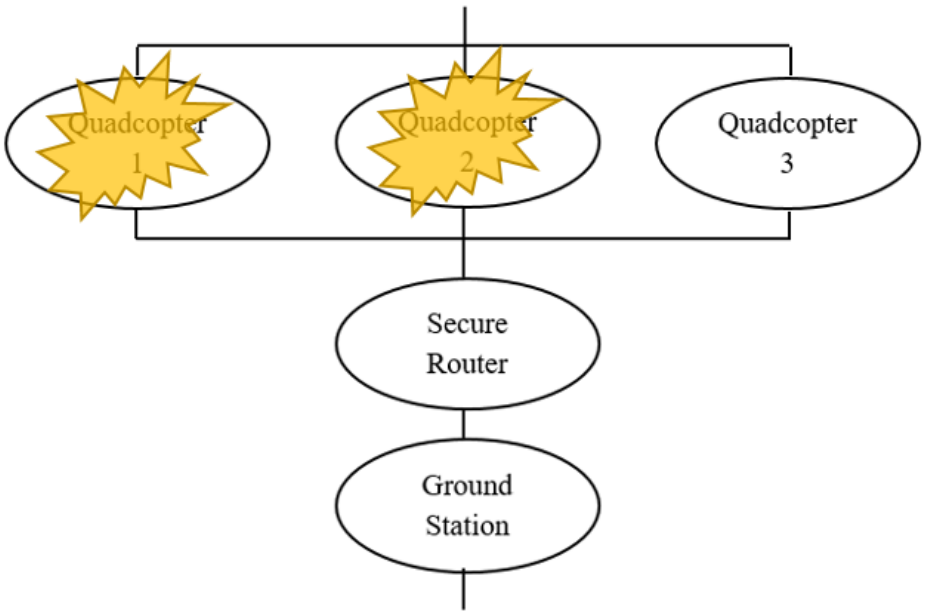


Figure 16. System Partial Kill

G. MISSION DEGRADATION DURING SYSTEM SWAP

While the system cannot be killed by the loss of a single vehicle, removing one quadcopter from flight requires time to conduct vehicle replacement. There would be a momentary lapse in coverage as the next available quadcopter is launched and moves into position. The SUAS vehicle swaps conducted during field testing in August 2017 show the system proof of concept. In field testing, three identical UAS launched individually to provide 54 minutes of continuous platform coverage through system swapping. If a single quadcopter is removed from the system, the replacement time would be approximately one minute. During this time, the system does not perform the intended mission.

THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSION AND FUTURE WORK

A. CONCLUSION

To address the problem of persistent platforms formulated, the multi-rotor paradigm and architecture was developed. The developed prototype was field tested at Camp Roberts August 7–8, 2017. These tests provided results which evaluated feasibility of the system based on performance.

The system of quadcopters is a feasible option to provide a persistent aerial platform. By monitoring battery health, vehicles intelligently swap to ensure the system always has an airborne asset. The system provides coverage in a loitering scenario that far exceeds the capability of a single quadcopter while also providing the user with a survivable asset.

Testing revealed certain limitations with the system, specifically with current battery and charging technology. Until the technology advances significantly, a store of batteries is necessary to keep the system airborne for extended periods of time. If the number of batteries available and the charging capacity can exceed the vehicle demands, the system could operate seemingly indefinitely, or at least until all vehicles mechanically fail.

The feasibility of the vehicle swapping system has been established. The system operates autonomously and only requires operators to change batteries to keep the platform in the air.

This multi-rotor system is robust, built on redundancy. The system is most susceptible to degradation if a quadcopter is killed in flight. To the mission commander operating this system, the one-minute gap may be unacceptable. This degradation is minimized with the low susceptibility of the system in operation. The likelihood that an enemy could remove a portion of the system effectively and conduct a strike in that gap requires a high level of coordination and does not present an imminent threat. The multi-rotor system is a survivable for the near future and ready for further testing and demonstration.

B. FUTURE WORK

Through development of the multiple quadcopter system, future work recommendations are as follows:

- Conduct extensive duration testing using at least the nine recommended batteries and chargers to determine when the system will fail.
- Test existing batteries and provide recommendations for advancement of battery life.
- Test flight patterns to find potential battery efficiencies.
- Develop sensors or video systems to integrate onto the aerial platforms and seamlessly string together between vehicles.
- Develop the system to further survive in combat scenarios. If one vehicle is lost in a maintenance or combat scenario, the system should be able to autonomously replace the vehicle.

With further development, this system can provide the full proof of concept for use with our armed forces and establish the technology for use with society. Battery technology is developing slowly, persistent technologies—such as vehicle swapping using intelligent battery monitoring—are crucial to keeping the United States military the most capable in the world.

APPENDIX A. 3DR SOLO STEP-BY-STEP CONFIGURATION

This step-by-step configuration guide was created during the process of developing this thesis. These are the steps used to allow multiple 3DR Solos to access a single wireless network. This closely follows the 3DR Solo Development Guide (https://dev.3dr.com) with notes that helped me get things working.

A. HOW TO ACCESS SOLO

These are the steps necessary to access the 3DR Solo controller and vehicle:

1) Accessing Solo. Reference: https://dev.3dr.com/starting-network.html

   a) When connected to the SoloLink wifi network (SoloLink-###) created by the 3DR Solo Controller and Quadcopter, you are able to find the following addresses:

      i) `10.1.1.1` — Controller

      ii) `10.1.1.10` — Solo

      iii) `10.1.1.100`–`10.1.1.255` — Computers, phones, or other devices on the network

   b) You are able to SSH into the controller:
   `ssh root@10.1.1.10`

      You will be prompted for the password: **TjSDBkAu**

      If you see an error "WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!", you can remove the keys to allow connection using the command:
      `ssh-keygen -R 10.1.1.10`

   c) If you have trouble logging in, it may be due to "Host Identification" error which can happen if you previously SSH'd into a different solo. See: https://dev.3dr.com/starting-network.html

d) Instructions for changing WiFi SSID and password: https://3drobotics.com/kb/setting-wifi-password/

B. PREPARING YOUR COMPUTER

1) Setting up your computer: Solo Command Line Tool (Solo CLI Tool). Reference: https://dev.3dr.com/starting-utils.html

a) The Solo CLI should be installed on the PC to execute solo commands (more on this in steps to follow). If it is already installed on the computer, you can skip step 2.

b) Connect to valid WiFi network with internet and run:

```
sudo -H pip install git https://github.com/3drobotics/solo-cli
```

c) Optional: If you get an error: "distutils.errors.DistutilsError: Setup script exited with error: command 'x86_64-linux-gnu-gcc' failed with exit status 1", as I did, run this and then try again:

```
sudo apt-get install libffi-dev
sudo apt-get install libssl-dev
sudo apt-get install python-dev
```

Also, if you get an error: "ImportError: No module named virtualenv", you should run:

```
sudo pip install virtualenv
```

d) Reconnect to the solo's wifi network. You can now run solo commands. Note that you MUST be connected to the solo network for these commands to execute properly. To see all solo commands, use command "solo":

```
$ solo
Usage:
 solo info
 solo wifi --name=<n> [--password=<p>]
 solo flash (drone|controller|both)
                   (latest|current|factory|<version>) [--clean]
 solo flash --list
 solo flash pixhawk <filename>
 solo provision
 solo resize
 solo logs (download)
 solo install-pip
```

```
solo install-smart
 solo install-runit
 solo video (acquire|restore)
 solo script [<arg>...]
```

i) First time running solo command and results:

```
solo info
```

Response:

```
connecting to Solo and the Controller...
{
  "controller": {
    "ref": "3dr-controller-imx6solo-3dr-artoo-20160926202703",
    "version": "2.4.2"
  },
  "gimbal": {
    "connected": false
  },
  "pixhawk": {
    "apm_ref": "7e9206cc",
    "px4firmware_ref": "5e693274",
    "px4nuttx_ref": "d48fa307",
    "version": "1.3.1"
  },
  "solo": {
    "ref": "3dr-solo-imx6solo-3dr-1080p-20160926202940",
    "version": "2.4.2"
  }
}
```

Error Response:

```
connecting to the Controller...
connecting to encrypted wifi network.
 (your computer may disconnect from Solo's network.)
please manually reconnect to Solo's network once it becomes available.
it may take up to 30s to a reconnect to succeed.
```

If this occurs, check that you are connected to the solo network and try again.

C. CONNECTING SOLO TO WIFI

1) Connecting the solo to a common network or network with internet. Reference: https://dev.3dr.com/starting-utils.html

a) Install the Solo CLI. This allows more control over the solo network.

b) In order to connect the solo to another network, ensure that you are still connected to the existing solo network (SoloLink-####) and run the following code replacing <ssid> with your WAP SSID and <password> with your WAP password:

```
solo wifi --name=<ssid> --password=<password>
```

For this system configuration:

```
solo wifi --name=solocommand --password=commandsolo
```

c) You should hear a sound from the vehicle and it will reboot. Another sound will indicate you are connected.

d) Your solo now has internet access.

D. MAKING SOLO MORE ACCESSIBLE (OPTIONAL)

1) Follow the procedures for Connecting Solo to Wi-Fi, then proceed.

2) First time solo CLI. Reference: https://dev.3dr.com/starting-utils.html

a) Ensure you are still connected to the solo network (you do not need to SSH in first).

b) The first time you run solo CLI, we want to install a few repositories on the solo. Install all of the following libraries.

i) smart

```
solo install-smart
```

ii) runit

```
solo install-runit
```

Troubleshooting: If you get the error shown below, reboot the controller, repeat steps 4.i through 4.ii

```
NOTE: this process requires simultaneous access to
Solo and to the Internet. if you have not yet done so,
run `solo wifi` to connect to Solo and to a local
wifi connection simultaneously.
```

```
connecting to solo...
waiting for Internet connectivity...

Loading cache...
Updating cache... ######################################## [100%]


error: busybox-1.21.1-r1@cortexa9hf_vfp_neon is not available for
downloading
```

iii) pip

```
solo install-pip
```

iv) DroneKit script pack

```
solo script pack
```

If successful, the command will create an archive in the `solo-script.tar.gz` in the current directory.

Deploy this archive to Solo and run a specified script using the `solo script run <scriptname>` command. The host computer must be connected to the Solo wifi network, and Solo must also be connected to the internet.

For example, to deploy and run the helloworld example:

```
solo script run helloworld.py
```

E. DRONEKIT-PYTHON WITH 3DR SOLO

1) Clone my working repository

```
git clone https://github.com/awilliams84/solo.git
```

2) Connect to the Solo network created when the controller and vehicle connect together (likely named SoloLink_<name>). On the Solo's network, you can connect to the vehicle as a UDP client 'udpin:0.0.0.0:14550'. Reference: https://dev.3dr.com/concept-dronekit.html

3) Run a simple script to ensure that everything is working. Navigate to the directory that contains solohello.py and execute the script. The key is declaring your vehicle.

```
python solohello.py
```

If successful, you should see a response the looks similar to Figure A-1.

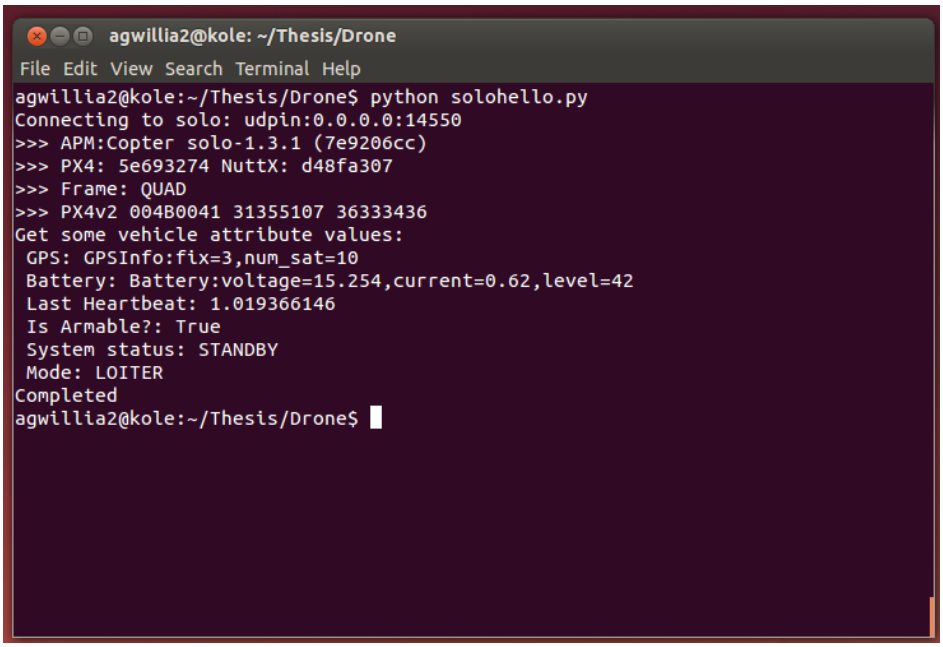


Figure A-1. Execution of solohello.py

F. SETTING UP FOR MULTIPLE 3DR SOLO

Figure A-2. Network Concept

These are step by step instructions modified from the 3DR Google document about operating swarms.

Reference: https://docs.google.com/document/d/1heLTpFEyNC_52BnZnz78lZxPeW15iZX98VUEXM4GIBM/edit#heading=h.ma685xmoqc83

Steps 1 will only be necessary the first time you are running through this tutorial. Once you have cloned the 3DR Git, you can skip step 1 for additional Solo vehicles.

1) Clone the 3DR github repository containing the installation extras (https://github.com/3drobotics/swarm):

```
mkdir ~/solo_ws/src
cd ~/solo_ws/src
git clone https://github.com/3drobotics/swarm.git
```

2) Modifying Solo's UDP broadcast port. Since Solo communicates with the ground station via UDP, we will have to change default port used by each Solo to make sure they do not interfere with each other. The first step is to layout which ports will be used by which Solo. The first Solo starts on port 15550, and the next on 16550, etc., as that leaves the default Solo port (14550) open in case someone accidentally connects their solo to our network.

Table A-1. Solo Port Assignments

Solo	Port
SoloLink_redleader	15550
SoloLink_blueleader	16550
SoloLink_goldleader	17550
SoloLink_greenleader	18550

With the ports laid out, change the drone's network configurations to broadcast on these ports. Start by connecting to the first Solo's network (usually SoloLink_###), then run these commands: (Note, the password of the artoo and the solo is TjSDBkAu)

Connect to vehicle:

```
ssh root@10.1.1.10
```

You will be prompted for the password: **TjSDBkAu**

Edit file on solo:

```
nano /etc/sololink.conf
```

In the sololink.conf file, change a parameter called “TelemDestPort”. The parameter should be set to the default port (14550); change it to the port you would like to assign.

Then save, close, and run (if you do not run this command, rebooting will reset the .conf file to the default)

```
md5sum /etc/sololink.conf > /etc/sololink.conf.md5
```

Next, reboot solo:

```
reboot
```

Connect to artoo:

```
ssh root@10.1.1.1
```

You will be prompted for the password: **TjSDBkAu**

Edit file on solo:

```
nano /etc/sololink.conf
```

In the sololink.conf file, change the same parameter called “TelemDestPort”. The parameter should be set to the default port; change it to the port you would like to assign.

Then save, close, and run (if you do not run this command, rebooting will reset the .conf file to the default)

```
md5sum /etc/sololink.conf > /etc/sololink.conf.md5
```

Next, reboot solo:

```
Reboot
```

3) Setup automatic network connection and port forwarding. Next, you'll need to set up your solo so it automatically connects to the swarm network and forwards its mavlink data. This process is automated by the make permanent script found in installer/make_permanent.py, which we already have from cloning the 3DR git repository.

   a) Run the make_permanent script:

```
python ~/solo_ws/src/swarm/installer/make_permanent.py
```

   b) Input the parameters:

      i) What is the ssid of the swarm network: **solocommand**

      ii) What is the passkey of the swarm network: **commandsolo**

      iii) What is the port you want your artoo to forward: **15550** (or whichever port you set for that specific solo)

   Your solo should reboot.

   OPTIONAL: If you would like to check whether the Solo is configured properly with your network, reconnect to the SoloLink_### network and SSH into artoo:

```
ssh root@10.1.1.1
```

   You will be prompted for the password: **TjSDBkAu**

   To check the network configuration:

```
ifconfig
```

You should see something that indicates you are connected to a wlan0 network, as shown in Figure A-3.

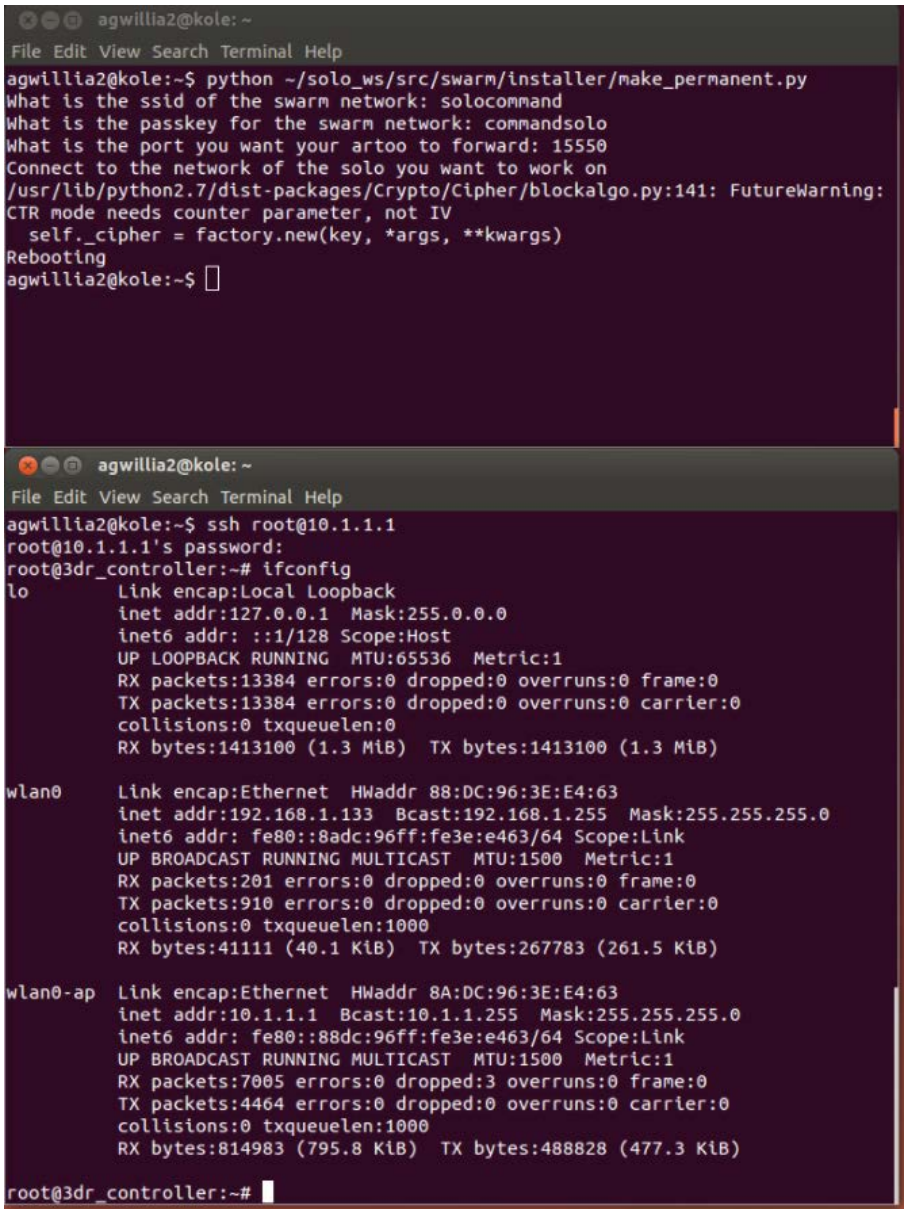


Figure A-3. Network Configuration

4) Record your IP address for each solo in your network using a table similar to Table C-2.

Table A-2. Network IP Address Assignments

Solo	IP solocommand	Port
SoloLink_redleader	192.168.1.133	15550
SoloLink_blueleader	192.168.1.121	16550
SoloLink_goldleader	192.168.1.138	17550
SoloLink_greenleader	192.168.1.106	18550

G. PREFLIGHT CHECKLIST

1) SSH into each Solo from your Router's network (To enable port forwarding)
2) Connect to Solo with a the swarm.launch file and use a unique, sequential copter_id
3) Run rostopic echo /copter#/mavros/rc/in to confirm sticks are publishing
4) Run your python file (For example python velocity_goto.py)
    a) python velocity_goto.py (takeoff and land)
    b) python dualo.py (fly 2 solos with one controller)
    c) python pong.py (play pong, requires 5 solos)

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. PYTHON SCRIPT

The following script was used to control three quadcopters demonstrated in field testing. Written in Python programming language, it launches vehicles and executes stationing commands using DroneKit-Python and includes data collection to an external file.

```
__author__ = "Alexander Williams"
__email__ = "alexander.williams@gmail.com"
__status__ = "Production"

'''''
This code was developed as part of the thesis:

Naval Postgraduate School
FEASIBILITY OF AN EXTENDED-DURATION AERIAL PLATFORM USING AUTONOMOUS
MULTI-ROTOR VEHICLE SWAPPING AND BATTERY MANAGEMENT

By LCDR Alexander Williams

This code was written and tested in 2017 at Camp Roberts. All
code has parameters set for testing at a specific site. If you
intend to use this code, ensure you change the parameters,
especially the latitude and longitude.

'''

# Import necessary libraries
from dronekit import connect, VehicleMode
from dronekit import LocationGlobalRelative, LocationGlobal, Command
import time
import math
from pymavlink import mavutil

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

import signal
import sys
from datetime import datetime, timedelta
from collections import defaultdict
import csv

from mpl_toolkits.mplot3d import Axes3D

# Initialize global variables
sortiedata=defaultdict(list)
solos=defaultdict(list)
sortie=0

#########################################################################
####################### ADJUSTABLE VARIABLES ############################
```

```
47. ########################################################################
48.
49. # Define all Solo ports and names (ports and names)
50. # If you add vehicles, this is where to add their ports and assign names for
51. # inclusion
52. soloports=['udpin:0.0.0.0:15550','udpin:0.0.0.0:16550','udpin:0.0.0.0:17550','ud
    pin:0.0.0.0:14550']
53. soloids=['redleader','blueleader','goldleader','greenleader']
54.
55. # Static test parameters
56.
57. # Min voltage we want to see from any vehicle
58. battery_volt_limit = 10
59. # Min battery level we want to see from any vehicle
60. battery_level_critical_limit = 30
61.
62. # Test location - modify for future application not at Camp Roberts
63. loiterlat = 35.7167982
64. loiterlon = -120.7625160
65. loiteralt = 100 #meters
66. # loiterlat = 35.716014
67. # loiterlon = -120.763119
68. # loiteralt = 30 #meters
69.
70. # Adjustable variables
71. takeoffalt = 15              # Altitude vehicles will go to on takeoff
72. vehiclealtseparation = 5     # Seperation distance (in meters) at swap
73.
74. ########################################################################
75. ###################### END ADJUSTABLE VARIABLES ########################
76. ########################################################################
77.
78. # Initialize time for data collection
79. # DELETE
80. # starttime = datetime.now()
81.
82.
83. # Function for arming and takeoff to set altitude
84. def arm_and_takeoff(vehicle,aTargetAltitude,vehiclename):
85.
86.     print "Basic pre-arm checks"
87.     # Do not try to arm until autopilot is ready
88.     while not vehicle.is_armable:
89.         print " Waiting for vehicle to initialise..."
90.         time.sleep(1)
91.
92.
93.     print "Arming motors"
94.     # Copter should arm in GUIDED mode
95.     vehicle.mode = VehicleMode("GUIDED")
96.     vehicle.armed = True
97.
98.     while not vehicle.armed:
99.         print " Waiting for arming..."
100.                time.sleep(1)
101.
102.            print "Taking off! Heading to ",aTargetAltitude
103.
104.            takeofftime = datetime.now()
105.            landedalt=vehicle.location.global_relative_frame.alt
```

```
            global sortie
            sortie+=1

            # For data logging
            storesortiedata(vehiclename,vehicle)

            # Wait until the vehicle reaches a safe height before
            # processing the goto (otherwise the command
            # after Vehicle.simple_takeoff will execute immediately).
            while True:
                vehicle.simple_takeoff(aTargetAltitude) # Take off to target alt
    itude
                timeelapsed=((datetime.now()-takeofftime).total_seconds())
                print " Altitude: ", vehicle.location.global_relative_frame.alt,
     " Time Elapsed: ",timeelapsed, " sec"
                #Trigger just below target alt
                if vehicle.location.global_relative_frame.alt>=aTargetAltitude*0
    .95:
                    print "Reached target altitude in ",timeelapsed," sec"
                    status="success"
                    break
                if timeelapsed>3 and vehicle.location.global_relative_frame.alt<
    landedalt+1:
                    print "!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!Launch ERROR!!!!!!!!!!
    !!!!!!!!!!!!!!!!!!!!"
                    print "Land vehicle (%s)" %(vehiclename)
                    vehicle.mode = VehicleMode("LAND")
                    status="fail"
                    break
                time.sleep(1)

            # For data logging
            storesortiedata(vehiclename,vehicle)

            return status


        # Function to calculate the distance in meters between two position
        def get_distance_metres(aLocation1, aLocation2):
            """
            Returns the ground distance in metres between two LocationGlobal obj
    ects.

            This method is an approximation, and will not be accurate over
            large distances and close to the earth's poles. It comes from the
            ArduPilot test code:
            https://github.com/diydrones/ardupilot/blob/master/Tools/autotest/co
    mmon.py
            """
            dlat = aLocation2.lat - aLocation1.lat
            dlong = aLocation2.lon - aLocation1.lon
            return math.sqrt((dlat*dlat) + (dlong*dlong)) * 1.113195e5


        # Function to store data during testing
        def storesortiedata(vehiclename,vehicle):
            timenow = datetime.now()
            global sortie

            sortiename=vehiclename
```

```
            sortiedata[sortiename].append(
                (timenow,
                    vehicle.location.global_relative_frame.lat,
                    vehicle.location.global_relative_frame.lon,
                    vehicle.location.global_relative_frame.alt,
                    vehicle.battery.level,
                    vehicle.battery.voltage,
                    sortiename
                    ))

            print "Stored data for %s in dictionary"%(vehiclename)


        # Function to write data to csv file and plot data
        def showplot():
            if len(sortiedata)>0:
                fig=plt.figure()
                fig.suptitle('Battery Management Testing')
                plt1=fig.add_subplot(211)
                plt2=fig.add_subplot(212)
                fig2=plt.figure()
                plt3=fig2.add_subplot(111,projection='3d')

                # Open csv file and append
                f = open('data.csv','a')

                fulltime=[]
                fulllatitude=[]
                fulllongitude=[]
                fullaltitude=[]
                fullbatterypct=[]
                fullbatteryvolt=[]
                fullsolo=[]

                global starttime

                for key, value in sorted(sortiedata.iteritems()):
                    soloid=key
                    time=[]
                    latitude=[]
                    longitude=[]
                    altitude=[]
                    batterypct=[]
                    batteryvolt=[]
                    solo=[]

                    for var in value:
                        time.append((var[0]-starttime).total_seconds())
                        latitude.append(var[1])
                        longitude.append(var[2])
                        altitude.append(var[3])
                        batterypct.append(var[4])
                        batteryvolt.append(var[5])
                        solo.append(var[6])

                    plt1.plot(time,altitude,label=soloid)
                    plt1.set_xlabel('Time (sec)')
                    plt1.set_ylabel('Altitude (m)')
                    plt2.plot(time,batterypct,label=soloid)
                    plt2.set_xlabel('Time (sec)')
```

```
                    plt2.set_ylabel('Battery Life (%)')
                    plt3.plot(latitude,longitude,altitude)
                    plt3.set_xlabel('Latitude (m)')
                    plt3.set_xlabel('Longitude (m)')
                    plt3.set_xlabel('Altitude (m)')

                    plt1.scatter(time[0],altitude[0],marker='o',color='g',s=50)

                    plt1.scatter(time[-1],altitude[-1],marker='o',color='r',s=50)
                    plt2.scatter(time[0],batterypct[0],marker='o',color='g',s=50)
                    plt2.scatter(time[-1],batterypct[-1],marker='o',color='r',s=50)

                    fulltime.extend(time)
                    fulllatitude.extend(latitude)
                    fulllongitude.extend(longitude)
                    fullaltitude.extend(altitude)
                    fullbatterypct.extend(batterypct)
                    fullbatteryvolt.extend(batteryvolt)
                    fullsolo.extend(solo)

                duration=math.ceil(fulltime[-1]/60)
                start=starttime.strftime('%Y%m%d %H%M')
                stoptime=var[0]
                stop=stoptime.strftime('%Y%m%d %H%M')
                flightdata=['Start: '+start,'Stop: '+stop,'Duration: '+`duration`+ ' minutes','Sortie Count: '+`len(sortiedata)`]

                # Add row titles
                fulltime.insert(0,'Time (sec)')
                fullsolo.insert(0,'Vehicle ID')
                fulllatitude.insert(0,'Latitude')
                fulllongitude.insert(0,'Longitude')
                fullaltitude.insert(0,'Altitude (m)')
                fullbatterypct.insert(0,'Battery (%)')
                fullbatteryvolt.insert(0,'Battery (V)')

                # Write to csv file
                w = csv.writer(f, delimiter=',')
                w.writerow(flightdata)
                w.writerow(fullsolo)
                w.writerow(fulltime)
                w.writerow(fulllatitude)
                w.writerow(fulllongitude)
                w.writerow(fullaltitude)
                w.writerow(fullbatterypct)
                w.writerow(fullbatteryvolt)

                print "############## Data stored to CSV ##############"

                # Close csv file
                f.close()

                plt1.autoscale(enable=True, axis='both', tight=False)
                plt2.autoscale(enable=True, axis='both', tight=False)


```

```
274.        # Initialize lists of available Solos (connect and names) set in
275.        # connection (set empty here)
276.        solo=[]
277.        soloname=[]
278.
279.        print "======================== CONNECTING TO SOLOS ====================
    ====="
280.
281.        i=0
282.
283.        # Loop through solo ports to attempt connection with each vehicle
284.        for soloport in soloports:
285.
286.            index=soloports.index(soloport)
287.            soloid=soloids[index]
288.
289.            print("Connecting to solo: %s (%s)" % (soloport,soloid))
290.
291.            # Try to connect to the solo for XX heartbeat_timeout seconds
292.            try:
293.                vehicle=connect(soloport, wait_ready=True, heartbeat_timeout=90)

294.                solo.append(vehicle)
295.                soloname.append(soloid)
296.
297.                solos[soloid]=vehicle
298.
299.                # Print some vehicle attributes
300.                print "Get some vehicle attribute values:"
301.                print " GPS: %s" % solo[i].gps_0
302.                print " Battery: %s" % solo[i].battery
303.                print " Last Heartbeat: %s" % solo[i].last_heartbeat
304.                print " Is Armable?: %s" % solo[i].is_armable
305.                print " System status: %s" % solo[i].system_status.state
306.                print " Mode: %s" % solo[i].mode.name
307.
308.                # Increment 1 to look on the next solo port
309.                i+=1
310.
311.            except:
312.                print (soloid+" vehicle not found.")
313.
314.
315.        print "////////////////////// END CONNECTING TO SOLOS /////////////////
    ////"
316.
317.
318.        # Write which solos were found
319.        print "%s Solos available: " %len(solo)
320.        print soloname
321.
322.        # A check to see if the script should continue running in loop to follow

323.        continuecheck = 1
324.
325.        # Ask user whether to continue where 'y' and 'n' are only acceptable ans
    wers
326.        while continuecheck != 'y' and continuecheck != 'n':
327.            continuecheck = raw_input('Continue? (y/n):')
328.
```

```
329.        # Create new dictionary variables for home lat and longs
330.        homelat=defaultdict(list)
331.        homelon=defaultdict(list)
332.
333.        vehicle=0
334.        sortienum=-1
335.        unavailablevehicles=0
336.        takeoffstatus="true"
337.
338.
339.
340.        # Main script for flight operations
341.        try:
342.
343.            vehicleindex=0
344.            vehicleindex2=1
345.
346.            # Initialize time for data collection
347.            starttime = datetime.now()
348.
349.            # Loop through vehicles
350.            while True and continuecheck == 'y':
351.
352.                for thisvehicle in solo:
353.
354.                    if vehicleindex>len(solo)-1:
355.                        vehicleindex=0
356.
357.                    print "vehicleindex=%s" %(vehicleindex)
358.                    print "vehicleindex2=%s" %(vehicleindex2)
359.
360.                    if vehicle==solo[vehicleindex]:
361.                        if takeoffstatus=="fail":
362.                            print "Sleep to standby for another attempt. Same ve
    hicle."
363.                            time.sleep(15)
364.                        else:
365.                            print 'Vehicle Check: %s already flying, cannot use
    same vehicle to start while airborne' %(soloname[vehicleindex])
366.                            continuecheck='n'
367.                            break
368.                    if solo[vehicleindex].battery.voltage < battery_volt_limit o
    r solo[vehicleindex].battery.level < battery_level_critical_limit:
369.                        print 'Vehicle Check: %s battery too low for operation,
    skip to next' %(soloname[vehicleindex])
370.                        vehicleindex+=1
371.                        unavailablevehicles+=1
372.                        break
373.                    elif unavailablevehicles>len(solo):
374.                        print 'Vehicle Check: No vehicles available.'
375.                        continuecheck='n'
376.                        break
377.
378.                    vehicle=solo[vehicleindex]
379.                    sortienum+=1
380.
381.                    # Dynamic test parameters
382.                    battery_level_limit=0 # Set higher for testing
383.
384.                    print("Connected to solo: %s" % (soloname[vehicleindex]))
```

```
385.
386.                     # If connection is not current, skip vehicle and go to next
387.                     if vehicle.last_heartbeat>10:
388.                         vehicleindex+=1
389.                         break
390.
391.                     # If the next vehicle does not have enough battery
392.                     if vehicle.battery.level < battery_level_critical_limit:
393.                         print " %s battery too low for designated flight." %(sol
    oname[vehicleindex])
394.
395.                     # Else, continue
396.                     else:
397.
398.                         # If vehicle is not already airborne (likely same vehicl
    e)
399.                         if vehicle.location.global_relative_frame.alt < takeoffa
    lt-1:
400.
401.                             try:
402.                                 cmds = vehicle.commands
403.                                 cmds.download()
404.                                 cmds.wait_ready()
405.
406.                                 # Arm and take off - receive status
407.                                 takeoffstatus=arm_and_takeoff(vehicle,takeoffalt
    ,soloname[vehicleindex])
408.
409.                                 # If launch fail
410.                                 if takeoffstatus=="fail":
411.                                     print "Launch failed, vehicleindex=%s" %(veh
    icleindex)
412.                                     vehicleindex+=1
413.                                     print "Find next vehicle, vehicleindex=%s" %
    (vehicleindex)
414.                                     break
415.
416.                             except:
417.                                 if takeoffstatus=="fail":
418.                                     print "Take off aborted, vehicleindex=%s" %(
    vehicleindex)
419.                                 vehicleindex+=1
420.                                 print "Find next vehicle, new vehicleindex=%s" %
    (vehicleindex)
421.                                 break
422.
423.                             # Increasing vehicle airspeed to max (30)
424.                             print "Set vehicle airspeed to 30"
425.                             vehicle.airspeed = 30
426.
427.                             # Set home location as the location above takeoff
428.                             # (for return later)
429.                             homelat[vehicleindex]=(vehicle.location.global_frame
    .lat)
430.                             homelon[vehicleindex]=(vehicle.location.global_frame
    .lon)
431.
432.                             # Print for debug
433.                             print " Home Latitude: %s" % homelat[vehicleindex]
```

```
434.                             print " Home Longitude: %s" % homelon[vehicleindex]
435.
436.                         else:
437.                             print "Vehicle (%s) already airborne" %(soloname[veh
    icleindex])
438.
439.                         # Mission location
440.                         loiterpoint = LocationGlobalRelative(loiterlat,loiterlon
    , loiteralt)
441.                         vehicle.simple_goto(loiterpoint)
442.
443.
444.                         print "*****************Go to loiter point*************
    ***"
445.
446.                         storesortiedata(soloname[vehicleindex],vehicle)
447.
448.                         # Loop to track distance to loiter. Print distance to go
     until
449.                         # at loiter
450.                         while True:
451.                             currentaltitude=vehicle.location.global_relative_fra
    me.alt
452.
453.                             dist = get_distance_metres(vehicle.location.global_f
    rame, loiterpoint)
454.                             print " Distance to go: %s Altitude goal: %s Current
    : %s"%(dist,loiteralt,currentaltitude)
455.
456.                             if currentaltitude>loiteralt:
457.                                 goalpct=1.02
458.                             else:
459.                                 goalpct=0.98
460.
461.                             if dist < 1 and currentaltitude>=(loiteralt)*0.98:
462.                                 print 'Within 1m of waypoint'
463.                                 break
464.
465.                             time.sleep(1)
466.
467.                         storesortiedata(soloname[vehicleindex],vehicle)
468.
469.
470.                         print "*******************Maintain loiter point********
    **********"
471.
472.                         # Maintain loiter position until battery level too low
473.                         while True:
474.
475.                             try:
476.                                 print ' %s Battery limit: %s  Current battery: %
    s'%(soloname[vehicleindex],max(battery_level_limit,battery_level_critical_limit)
    ,vehicle.battery.level)
477.                                 time.sleep(1)
478.
479.                                 # If battery level below limit or voltage level
    below limit, break loop
```

```
                                if vehicle.battery.voltage < battery_volt_limit or vehicle.battery.level < battery_level_limit or vehicle.battery.level < battery_level_critical_limit:
                                    print 'Battery below limit'

                                    break

                            except:
                                # When battery loop is manually exited (CTRL+C)

                                print " BATTERY CHECK ABORTED"
                                break

                        storesortiedata(soloname[vehicleindex],vehicle)

                        if vehicleindex==len(solo)-1:
                            vehicle2=solo[0]
                            vehicleindex2=0
                        else:
                            vehicle2=solo[vehicleindex+1]
                            vehicleindex2=vehicleindex+1

                        # Look for replacement if more than one solo online
                        if len(solo) > 1:
                            print 'Checking for replacement'

                            vehiclecheck=0
                            while vehiclecheck <= len(solo) and (vehicle2.battery.level < battery_level_critical_limit or vehicle==vehicle2 or vehicle2.last_heartbeat>5):
                                if vehicleindex==len(solo)-1:
                                    vehicle2=solo[0]
                                    vehicleindex2=0
                                else:
                                    vehicle2=solo[vehicleindex+1]
                                    vehicleindex2=vehicleindex+1
                                vehiclecheck+=1
                                print "Vehicle check count: %s"%(vehiclecheck)

                            # Check if replacements are able to replace (battery checks)
                            if vehiclecheck > len(solo) or vehicle2.battery.level < battery_level_critical_limit:
                                replacementfails=vehiclecheck
                                # break
                            else:
                                replacementfails=0

                            # Reset takeoffstatus before trying another takeoff

                            takeoffstatus="fail"

                            # Start takeoff sequence until a replacement is launched or all fail
                            while takeoffstatus=="fail" and replacementfails<len(solo):
                                if vehicleindex2==vehicleindex:
                                    vehicleindex2+=1
                                if vehicleindex2>len(solo)-1:
                                    vehicleindex2=0
```

```
                                if solo[vehicleindex2].last_heartbeat<5:
                                    print 'Launch replacement'
                                    print 'Launch vehicleindex2=%s' %(vehicleind
    ex2)
                                    try:
                                        takeoffstatus=arm_and_takeoff(solo[vehic
    leindex2],takeoffalt,soloname[vehicleindex2])
                                    except:
                                        takeoffstatus="fail"

                                nextvehicle=vehicleindex2+1
                                if nextvehicle>len(solo):
                                    nextvehicle=0

                                if nextvehicle==vehicleindex2 or nextvehicle==ve
    hicleindex:
                                    time.sleep(15)

                                if takeoffstatus=="fail":
                                    print 'Launch replacement failed'
                                    vehicleindex2+=1
                                    replacementfails+=1
                                    # break

                            vehicle2=solo[vehicleindex2]

                            if replacementfails>len(solo):
                                print "Replacement Launch Failed %s times, quitt
    ing" %(replacementfails)
                                continuecheck='n'
                            else:

                                print "Set vehicle airspeed to 30"
                                vehicle2.airspeed = 30

                                storesortiedata(soloname[vehicleindex],vehicle)

                                storesortiedata(soloname[vehicleindex2],vehicle2
    )

                                if vehicleindex2>len(homelat)-1:
                                    print "Adding home point for %s" %(soloname[
    vehicleindex2])

                                    cmds = vehicle2.commands
                                    cmds.download()
                                    cmds.wait_ready()

                                    homelat[vehicleindex2]=(vehicle2.location.gl
    obal_frame.lat)
                                    homelon[vehicleindex2]=(vehicle2.location.gl
    obal_frame.lon)

                                    print 'Home: (%s,%s)' %(homelat[vehicleindex
    2],homelon[vehicleindex2])

                                else:
                                    print "Home already exists, ignoring!"

```

```
578.                             print "**************Vehicle: %s airborne, takeoffst
    atus=%s"%(soloname[vehicleindex2],takeoffstatus)
579.
580.                             # Raise original vehicle
581.                             print 'Raising vehicle from %s to %s' %(vehicle.loca
    tion.global_relative_frame.alt,loiteralt+vehiclealtseparation)
582.                             highloiterpoint = LocationGlobalRelative(loiterlat,l
    oiterlon, loiteralt+vehiclealtseparation)
583.                             vehicle.simple_goto(highloiterpoint)
584.
585.                             # Loop to track raised altitude
586.                             while True and takeoffstatus!="fail":
587.                                 currentaltitude=vehicle.location.global_relative
    _frame.alt
588.
589.                                 print ' Altitude goal: %s Current: %s' %(loitera
    lt+vehiclealtseparation,currentaltitude)
590.                                 if currentaltitude>loiteralt+vehiclealtseparatio
    n:
591.                                     goalpct=1.02
592.                                 else:
593.                                     goalpct=0.98
594.                                 if currentaltitude>=(loiteralt+vehiclealtseparat
    ion)*goalpct:
595.                                     break
596.
597.                                 time.sleep(1)
598.
599.                             # Changing to guided mode in case vehicle coming onl
    ine is not already
600.                             print 'Change mode to guided'
601.                             vehicle.mode = VehicleMode("GUIDED")
602.
603.                             # If repacement launched, send to loiter point
604.                             if len(solo) > 1 and takeoffstatus!="fail":
605.
606.                                 print 'Send replacement'
607.
608.                                 storesortiedata(soloname[vehicleindex],vehicle)

609.                                 storesortiedata(soloname[vehicleindex2],vehicle2
    )
610.
611.                                 while True:
612.                                     vehicle2.simple_goto(loiterpoint)
613.
614.                                     currentaltitude=vehicle2.location.global_rel
    ative_frame.alt
615.
616.                                     dist = get_distance_metres(vehicle2.location
    .global_frame, loiterpoint)
617.                                     print " Distance to loiter pt: %s Altitude g
    oal: %s Current: %s"%(dist,loiteralt,currentaltitude)
618.
619.                                     if currentaltitude>loiteralt:
620.                                         goalpct=1.02
621.                                     else:
622.                                         goalpct=0.98
623.
```

```
624.                                     if dist < 1 and currentaltitude>=(loiteralt)
    *goalpct:
625.                                         print 'Within 1m of waypoint'
626.                                         break
627.
628.                                     time.sleep(1)
629.
630.
631.                         print 'Check altitude: %s' %(vehicle.location.global_rel
    ative_frame.alt)
632.                         print 'Take off alt minus 2: %s' %(takeoffalt-2)
633.
634.                     # If original vehicle still airborn (should be), return to l
    aunch position
635.                     # this check is conducted using vehicle's altitude. If loite
    r position is lower
636.                     # than takeoff alt, this will need to be rewritten
637.                     if vehicle.location.global_relative_frame.alt > takeoffalt-
    2:
638.
639.                         # Return to launch point
640.                         print 'Vehicle index=%s' %(vehicleindex)
641.                         print 'Returning to launch position: (%s,%s)' %(homelat[
    vehicleindex],homelon[vehicleindex])
642.
643.                         homepoint = LocationGlobalRelative(homelat[vehicleindex]
    ,homelon[vehicleindex], takeoffalt+vehiclealtseparation)
644.                         vehicle.simple_goto(homepoint)
645.
646.                         print " Go to %s" %(homepoint)
647.
648.                         storesortiedata(soloname[vehicleindex],vehicle)
649.                         storesortiedata(soloname[vehicleindex2],vehicle2)

650.
651.                         while True:
652.                             currentaltitude=vehicle.location.global_relative_fra
    me.alt
653.
654.                             dist = get_distance_metres(vehicle.location.global_f
    rame, homepoint)
655.                             print " Distance to home: %s Altitude goal: %s Curre
    nt: %s"%(dist,takeoffalt+vehiclealtseparation,currentaltitude)
656.
657.                             if currentaltitude>loiteralt+vehiclealtseparation:
658.                                 goalpct=1.02
659.                             else:
660.                                 goalpct=0.98
661.
662.                             if dist < 1 and currentaltitude<=(loiteralt+vehiclea
    ltseparation)*goalpct:
663.                                 print 'Within 1m of waypoint'
664.                                 break
665.
666.                             time.sleep(1)
667.
668.                         # Land original vehicle
669.                         print ' Landing'
670.                         vehicle.mode = VehicleMode("LAND")
671.
```

```
672.                         storesortiedata(soloname[vehicleindex],vehicle)
673.                         storesortiedata(soloname[vehicleindex2],vehicle2)
674.
675.                         # Loop to track landing. Was used for data storing
676.                         while True:
677.                             print " Landing %s Altitude: %s" %(soloname[vehiclei
    ndex],vehicle.location.global_relative_frame.alt)
678.
679.                             if vehicle.location.global_relative_frame.alt<=2: #T
    rigger just below target alt.
680.                                 print "Vehicle On Deck"
681.                                 break
682.                             time.sleep(1)
683.
684.                         storesortiedata(soloname[vehicleindex],vehicle)
685.
686.                     # showplot()
687.                     vehicleindex+=1
688.
689.                 if unavailablevehicles >= len(solo):
690.                     break
691.
692.
693.         # If interrupt
694.         except:
695.
696.             print "\n\n\n*************************Exited script early.*********
    ******************"
697.
698.             vehicleindex=0
699.             for vehicle in solo:
700.                 print "%s control released. Returned to LOITER mode"%(soloname[v
    ehicleindex])
701.                 vehicle.mode = VehicleMode("LOITER")
702.                 vehicleindex+=1
703.
704.             showplot()
705.             plt.show(block=True)
706.             sys.exit(0)
707.
708.
709.         # If script errors out
710.         if continuecheck=='y' and len(solo)==0:
711.             print 'No solo present for operation. Exited script.'
712.         elif continuecheck=='n':
713.             print 'Exited script.'
714.         else:
715.             print "\n\n\n************************Errored out of script early.**
    ************************"
716.
717.             vehicleindex=0
718.             for vehicle in solo:
719.                 print "%s control released. Returned to LOITER mode"%(soloname[v
    ehicleindex])
720.                 vehicle.mode = VehicleMode("LOITER")
721.                 vehicleindex+=1
722.
723.         if len(solo)!=0:
724.             showplot()
```

```
725.            plt.show(block=True)
726.            sys.exit(0)
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. FIELD TESTING RAW DATA

Table C-1 presents raw data collected at Camp Roberts, California, August 8, 2017.

Table C-1. Field Testing Results

				Altitude (m)				
ID	**Time (MM:SS)**	**Latitude**	**Longitude**	**Veh 1**	**Veh 2**	**Veh 3**	**Battery (%)**	**Battery (V)**
Vehicle 1	00:03	35.7156929	-120.7635823	0.01			96	16.214
Vehicle 1	00:11	35.7156898	-120.7635802	14.55			95	16.113
Vehicle 1	00:48	35.7167919	-120.7625102	99.88			91	15.774
Vehicle 1	11:17	35.7167971	-120.7625096	100.01			29	14.405
Vehicle 2	11:20	35.7155735	-120.7635051	100.01	0.01		95	16.457
Vehicle 2	11:28	35.715574	-120.76351	100.01	14.55		94	16.215
Vehicle 1	11:28	35.7167979	-120.762511	99.98	14.55		28	14.4
Vehicle 2	11:28	35.715574	-120.76351	99.98	14.55		94	16.215
Vehicle 1	11:32	35.7168006	-120.7625096	103.49	14.55		28	14.358
Vehicle 2	11:32	35.7155716	-120.7635059	103.49	14.9		94	16.149
Vehicle 1	12:11	35.7167976	-120.76251	104.97	14.9		24	14.312
Vehicle 2	12:11	35.7167941	-120.7625084	104.97	99.96		90	15.899
Vehicle 1	13:04	35.7156978	-120.7635808	20.39	99.96		19	14.215
Vehicle 2	13:04	35.7167983	-120.7625094	20.39	100.07		85	15.771
Vehicle 1	13:04	35.7156978	-120.7635808	20.39	100.07		19	14.215
Vehicle 1	13:28	35.7156866	-120.7635829	1.99	100.07		7	14.256
Vehicle 2	13:28	35.716797	-120.7625081		100.03		83	15.672
Vehicle 2	13:28	35.716797	-120.7625081		100.03		83	15.672
Vehicle 2	22:22	35.7167982	-120.7625094		99.97		29	14.537
Vehicle 3	22:25	35.7155368	-120.7633986		99.97	0.01	95	16.327
Vehicle 3	22:33	35.7155362	-120.7633981		99.97	14.61	94	16.236
Vehicle 2	22:33	35.7167964	-120.7625089		100.03	14.61	28	14.512
Vehicle 3	22:33	35.7155362	-120.7633981		100.03	14.61	94	16.236
Vehicle 2	22:36	35.7167974	-120.7625098		103.45	14.61	28	14.484
Vehicle 3	22:36	35.7155349	-120.7633984		103.45	14.95	94	16.165
Vehicle 2	23:11	35.7167969	-120.7625086		105	14.95	24	14.47
Vehicle 3	23:11	35.716791	-120.7625165		105	99.68	89	15.895
Vehicle 2	24:05	35.7155795	-120.7635053		19.88	99.68	19	14.426
Vehicle 3	24:05	35.7167971	-120.7625085		19.88	100.05	84	15.737
Vehicle 2	24:05	35.7155795	-120.7635053		19.88	100.05	19	14.426
Vehicle 2	24:29	35.7155807	-120.7635056		1.76	100.05	17	14.567
Vehicle 3	24:29	35.716797	-120.7625075			100.02	81	15.633
Vehicle 3	24:29	35.716797	-120.7625075			100.02	81	15.633

				Altitude (m)				
ID	Time (MM:SS)	Latitude	Longitude	Veh 1	Veh 2	Veh 3	Battery (%)	Battery (V)
Vehicle 3	32:34	35.7167968	-120.7625096			99.97	29	14.629
Vehicle 1	32:37	35.715692	-120.763558	0.01		99.97	94	16.45
Vehicle 1	32:45	35.7156886	-120.7635546	14.67		99.97	93	16.162
Vehicle 3	32:45	35.716796	-120.7625091	14.67		100.1	28	14.616
Vehicle 1	32:45	35.7156886	-120.7635546	14.67		100.1	93	16.162
Vehicle 3	32:48	35.7167977	-120.7625106	14.67		103.58	28	14.562
Vehicle 1	32:48	35.7156896	-120.7635573	14.89		103.58	93	16.109
Vehicle 3	33:24	35.7167964	-120.7625094	14.89		104.94	24	14.564
Vehicle 1	33:24	35.7167932	-120.762512	99.98		104.94	89	15.848
Vehicle 3	34:19	35.7155351	-120.7633912	99.98		20.23	19	14.539
Vehicle 1	34:19	35.7168003	-120.76251	99.92		20.23	84	15.694
Vehicle 3	34:19	35.7155351	-120.7633912	99.92		20.23	19	14.539
Vehicle 3	34:42	35.7155326	-120.7633931	99.92		1.63	17	14.656
Vehicle 1	34:43	35.7167928	-120.7625049	100.06			82	15.599
Vehicle 1	34:46	35.7167978	-120.7625091	100.07			82	15.612
Vehicle 1	43:24	35.7167965	-120.7625094	99.98			29	14.541
Vehicle 2	43:28	35.7155635	-120.7635258	99.98	0.1		95	16.278
Vehicle 2	43:36	35.7155628	-120.7635253	99.98	14.65		94	16.082
Vehicle 1	43:36	35.7167962	-120.7625093	100	14.65		28	14.531
Vehicle 2	43:36	35.7155628	-120.7635253	100	14.65		94	16.082
Vehicle 1	43:39	35.7167941	-120.7625073	103.48	14.65		28	14.527
Vehicle 2	43:39	35.7155639	-120.7635255	103.48	14.95		94	16.011
Vehicle 1	44:16	35.7167969	-120.7625075	105.07	14.95		24	14.486
Vehicle 2	44:16	35.7167925	-120.7625099	105.07	99.91		90	15.813
Vehicle 1	45:10	35.7156956	-120.7635865	20.2	99.91		20	14.433
Vehicle 2	45:10	35.7167985	-120.7625085	20.2	100.17		85	15.617
Vehicle 1	45:10	35.7156956	-120.7635865	20.2	100.17		20	14.433
Vehicle 1	45:33	35.7156894	-120.7635844	1.92	100.17		18	14.384
Vehicle 2	45:33	35.7167977	-120.7625088		100.05		83	15.511
Vehicle 2	45:33	35.7167977	-120.7625088		100.05		83	15.511
Vehicle 2	54:11	35.7167977	-120.7625084		100.02		29	14.413
Vehicle 2	54:11	35.7167977	-120.7625084		100.02		29	14.413
Vehicle 3	54:11	35.715537	-120.7633971		100.02	2.7	17	14.793
Vehicle 2	55:03	35.7155802	-120.7635055		19.9	2.7	25	14.348
Vehicle 3	55:03	35.715537	-120.7633971		19.9	2.7	17	14.793
Vehicle 2	55:03	35.7155802	-120.7635055		19.9	2.7	25	14.348
Vehicle 2	55:25	35.7155778	-120.7635076		1.95	2.7	22	14.338

LIST OF REFERENCES

3DR. 2015. “Solo Specs: Just the Facts.” May 4, 2015. https://3dr.com/blog/solo-specs-just-the-facts-14480cb55722.

Adams, Christopher. 2017. “Designing for Survivability.” Class notes for ME4751: Combat Survivability, Reliability and System Safety Engineering. Naval Postgraduate School, Monterey, CA.

Ball, Robert. 2003. *The Fundamentals of Ship Combat Survivability Analysis and Design*. 2nd ed. Reston, VA: AIAA.

Baraniuk, Chris. 2017. “Small Drone ‘Shot with Patriot Missile.’” BBC. March 15, 2017. http://www.bbc.com/news/technology-39277940.

Defense Advanced Research Projects Agency. 2017. “Tern.” Accessed October 9, 2017. https://www.darpa.mil/program/tern.

Defense Science Board. 2012. *The Role of Autonomy in DOD Systems.* Washington, DC: Office of the Under Secretary of Defense for Acquisition, Technology and Logistics. https://www.acq.osd.mil/dsb/reports/2010s/AutonomyReport.pdf.

Gettinger, Dan. 2016. *Drone Spending in the Fiscal Year 2017 Defense Budget*. Annandale-on-Hudson, NY: Center for the Study of the Drone at Bard College.

Holland, Terry. 2015. “The 3D Robotics Solo: A DroneLife Review.” Drone Life. July 30, 2015. https://dronelife.com/2015/07/30/the-3d-robotics-solo-a-dronelife-review.

Li, Chenchen and Hao Ling. 2016. “Radar Signatures of Small Consumer Drones.” Paper presented at the AP-S/USNC-URSI IEEE, Puerto Rico.

Northrop Grumman. 2016. “Tern UAS Concept Overview.” Video, 0:46. November 28, 2016. https://news.northropgrumman.com/news/releases/northrop-grumman-passes-key-development-milestones-on-darpaus-navy-tern-program.

Seng, Lee Kian, Mark Ovinis, T. Nagarajan, Ralph Seulin, and Olivier Morel. 2015. “Autonomous Patrol and Surveillance System using Unmanned Aerial Vehicles.” In *2015 IEEE 15th International Conference on Environment and Electrical Engineering (EEEIC)*, 1291–1297. Rome, Italy: IEEE.

Scott, Alwyn. 2017. “China Drone Maker Steps Up Security after U.S. Army Ban.” Reuters. Accessed October 12, 2017. https://www.reuters.com/article/us-usa-drones-dji/china-drone-maker-steps-up-security-after-u-s-army-ban-idUSKCN1AU294.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California