Charger Active Defense (ChAD) v1.0

Sickels, Noah  
*Electrical and Computer Engineering Dept.*  
*University of Alabama in Huntsville*Huntsville, AL 35899  
ncs0016@uah.edu  
  
Lochte, William  
*Electrical and Computer Engineering Dept.*  
*University of Alabama in Huntsville*Huntsville, AL 35899  
wpl0001@uah.edu  
  
Brannon, Adam  
*Electrical and Computer Engineering Dept.*  
*University of Alabama in Huntsville*Huntsville, AL 35899  
ajb0093@uah.edu  
  
Coe, David  
*Electrical and Computer Engineering Dept.*   
*University of Alabama in Huntsville*Huntsville, AL 35899  
david.coe@uah.edu

Jovanov, Emil  
*Electrical and Computer Enigneering Dept.*  
*University of Alabama in Huntsville*Huntsville, AL 35899  
emil.jovanov@uah.edu

Wells, B. Earl  
*Electrical and Computer Engineering Dept.*  
*University of Alabama in Huntsville*Huntsville, AL 35899  
wellsbe@uah.edu

*Abstract*— Modern attack tools are highly efficient and can send hundreds of thousands of requests per minute. The Charger Active Defense project attempts to solve this problem by crashing or hanging the attacker’s tools with invalid network packets. We provide an automated fuzzing workflow to test existing and AI-generated attack tools and a Python network service to replay found responses.

Keywords—active defense, cybersecurity, network-based fuzzing, artificial intelligence, attack tools, adversarial testing, ai-generated threats, program crash detection, prompt engineering

Contents

[1. Introduction 3](#_Toc197004304)

[1.1. Summary 3](#_Toc197004305)

[1.2. The Need 4](#_Toc197004306)

[1.3. Background Information 4](#_Toc197004307)

[2. Survey 4](#_Toc197004308)

[2.1. Research 4](#_Toc197004309)

[2.2. Patents 7](#_Toc197004310)

[2.3. Market Survey 9](#_Toc197004311)

[2.4. Projects 10](#_Toc197004312)

[3. Project Requirements 11](#_Toc197004313)

[3.1. Marketing Requirements 11](#_Toc197004314)

[3.2. Engineering Requirements 12](#_Toc197004315)

[4. Network Active Defense 13](#_Toc197004316)

[4.1. Big Picture 13](#_Toc197004317)

[4.2. Detailed Description 13](#_Toc197004318)

[4.3. Alternative Solutions & Trade-Offs 13](#_Toc197004319)

[5. Functional Decomposition 14](#_Toc197004320)

[6. Behavioral Decomposition 15](#_Toc197004321)

[7. Project Funcionality 15](#_Toc197004322)

[7.1. Testing 15](#_Toc197004323)

[7.1.1. Python Replay Service 15](#_Toc197004324)

[7.1.2. Fuzzing Workflow 18](#_Toc197004325)

[7.2. Project features 19](#_Toc197004326)

[8. Project management and timeline analysis 20](#_Toc197004327)

[8.1. Project Timeline 20](#_Toc197004328)

[8.2. Milestones 21](#_Toc197004329)

[8.3. Milestone Analysis 22](#_Toc197004330)

[8.4. Individual Contributions 23](#_Toc197004331)

[8.5. Level of Effort 23](#_Toc197004332)

[9. Future work 23](#_Toc197004333)

[10. Lessons learned 24](#_Toc197004334)

[11. Conclusions 24](#_Toc197004335)

[11.1. The problem and the solution 24](#_Toc197004336)

[11.2. Innovation 24](#_Toc197004337)

[11.3. Results 24](#_Toc197004338)

[12. Societal Impact Analysis 25](#_Toc197004339)

[12.1. Environmental Impact 25](#_Toc197004340)

[12.1.1. Energy Consuumption 25](#_Toc197004341)

[12.1.2. Electronic Waste (E-Waste) 25](#_Toc197004342)

[12.1.3. Raw Material Procurement 25](#_Toc197004343)

[12.2. Security Issues 25](#_Toc197004344)

[12.2.1. Vulnerabilities 25](#_Toc197004345)

[12.2.2. Other Associated Risks 26](#_Toc197004346)

[12.3. Interoperability Issues 26](#_Toc197004347)

[12.3.1. Legacy Systems 26](#_Toc197004348)

[12.4. Personal Privacy 26](#_Toc197004349)

[12.4.1. Data Handling 26](#_Toc197004350)

[12.5. Health and Safety 26](#_Toc197004351)

[12.5.1. Developmental Safety 26](#_Toc197004352)

[12.5.2. Product Safety 26](#_Toc197004353)

[12.6. Regulatory and Legal Compliance Issues 27](#_Toc197004354)

[12.6.1. Compliance 27](#_Toc197004355)

[12.7. Trade-Offs 27](#_Toc197004356)

[12.7.1. Environmental vs. Economic 27](#_Toc197004357)

[12.7.2. Health and Safety vs. Economic 27](#_Toc197004358)

[12.8. Security and Risk Analysis of Components and Systems 27](#_Toc197004359)

[12.9. Threat Model 28](#_Toc197004360)

[12.9.1. System Architecture 28](#_Toc197004361)

[12.9.2. Data Flow 28](#_Toc197004362)

[12.9.3. STRIDE Model 29](#_Toc197004363)

[12.9.4. Attack Tree 29](#_Toc197004364)

[12.9.5. Mitigations – Defensive Measures 30](#_Toc197004365)

[12.9.6. DREAD Analysis 31](#_Toc197004366)

[12.9.6.1. Test Cases 31](#_Toc197004367)

[13. Acknowledgment 32](#_Toc197004368)

[14. References 32](#_Toc197004369)

# Introduction

## Summary

Modern attack tools are highly efficient, while the cybersecurity industry struggles with active defense capabilities. Current passive defenses primarily focus on mitigating threats through means that can disrupt business operations or can be circumvented. Additionally, with the rapid advancement of artificial intelligence and prompt engineering, attackers can instantly generate various attack tools to target organizations.

The Charger Active Defense project focuses on a network-based fuzzing workflow that effectively and comprehensively tests these known and AI/LLM-generated attack tools. It aims to identify responses that may cause the attacking application to crash or hang. Responses generated are then saved and sent back to the adversary through our Python replay service during a detected attack. Due to the multi-threading nature of many attack tools like Masscan and Medusa, effectively fuzzing these tools is complicated. If a fuzzed response leads to a crash or hang, it typically occurs in a thread separate from the attacking application's instance. This crashed thread may either be reported as a false positive or not counted at all. Because of this discovery, at the sponsor's request, we pivoted towards thoroughly fuzzing generated attack tools from two large language models: GitHub Copilot and Phind.

Since we successfully applied the fuzzing workflow to Masscan, we also pursued the integration of ThreadSanitizer (TSan) to determine whether current fuzz testing properly crashes threads. The end goal of this workflow is to assess how we can broaden this method to apply to other attack tools, providing organizations with a strategy to defend their systems against an ever-evolving threat landscape.

## The Need

The cybersecurity industry currently lacks effective network-based active defense capabilities against incoming attacks. Most existing measures involve reactive defense, where the attacker is detected only after gaining initial entry or through methods that an attacker can circumvent through spoofing or other means.

This threat only continues to grow as new open-source attack tools are released. Easy-access OSs like Kali Linux come pre-packaged with over six hundred attack tools, ready to use right out of the box. The exponential growth of artificial intelligence has bolstered this threat, as any individual can generate attack tools effortlessly through prompt engineering. These threats present serious risks to organizations as they struggle against an ongoing battle.

## Background Information

"Active Defense" is a method of protecting a system by directly interacting with the adversary while an attack is in progress. "Attacking applications" refers to the programs that an adversary uses to attack your system. These programs often exploit vulnerabilities within a service, protocol, or application on the victim's system.

Fuzz testing is a software testing methodology that injects invalid input data into software to find crashes or hangs. A fuzzing tool typically does this by taking known, valid input from the tool under test and modifying it to cause unexpected behavior. There are several variations of how they do this, including dumb fuzzing, smart fuzzing, and mutation-based fuzzing. For this project, we utilize a smart and dumb mutation-based fuzzer, which generates input by changing the provided valid input to the application.

Please note that throughout this paper, we may refer to our developed Python program as a "Python active defense tool," "Python service," or "Python replay service" interchangeably. These terms refer to the program we created to demonstrate and implement active defense capabilities.

# Survey

## Research

1. J. Wang, Z. Zhang and M. Wang, "A Trust Management Method Against Abnormal Behavior of Industrial Control Networks Under Active Defense Architecture," in IEEE Transactions on Network and Service Management, vol. 19, no. 3, pp. 2549-2572, Sept. 2022, doi: 10.1109/TNSM.2022.3173398.
2. **Key Accomplishments**
3. Examined the trust management framework in industrial control networks, focusing on the security of control operations and clarifying fundamental concepts such as trust, security, and reliability in computer networks.
4. Analyzed unknown threats from Stuxnet, focusing on the operational behavior of instructions as the protected object, and extracted model trust information to identify abnormal behavior.
5. Provided a comprehensive deployment strategy for trust management, utilizing a trust server authority to facilitate network transactions.
6. **Main Findings**
7. Using a trusted server authority as a mediator helps protect network hosts before malicious traffic is detected.
8. The industrial control network is vulnerable to external attacks and internal malicious behaviors, leading to abnormal actions.
9. Abnormal behavior is characterized by unauthorized access to control instructions, non-compliant operation of these instructions, and interference with normal operations.
10. L. Fernandez and G. Karlsson, "Black-Box Fuzzing for Security in Managed Networks: An Outline," in IEEE Networking Letters, vol. 5, no. 4, pp. 241-244, Dec. 2023, doi: 10.1109/LNET.2023.3286443.
11. **Key Accomplishments**
12. Identify and illustrate how injection-type vulnerabilities can manifest in open service provider networks.
13. Provides a testing framework for identifying these vulnerabilities.
14. **Main Findings**
15. LLDP and SNMP do not correctly sanitize incoming LLDP data despite the LLDP data standard limiting the field to alphanumeric values.
16. The fuzzer host employs a grammar-based model to generate network traffic composed of malicious LLDP frames. A probe checks whether the generated frames are processed correctly. If the LLDP service improperly handles the frames, the fuzzer stores the relevant data from the test case into a database.
17. Many black-box fuzzing tools are implemented as plug-ins.
18. The probes communicate with the fuzzer using a well-defined interface, which allows customization of the language or model used.
19. Shackelford, Scott, "Rethinking Active Defense: A Comparative Analysis of Proactive Cybersecurity Policymaking," University of Pennsylvania Journal of International Law, vol. 41, no. 2, pp. 377, 2019-2020.
20. **Key Accomplishments**
21. Outline the different legal aspects of "hacking back" adversaries from a cybersecurity perspective.
22. This paper compares the efforts and outcomes of various countries regarding implementing sanctioned proactive (retaliatory) attacks on systems under threat.
23. This journal also highlights the effects of the U.S. Congress Graves bill passed in 2016.
24. **Main Findings**
25. The most significant impediment to proactive defense lies in its legal and ethical nature.
26. Legal precedents on regulating defense mechanisms such as honeypots and honeynets are unclear.
27. Many possible solutions violate the United States's Computer Fraud and Abuse Act (CFAA) of 1986.
28. M. Şenol, "Cyber Security and Defense: Proactive Defense and Deterrence," 2022 3rd International Informatics and Software Engineering Conference (IISEC), Ankara, Turkey, 2022, pp. 1-6, doi: 10.1109/IISEC56263.2022.9998314.
29. **Key Accomplishments**
30. Identifies current legal implementations for active and passive defense of networks.
31. The text examines the current landscape of "back hacking" and the counterattack strategies as part of active defense.
32. Highlights deficiencies and shortcomings in implementing proactive cyber defense measures and the lack of international cooperation.
33. **Main Findings**
34. The current "back hacking" concept for proactive defense is a legal grey area.
35. Current legal measures that organizations can implement include threat monitoring and response and deception techniques using honeypot systems.
36. Z. Cheng, Y. Guo, X. Li and J. Hu, "Based on Generative Adversarial Networks Seed Generation Method for Fuzzing," 2024 3rd International Conference on Big Data, Information and Computer Network (BDICN), Sanya, China, 2024, pp. 97-101, doi: 10.1109/BDICN62775.2024.00025.
37. **Key Accomplishments**
38. Employs base64 encoding and decoding technology to expand the types of generated seeds with fuzzing tools in a flexible way.
39. Uses the RelGAN model for seed generation and subsequently modifies the generated files with a hot-spot stitching algorithm.
40. Successfully improves the performance of fuzzing on target programs with various input formats.
41. **Main Findings**
42. GAN-based fuzzing of complex data with various formats improves the accuracy and quality of test input.
43. Employing a RelGAN model can allow you to generate seed files and address the imbalance between loss values and gradient updates in GAN's task of processing discrete data.
44. Combining the RelGAN model and its output with a hot-spot stitching algorithm enhances the quality of generated seeds, improving the efficiency of AFL++ in detecting crashes.
45. E. Yang, "Fuzz testing & software composition analysis in software engineering," 2018 International Symposium on VLSI Design, Automation and Test (VLSI-DAT), Hsinchu, Taiwan, 2018, pp. 1-3, doi: 10.1109/VLSI-DAT.2018.8373240.
46. **Key Accomplishments**
47. Introduces Software Composition Analysis (SCA) for fuzz testing applications where the source code is available.
48. Highlights the risks associated with using open-source software for organizational use.
49. Discusses compatibility of SCA with DevOps pipelines.
50. **Main Findings**
51. Open-source software results in more time and cost efficiency in application development, with higher code quality tested by a broader community.
52. Synopsys Defensics provides a testing platform for developers to discover unknown vulnerabilities proactively through mutation-based fuzz testing.

## Patents

1. Casas, Jose Carlos, “Continuous Active Defense for Digital Services”, United States Patent US20240022581A1, Jan. 18, 2024.
2. **Key Accomplishments**
3. Creating a secure client-server session method by implementing countermeasures based on predefined session-security challenge-response pairs.
4. Implement a collection of behavior patterns on the client's server to identify potential threats.
5. Implement session tracking and server response requests based on behavioral patterns and an agreed-upon client-server protocol.
6. Developed a server response system to verify the implementation of countermeasures executed by the client.
7. **Main Findings**
8. This method allows for dynamic application countermeasures based on real-time behavior analysis and predefined protocols.
9. This method relies on the client device to initiate countermeasures, providing a proactive defense response.
10. The patent lists various countermeasures, including screen capture blackening, challenge responses (text completion, mouse movement, image classification), and session logout.
11. This system includes databases for storing affected sessions, countermeasures, and rules for determining the application of countermeasures based on specific triggers.
12. Humphrey, Dickon Murray, “Cyber Threat Defense System and Method”, United States Patent US20210273960A1, Aug. 20, 2024
13. **Key Accomplishments**
14. A cyber threat defense system was developed using machine learning and artificial intelligence to analyze network data and detect threats.
15. Integration of multiple machine learning modules to evaluate overall network data, identify metrics, and determine the likelihood of a breach.
16. Implemented an autonomous response module that can transmit reports and initiate mitigation actions when a breach is detected.
17. The system is designed to train its artificial intelligence classifier continually during its operational life to improve its ability to identify threats.
18. **Main Findings**
19. Network data analysis can ingest and analyze network data associated with network structures and users to detect anomalies indicative of cyber threats.
20. Probability distributions for network data metrics provide a means to indicate the likelihood of an attack.
21. Machine learning models can store and generate historical network data and use it to train other models.
22. Meijer, Erik, “Fuzz Testing of Asynchronous Program Code”, United States Patent US20120089868A1, Apr. 20, 2015
23. **Key Accomplishments**
24. Developed a fuzz testing methodology for asynchronous event processing applications.
25. Introduces the concept of setting up event processors between event sources and event sinks to allow modification of the event stream based on received test information.
26. Starts the application with asynchronous behavior, managing event sources and sinks by receiving, transforming, and supplying the modified source events to the event sink through event processors.
27. It defines numerous modules used for the methodology, including components like event abstraction, event processor, source interface, sink interface, statistical distributions, and test storage.
28. **Main Findings**
29. A uniform interface connecting event processing to the sources allows abstracted event-related program code.
30. Event processors can sit between the source and sink, modifying or creating events to violate the contract and determine the application's behavior in response.
31. The statistical distribution component can store seeds used to generate particular test runs, allowing for the reproduction of identified software defects.

## Market Survey

1. BlackDuck, AppSec. “Defensics Fuzz Testing Tools & Services | Black Duck.” blackduck.com. www.blackduck.com/fuzz-testing.html. (accessed Feb 10. 2025)
2. **Features**

Comprehensive and flexible fuzzing tool that provides over 300 pre-built testing suites for various protocols, file formats, and interfaces. It also offers vulnerability mapping to industry standards like CWE and injection types.

1. **Price**

BlackDuck’s Defensics tool is currently quote-only; however, an estimation from a reference states that 30 on-premises servers and licensing cost $400k, or around ~$13k per server.

1. **Comparison**

Defensics offers pre-built test suites for many different protocols. However, due to time and resource constraints, the ChAD workflow is limited to three main protocols that must be implemented by hand. Defensics does not interact with the adversarial application directly.

1. Security, Beyond. “Dynamic Application Security Testing Tool (DAST) | BeSTORM.” beyondsecurity.com. www.beyondsecurity.com (accessed Feb. 9, 2025)
2. **Features**

A black-box dynamic application security testing and fuzzing suite designed to test millions of attack combinations. Able to perform test cases without access to the source code of the targeted application. Offers 250+ pre-built modules and protocols with the ability to implement custom protocols.

1. **Price**

beSTORM’s DAST tool requires a license, which costs $50,000 as a one-time purchase.

1. **Comparison**

beSTORM provides many different protocols that are standard for fuzz tests but do not directly interact with adversarial applications. beSTORM is not open-source and requires new licensing for use with other systems.

1. PortSwigger. “Burp Suite Professional.” portswigger.net. portswigger.net/burp/pro (accessed Feb. 9, 2025)
2. **Features**

A complete suite of web-based application testing tools capable of intercepting and manipulating network requests before repeating them back. Allows you to capture, filter, and query automated attack results.

1. **Price**

Burp Suite Professional requires a subscription license, which is $449 per year.

1. **Comparison**

Burp Suite primarily focuses on fuzz testing and input validation on HTTP/HTTPS web-based traffic, but it does not support many other protocols. Burp Suite also requires a proxy such as BurpProxy or FoxyProxy to intercept incoming traffic before relaying it to the target host.

## Projects

1. srg-imperial. “GitHub - Srg-Imperial/SnapFuzz.” github.com. github.com/srg-imperial/SnapFuzz. (accessed Feb. 9, 2025)
2. Offers a robust architecture that transforms slow asynchronous network communication into fast synchronous communications.
3. Snapshots the target at the latest point, speeds up all file operations by redirecting them to a custom in-memory filesystem, and removes the need for many fragile modifications, such as configuring time delays or writing clean-up scripts.
4. Integrates directly with AFLnet to improve overall performance and speed of protocol fuzzing.

1. honggfuzz, Google. “Honggfuzz.” github.com. github.com/google/honggfuzz. (accessed Feb. 9, 2025)
2. Security-oriented software fuzzer that supports feedback-driven fuzzing based on code coverage for software and hardware fuzz testing.
3. It is multi-process and multi-threaded, making it incredibly efficient. File corpus is automatically shared and improved between all fuzzed processes
4. It provides a corpus minimization mode, enabling it to optimize input data to improve fuzzing results.
5. PeterWeiJust. “GitHub – PeterWeiJust/SGANFuzz.” github.com. github.com/PeterWeiJust/SGANFuzz. (accessed Feb. 9, 2025)
6. Uses Generative Adversarial Networks (GANs) to generate realistic MQTT message sequences.
7. Supports multiple message types and parameters, allowing users to configure message sequence lengths and batch sizes.
8. It provides visualization tools to help users analyze generated message sequences.

# Project Requirements

## Marketing Requirements

1. Project Marketing Requirements

| ChAD Marketing Requirements | |
| --- | --- |
| Marketing Requirement | Description |
| M1 | The ChAD fuzzing workflow must conduct network-based fuzzing to identify network responses, also known as Active Defense Responses, that can crash or hang adversarial attack tools. |
| M2 | The existing vulnerabilities of six well-known attack tools must be documented. |
| M3 | Down select to a single well-known attack tool for testing. |
| M4 | Must use two different AI/LLM models to generate additional attack tools. |
| M5 | Use each AI/LLM to generate three attack tools for fuzz testing. |
| M6 | Software fuzzing tools capable of testing the attack tools must be identified. |
| M7 | Demonstrate a fuzz testing workflow for Masscan and AI-generated attack tools. |
| M8 | The ChAD program must monitor incoming network traffic to the host machine. |
| M9 | All transaction history of incoming and outgoing response packets must be logged and recorded. |
| M10 | The ChAD program must provide an active defense response. |
| M11 | Evaluate the effectiveness of any active defense responses found. |
| M12 | Findings must be documented in an IEEE/ACM-style paper. |

## Engineering Requirements

1. Project Engineering Requirements

| ChAD Engineering Requirements | |
| --- | --- |
| Marketing Requirement | Engineering Requirement |
| M3 | E1: LDRA static analysis and Valgrind memory leak analysis must be performed on both selected attack tools for present vulnerabilities for more favorable testing targets. |
| M6 | E2: Develop a fuzzing workflow using three fuzzing tools and approaches and rank them based on probability of success with selected well-known attack tools. |
| M4, M5 | E3: Must use GitHub Copilot and Phind models to generate three types of attack tools each – a banner-grabber, password brute-force, and a simplistic, multi-threaded banner-grabber with one additional thread. |
| M7 | E4: Must demonstrate the fuzzing workflow on selected and generated attack tools. |
| M1, M7, M10 | E5: The workflow must be capable of finding vulnerabilities (crashes or hangs) within attack tools, if any exist. |
| M11 | E6: Must document whether network responses crashes or hangs the attacking application and calculate and record the average hang time if it hangs the application. |
| M12 | E7: The rationale must be compiled for the selection of existing attack tools, fuzzing tools, compatibility results, and analysis for testing into an IEEE conference paper using proper formatting with font type, size, headers, and two columns per page. |
| M1, M2 | E8: Search MITRE CVE and Exploit-DB databases and compile known vulnerabilities for all six possible attack tools. |
| M8, M10 | E9: The ChAD service must send active defense responses within one minute of detection of an incoming attack. |

# Network Active Defense

## Big Picture

1. System Overview Diagram

## Detailed Description

Our project includes our fuzzing workflow, active defense Python service, and two virtual machines. Our fuzzing workflow uses AFLnet to comprehensively fuzz test network-based attack tools written in C or C++. Any test cases that cause the targeted application to crash or hand (e.g., active defense responses) are stored in a database for later use in our Python service.

Our two virtual machines consist of an attacking machine (Kali Linux) and a vulnerable victim virtual machine (Metasploitable2, Ubuntu 18.04). The victim virtual machine hosts vulnerable network applications that the attacker targets. The attacking virtual machine uses AFLnet to run the different attack tools against the victim on the port and protocol specified, simulating a real threat. In this case, AFLnet acts as a wrapper around the attack tool and fuzzes incoming network responses from the victim, looking for active defense responses.

Our Python application sits on the victim's virtual machine and actively monitors incoming network connections of their network adapter. Once the application detects the traffic from the network adapter, it retrieves and sends out the active defense response from the database to crash or hang the attack tool.

## Alternative Solutions & Trade-Offs

During our project, we examined various methods for implementing the fuzzing workflow, focusing primarily fuzz testing targets and methodology.

1. Modifying Existing Attack Tools

The initial attack tools we selected during the first semester were Masscan and Medusa. Between semesters, we selected Masscan to perform continuous fuzz testing until the start of the second semester. However, this testing proved unsuccessful, as none of the reported crashes or hangs worked to crash the application when reused. We believed this result was because of the extreme multi-threaded nature of the tools. Our theory was that the reported test cases were false positives, where the program could cause a crash or hang within one of the threads instead of the underlying application thread itself.

One possibility we discussed to remedy this problem was to refactor the attack tools to remove each tool's multi-threaded aspects. This option would allow us to perform the fuzz testing while maintaining the originally selected attack tools. We ultimately decided against this option, as this aspect was deeply rooted within these tools and would not be feasible within the project's time constraints.

1. Introducing Vulnerabilities into AI-Generated Attack Tools

Due to our decision against modifying existing tools, we shifted to generating attack tools from two different AI models at our sponsor's recommendation. To achieve better results, one option we explored was to manually introduce vulnerabilities within each generated attack tool to find valid crashes or hangs within them. This option would provide a more definitive result on whether the active defense methodology was feasible. We ultimately chose this method for one set of attack tools, as we wanted to find a positive active defense response.

However, doing so would give a less accurate representation of the active defense response capabilities within AI tools that an adversary generates, as it is unlikely that they would introduce vulnerabilities within the tools themselves. Ultimately, we decided to implement this option, but we did so incrementally and settled on removing the stack canary via the compiler flags used when building the tools.

1. Fuzzed PCAP Generation

For our fuzzing workflow, one option we looked at was fuzzed pcap generation over live protocol fuzzing. Our initial attack tools, Masscan and Medusa, used in the first semester, can perform their attacks using PCAP files as input data. While searching for various fuzzing tools, we discovered tools like Wireshark's FuzzShark or Scapy that could generate PCAP files with fuzzed network packets. This process could be scripted to provide additional test cases for us to use during fuzz testing, and it did not require the same amount of resources as live protocol fuzzing.

While this option would simplify the fuzz testing, we decided against it, as live protocol fuzzing provided a more accurate representation of the packets on the network and how the attack tools responded.

# Functional Decomposition

1. Functional Decomposition

# Behavioral Decomposition

1. Behavioral Decomposition

# Project Funcionality

## Testing

We broke our project testing into three main areas: the Python replay service, the fuzzing workflow, and the AI-generated attack tools.

## Python Replay Service

We tested each component of the Python replay service, including the user interface, network handling, error handling, and the logger.

Testing on the user interface was straightforward, as it was the primary component used to develop the program and gave us visual confirmation, as shown in Fig 4. below.

1. Python Replay Service - User Interface

Network handling involved ensuring that we could receive incoming network traffic, display it on the screen, and send out crafted packets across the network. We validated that we could receive incoming network traffic through visual confirmation as we printed out the incoming packets to the program's terminal, allowing us to see that the tool correctly ingests packets, as shown in Fig. 5 below.

1. Network Traffic Displayed from Python Service

To verify that we could send traffic, we used the same visual confirmation of sent packets—checking the port, protocol, and source and destination IP addresses of packets printed to the terminal—and monitored network traffic on the adapter through Wireshark, as shown in Fig. 6.

1. Network Traffic Displayed from Python Service

The logging module was easy to verify, as we utilize it throughout our program. However, to simplify it, we wrote a unit test command that printed out test messages from each logging type (information, warning, error, sent traffic, received traffic, and active defense responses), as shown in Fig. 7.

1. Logger Module Unit Test

Additionally, we verified all transaction and command history for the Python active defense tool through the generated log files containing all traffic history once the program starts, as shown in Fig. 8.

1. Network Traffic History Log File

## Fuzzing Workflow

Validating the fuzzing workflow involved ensuring that all generated attack tools would successfully compile with AFLnet and that Masscan would correctly compile with the added ThreadSanitizer hook. We manually compiled all AI attack tools, ensuring we resolved any errors preventing them from generating the binary file. An example of this is shown in Fig. 9 below.

1. Example Compilation and Fuzzing of AI Generated Attack Tool

We validated Masscan with ThreadSanitizer by looking at the compilation flags used in the output generated from afl-gcc, as highlighted in Fig. 10 below.

1. Masscan with ThreadSanitizer Hook

## Project features

We developed a fuzzing workflow capable of comprehensively fuzz-testing network-based attack tools written in C/C++. This workflow includes an automated script that installs and builds all necessary dependencies, including our fuzzing tools, known attack tools, and AI-generated attack tools. With this workflow, we performed fuzz testing on Masscan and six total AI-generated attack tools and found an exploitable active defense response within two of them.

We successfully generated three different attack tools from the GitHub Copilot and Phind AI models, including a banner grabber, a password brute forcer, and a multi-threaded banner grabber, allowing us to assess AI's feasibility for generating attack tools.

For each AI attack tool, we performed a series of tests with several tools, including LDRA Testbed for code structure and quality reports, Valgrind for memory leak analysis, and Flawfinder for relevant CVEs and CWEs within the program's source code.

Our Python replay service is fully functional and capable of sending and receiving network traffic and responding to the incoming traffic from the attack tool by providing the active defense response to crash the application. The user interface provides a suite of features, including custom commands, command auto-complete, and a status bar displaying the current testing configuration. Our custom logger provides colorized logging messages, informing the user of occurring events. Additionally, the program stores the transaction history of all packets across the network, errors that occur, and command history into separate log files for later analysis.

# Project management and timeline analysis

## Project Timeline

1. Proposed Timeline
2. Actual Timeline

## Milestones

1. Proposed Milestones

| Proposed Milestones | | |
| --- | --- | --- |
| Num. # | Milestone | End Date |
| 1 | Masscan ThreadSanitizer Integration for AFLnet Fuzzing | 1/31 |
| 2 | Python Service User Interface Complete | 2/3 |
| 3 | Python Relay Error and Traffic Logging Module | 2/7 |
| 4 | Gen-AI Banner-Grabbing Attack Tool | 2/14 |
| 5 | Python Relay Network Handling Module | 2/14 |
| 6 | LDRA Static Analysis on Gen-AI Banner-Grabber Tools | 2/21 |
| 7 | Gen-AI Brute-Force Attack Tool | 2/28 |
| 8 | Comparison of Results of Analysis & Fuzzing on Gen-AI Banner-Grabber Attack Tools Complete | 2/28 |
| 9 | Gen-AI Multi-Threaded Banner-Grabbing Attack Tool | 3/7 |
| 10 | Compare Gen-AI Brute-Force Attack Tool Performance Between Models | 3/14 |
| 11 | Static Analysis Performed on Gen-AI Multi-Threaded Banner-Grabber Attack Tools | 3/21 |
| 12 | Comparison of Results of Analysis & Fuzzing on Gen-AI Brute Force Attack Tools Complete | 3/21 |
| 13 | Python Service Network Receiver Module Developed | 3/28 |
| 14 | Python Service Network Sender Module Developed | 3/28 |
| 15 | Gen-AI Attack Tool Comparison Report Finished | 3/28 |
| 16 | Unit Testing Complete | 4/4 |
| 17 | Review of Masscan ThreadSanitizer Results | 4/4 |
| 18 | Integration Testing Complete | 4/11 |
| 19 | Acceptance Testing Complete | 4/18 |

1. Actual Milestones

| Actual Milestones | | | |
| --- | --- | --- | --- |
| Num. # | Milestone | Projected End Date | Actual End Date |
| 1 | Masscan with ThreadSanitizer Integrated into Fuzzing Workflow | 1/31 | 1/31 |
| 2 | Python Service User Interface Complete | 2/3 | 2/3 |
| 3 | Python Service Error & Traffic Logging Module Complete | 2/7 | 2/7 |
| 4 | AI Banner-Grabber Attack Tools Generated | 2/14 | 2/14 |
| 5 | Python Service Response Handling Module Complete | 2/14 | 2/14 |
| 6 | Fuzzing Workflow Applied to AI Banner-Grabber Attack Tools | 2/21 | 2/21 |
| 7 | Static Analysis Performed on AI Banner-Grabber Attack Tools | 2/21 | 2/21 |
| 8 | AI Brute Force Attack Tools Generated | 2/28 | 2/24 |
| 9 | Comparison of Results of Analysis & Fuzzing on AI Banner-Grabber Attack Tools Complete | 2/28 | 2/28 |
| 10 | Fuzzing Workflow Applied to AI Brute Force Attack Tools | 3/7 | 3/7 |
| 11 | Static Analysis Performed on AI Brute Force Attack Tools | 3/7 | 3/2 |
| 12 | AI Multi-threaded Banner-Grabber Attack Tools Generated | 3/14 | 3/10 |
| 13 | Comparison of Results of Analysis & Fuzzing on AI Multi-Threaded Banner-Grabber Attack Tools Complete | 3/14 | 3/14 |

1. Actual Milestones Continued

| Actual Milestones | | | |
| --- | --- | --- | --- |
| Num. # | Milestone | Projected End Date | Actual End Date |
| 14 | Fuzzing Workflow Applied to AI Multi-Threaded Banner-Grabber Attack Tools | 3/21 | 3/21 |
| 15 | Static Analysis Performed on AI Multi-Threaded Banner-Grabber Attack Tools | 3/21 | 3/15 |
| 16 | Comparison of Results of Analysis & Fuzzing on AI Brute Force Attack Tools Complete | 3/21 | 3/21 |
| 17 | Python Service Network Receiver Module Developed | 3/28 | 3/4 |
| 18 | Python Service Network Sender Module Developed | 3/28 | 3/1 |
| 19 | AI Attack Tool Comparison Report Finished | 4/4 | 4/4 |
| 20 | Unit Testing Complete | 4/4 | 4/4 |
| 21 | Masscan ThreadSanitizer Fuzzing Results Analyzed | 4/4 | 4/11 |
| 22 | Integration Testing Complete | 4/11 | 4/11 |
| 23 | Acceptance Testing Complete | 4/18 | 4/17 |
| 24 | IEEE Conference Paper Written | 4/25 | - |
| 25 | Documentation & GitHub Repository Updated | 5/2 | 5/2 |
| 26 | GitHub Repository & Research Published | 5/2 | 5/2 |

## Milestone Analysis

The only milestone we missed was “Masscan ThreadSanitizer Fuzzing Results Analyzed”, which we projected to complete on April 4th but did not complete until April 11th. The missed deadline occurred because the Proxmox server running the fuzz testing of Masscan crashed due to an unrelated kernel error in which we re-imaged it from a backup. Missing this milestone did not set us back; we scheduled the unit testing for the fuzzing workflow with Masscan later that week.

1. Early Milestones

| Early Milestones | | | |
| --- | --- | --- | --- |
| Num. # | Milestone | Projected End Date | Actual End Date |
| 8 | AI Brute Force Attack Tools Generated | 2/28 | 2/24 |
| 11 | Static Analysis Performed on AI Brute Force Attack Tools | 3/7 | 3/2 |
| 12 | AI Multi-Threaded Banner-Grabber Attack Tools Generated | 3/14 | 3/10 |
| 15 | Static Analysis Performed on AI Multi-Threaded Banner-Grabber Attack Tools | 3/21 | 3/15 |
| 17 | Python Service Network Receiver Module Developed | 3/28 | 3/4 |
| 18 | Python Service Network Sender Module Developed | 3/28 | 3/1 |

We completed several milestones earlier than expected, primarily because we anticipated more complications from generating the specific attack tools that would compile and perform as expected. We also had other milestones related to the Python service earlier due to an oversight in our planning, as we scheduled milestones that required the network functionality earlier than the actual network sending and receiving module milestones.

## Individual Contributions

1. Individual Contributions Overview

| Individual Contributions | |
| --- | --- |
| Member | Description of Tasks |
| Noah | Maintaining the GitHub repository – organization, structure, general upkeep.  Python replay service development – UI, logging, and network handling.   Application of fuzzing workflow on AI generated tools. |
| Adam | Generating the three attack tools from the Phind model.  Performing Valgrind and Flawfinder analysis on all generated attack tools. |
| William | Generating the three attack tools from the GitHub Copilot model.  Performing LDRA static analysis on all generated attack tools. |

## Level of Effort

1. Work Hours Breakdown

| Work Hours Breakdown | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Member | Brief 1 | Brief 2 | Brief 3 | Brief 4 | Brief 5 | Brief 6 | Total |
| Noah | 15 | 28 | 23 | 30 | 64 | 7 | 167 |
| Adam | 5 | 18 | 8 | 8 | 24 | 2 | 65 |
| William | 7 | 20 | 5 | 3 | 20 | 3 | 58 |
| Total | 29 | 66 | 36 | 41 | 108 | 12 | 290 |

# Future work

If we were given more time to work on the project, our goals for continuing would focus on the extensibility of the fuzzing workflow and our targeted attack tools.

1. Extend the fuzzing workflow to include other network-based fuzzing and attack tools.
   1. While our first semester’s efforts focused on testing several fuzzing tools to use for the fuzzing workflow, there are many still available that we did not have time to assess.
   2. The time and effort required would depend on the scope of additional tools, but could vary from a few weeks or an entirely new project.
2. Explore other AI models to use for generating attack tools.
   1. Other models could include open-source alternatives that are more freely available to consumers. The models used in our case were licensed or free-trial access.
   2. The time and effort required would likely be similar to the second semester – two AI models for three different attack tools.
3. Implement attack tool traffic identification to the Python replay service.
   1. Attack tool traffic identification was originally included in the scope of our initial project; however, we removed it at the sponsor’s recommendation, as they deemed it unfeasible given our remaining time available.
   2. The time and effort required would likely warrant an entire design project – two full semesters.

# Lessons learned

After completing this project, we have learned a great deal.

1. Fuzzing networks is far more difficult than we initially believed, as the dynamic nature of networking complicates our ability to create stable test environments without outside interference.
2. Documentation is not always accurate or up to date.
3. Just because an established brand or company owns it does not make it a perfect product.
4. AI models are surprisingly good at generating attack tools that do not require much, if any, modification. Generating the different attack tools was far easier than we anticipated.
5. Time and project management are challenging, and it was not always easy to stay on track with our proposed timeline and plan.
6. Communication between group members is paramount and must be consistent and frequent to minimize wasted efforts or confusion.

# Conclusions

## The problem and the solution

The cybersecurity industry needs more solutions to the ongoing threat to its systems and organizations. Over the years, the skill gap required to attack other systems has diminished as new, easy-to-use tools have become available. The Charger Active Defense project provides a methodology and tool to actively stop attackers before they can cause damage. The fuzzing workflow is an automated process to find crashes or hangs within these attacking applications, allowing them to be used against them through our Python replay service.

## Innovation

Traditional methods in the cybersecurity industry focus on reactive defense, where organizations put mitigations in place to prevent future attacks. These methods are only effective once the attacker is detected and already in the network or targeting their systems. Our solution provides an active defense method that stops an attacker before they can damage a system by targeting the specific tool they are using.

## Results

We developed an automated fuzzing workflow script and methodology that performs extensive network-based fuzz testing on attack tools written in C/C++. We generated three different attack tools from the GitHub Copilot and Phind AI models and applied our fuzzing workflow to assess AI's capability for creating programs based on the desired functionality. From the fuzzing workflow testing, we successfully found active defense responses (crashes or hangs) within two of the AI-generated attack tools.

Our Python replay service is rich with features, providing a sleek user interface, custom colorized logging messages, network traffic capture and transmission, and the ability to send out discovered active defense responses to the targeted applications.

Based on our proposed solution, we slightly adjusted the project's scope and requirements. We originally included identifying the attack tool that is actively used within our Python replay service, but we removed it due to feasibility at the sponsor's recommendation. Our Python replay service now assumes that all incoming traffic on the specific network adapter from the service and port number is the attack tool, as the environment we used for testing is isolated enough not to cause interference.

Additionally, we originally intended to use Masscan as a primary attack tool for further testing but expanded this scope to include AI-generated tools of various types due to Masscan's multi-threaded nature, which made testing difficult.

We hope our project will help further research and development into network active defense capabilities through the cybersecurity industry.

# Societal Impact Analysis

## Environmental Impact

## Energy Consuumption

The fuzzing workflow demands substantial computational resources to operate the fuzzing tools and analyze network responses. The process involves several virtual machines running concurrently for several weeks at a time. This higher energy consumption contributes to greenhouse gas emissions, as much of our energy production relies on fossil fuels. To address this issue, we can investigate using energy-efficient hardware and optimize software to minimize computational overhead.

Training and running the AI models also require extensive resources and energy from each prompt. Since both models are cloud-based and hosted by the respective company, this may increase energy usage.

## Electronic Waste (E-Waste)

Frequent upgrades and hardware replacements required to maintain security updates can increase electronic waste. These replacements include the server space used for fuzz testing and the servers and edge devices used for hosting the AI models. Proper disposal methods, including recycling programs and e-waste management, should be implemented to minimize environmental impact. Additionally, extending the lifespan of hardware through regular maintenance and updates can help reduce the need for replacements.

## Raw Material Procurement

The raw materials for the hardware used in the project are typically procured through mining and manufacturing processes that can adversely affect wildlife, forests, and water quality. Sustainable sourcing practices and using recycled materials can help mitigate these impacts.

## Security Issues

## Vulnerabilities

If not properly secured, the fuzzing workflow itself could introduce vulnerabilities. Ensuring that the fuzzing tools and the active defense service application are regularly updated and patched is crucial. Implementing secure coding practices and regular security audits can help identify and address potential vulnerabilities.

The Metasploitable2 virtual machine is designed to be an insecure target for fuzz testing. It features multiple vulnerabilities due to outdated protocol versions and insecure configurations. As such, it will not be connected to the Internet and is strictly limited to the internal network between the two virtual machines.

Each AI-generated attack tool's source code inherently contains vulnerabilities, including the absence of stack canaries and the use of insecure functions. Careful management of the program during testing is essential to prevent any harm to the system.

## Other Associated Risks

The attacking virtual machine for this project is Kali Linux 2024.1. This OS is pre-built with hundreds of attack tools designed for penetration testing or adversarial emulation. If used maliciously, it presents an immediate threat to any corporate or institutional network. We actively monitor the virtual machine's interaction and traffic and have it properly segmented from critical network traffic to ensure the safety of the network in use.

## Interoperability Issues

## Legacy Systems

The fuzzing workflow may need to interact with legacy systems that use outdated protocols and insecure security measures, such as outdated PostgreSQL or SSH versions. One of the attacking virtual machines is an Ubuntu 18.04 OS, which no longer receives long-term support (LTS) updates from Canonical. This system is actively segmented on the network to avoid potential issues.

## Personal Privacy

## Data Handling

The project involves analyzing network traffic that may contain sensitive information. We conduct fuzz testing on a private, configured network equipped with a firewall specifically set up for this purpose. It is crucial to ensure that personal data is anonymized and handled securely to safeguard privacy. Strict access controls and encryption can help protect personal data without compromising services.

This project's proactive defense aspect (e.g., "back hacking") involves targeting another system from the perspective of that attacking system being foreign to the defenders. In our use case, both machines are owned and operated by us or a trusted institution. In a real-time situation where we are actively defending our system against a foreign adversary, trying to hang or crash the attacker's application would directly violate the integrity of the attacker's personal computer and the data it holds.

## Health and Safety

## Developmental Safety

Development and testing of the fuzzing workflow involve dealing with potentially harmful network traffic. It is essential to ensure that the development environment is secure and that developers are trained in safe handling practices to prevent accidental exposure to malicious data.

## Product Safety

Additionally, we configured and designed the active defense fuzzing workflow and our testbed to avoid unintended disruptions to legitimate network traffic, which could impact the availability of critical services. Thorough testing and validation are necessary to ensure the product's safety and reliability.

## Regulatory and Legal Compliance Issues

## Compliance

The project must comply with various regulations and standards related to cybersecurity, data protection, and electronic waste management. In the United States, this includes regulations such as the General Data Protection Regulation (GDPR) for data privacy and the Resource Conservation and Recovery Act (RCRA) for e-waste.

This project's proactive defense aspect (e.g., "back hacking") involves targeting another system from the perspective of that attacking system being foreign to the defenders. In our use case, both machines are owned and operated by us or a trusted institution. In a real-time situation where we are actively defending our system against a foreign adversary trying to hang or crash the attacker's application, this would directly violate the Computer Fraud and Abuse Act (CFAA) of 1986.

## Trade-Offs

## Environmental vs. Economic

Balancing environmental impact with economic considerations presents a significant trade-off. While investing in energy-efficient hardware and sustainable materials may increase initial costs, the long-term benefits, such as reduced environmental harm and compliance with regulations, can outweigh the initial cost. However, sourcing environmentally friendly hardware that meets the performance demands of virtual machines and fuzz testing applications could be problematic. Additionally, increased regulatory oversight may lead to even higher costs for that hardware.

## Health and Safety vs. Economic

Ensuring the health and safety of the group members may require more time and resources, as configuring virtual machines and applications does not compromise the integrity of the network or the machines used for testing.

## Security and Risk Analysis of Components and Systems

We configured the fuzzing workflow and necessary software (fuzz testing tools, e.g., AFLnet, Fuzzowski, Radamsa, etc.) on the virtual machines (Kali Linux 2024.1, Ubuntu 18.04, Metasploitable2) to not disrupt regular network traffic while in use to maintain the operations of adjacent systems. Implementing robust security measures (two-factor authentication, complex password policy, etc.), regular updates on relevant hardware, and continuous monitoring to detect and respond to potential attacks is paramount. We also take regular backups or snapshots of the virtual machines to preserve their integrity.

## Threat Model

1. Threat Model Diagram

## System Architecture

1. Kali Linux 2024.4 Virtual Machine (Attacker)
   1. Runs AI-generated attack tools.
   2. Runs Masscan attack tool.
   3. Runs AFLnet wrapper on each attack tool for network-based protocol fuzzing.
2. Metasploitable2 Virtual Machine/Ubuntu 18.04 (Victim)
   1. Hosts vulnerable services.
   2. Runs Python service to monitor network traffic and replay active defense responses.
3. Network Interface
   1. Facilitates the communication between the attack and the victim.
   2. Configured as a virtual network (vEthernet, Host-Only or Internal) adapter through VirtualBox.
4. Active Defense Response Database
   1. Storage of all found crashes or hangs within attacking applications.
   2. Stored as text entries in respective directory of the Victim machine.

## Data Flow

1. **Attack Initiation:** The Kali virtual machine sends traffic to the Metasploitable2/Ubuntu 18.04 (victim) virtual machine.
2. **Response Handling:** The ChAD service on the victim virtual machine sends back active defense responses to the attacking virtual machine.
3. **AFLnet Fuzzing:** AFLnet modifies the attack patterns to detect crashes or hangs within the attacking application.
4. **Replay Mechanism:** The ChAD service replays the active defense responses to the attack tools once it detects network traffic from the specific network interface.
5. **Analysis:** Logs for incoming and outgoing network traffic, sent active defense responses, and found crashes or hangs from AFLnet are collected and stored.

## STRIDE Model

1. Microsoft STRIDE Threat Model

| Threat Analysis | | |
| --- | --- | --- |
| Threat | Description | Mitigation |
| Spoofing | An attacker may impersonate the Kali or Metasploitable2 virtual machines to inject false data. | Mutual authentication, network isolation, cryptographic signing of attack payloads. |
| Tampering | Ad adversary may launch malicious code into the network interface and execute it or alter the attack/replay payloads. | Integrity verification, logging and monitoring payload transformations. |
| Repudiation | An adversary could deny actions on the database due to a lack of auditing. | Comprehensive logging and timestamping of events. |
| Information Disclosure | An adversary could gain access to sensitive PII or HBI data in the database, from the datasets used for the attack tools, or from active defense responses (passwords, usernames, hostname, IP addresses). | Restrict logging of sensitive data in the controlled environment. |
| Denial of Service (DoS) | Overloading the replay service or attack tools with excessive traffic could cause failure and downtime. | Rate limiting, monitoring resource usage, segmentation of test cases. |
| Elevation of Privilege | Exploiting vulnerabilities with the attack tools to escalate privileges. | Running tests in sandboxed environments with least privilege. |

## Attack Tree

**Goal:** Compromise the Integrity or Availability of Attack Tools.

1. Exploit AFLnet to Cause Unintended Behavior
   1. Inject malformed responses to crash the fuzzing workflow.
   2. Use crafted responses to trigger crashes or hangs.
2. Overload Attack Tools via Replay Mechanism
   1. Send active defense response payloads to cause a crash or hang within the attacking application.
   2. Replay malformed responses to trigger tool exceptions.
3. Interfere with Metasploitable2’s Response Handling
   1. Alter attack payloads mid-transit.
   2. Poison replay service logs with false data.
4. Trigger Unhandled Exceptions in AI-Generated Attack Tools
   1. Identify input validation gaps.
   2. Inject adversarial attack patterns.

## Mitigations – Defensive Measures

1. Input Validation
2. Rate Limiting
3. Logging and Monitoring: Log anomalous behaviors and unexpected failures.
4. Sandboxing: Isolate attacker and victim virtual machines on a separate network, disconnected from the internet or other critical assets.
5. Integrity Checks: Ensure active defense responses maintain integrity while at rest and in-transit.

## DREAD Analysis

1. DREAD Model Analysis

| DREAD Analysis Scores | | | | | | |
| --- | --- | --- | --- | --- | --- | --- |
| Threat Scenario | Damage | Reproducibility | Exploitability | Affected Users | Discoverability | Total Risk |
| Crash attack tools via malformed responses. | 7 | 9 | 8 | 6 | 8 | 38 (High) |
| Unhandled exception in attack tools. | 6 | 8 | 7 | 5 | 8 | 34 (Medium) |
| Denial of service on the Python replay tool. | 8 | 9 | 6 | 5 | 7 | 35 (High) |
| Information leakage from attack tools. | 7 | 7 | 6 | 4 | 6 | 30 (Medium) |

## Test Cases

1. Crash Testing AFLnet and Attack Tools
   1. Send specially crafted responses with invalid protocol or data formats.
   2. Replay responses at high speed to cause memory exhaustion.
   3. Introduce unexpected data types (e.g., Unicode, null bytes, overflows).
2. Monitoring Replay Service Stability
   1. Test with varying response rates to assess service resilience.
   2. Log replay accuracy and ensure payloads are delivered correctly.
3. Analyzing Attack Tool Responses
   1. Check if tools handle malformed responses gracefully.
   2. Detect if certain payloads consistently crash the attack tool.
4. Security Hardening
   1. Validate authentication mechanisms to prevent unauthorized modifications.
   2. Ensure data integrity when passing responses between systems.

# Acknowledgment

On behalf of Noah Sickels, William Lochte, and Adam Brannon, we want to thank Dr. David Coe, our sponsor and mentor for this project, for his guidance and support throughout the semesters.

We would also like to thank Dr. Emil Jovanov and Dr. Earl Wells from the UAH ECE department for their guidance and oversight of project management and deliverables.

# References

1. J. Wang, Z. Zhang and M. Wang, "A Trust Management Method Against Abnormal Behavior of Industrial Control Networks Under Active Defense Architecture," in IEEE Transactions on Network and Service Management, vol. 19, no. 3, pp. 2549-2572, Sept. 2022, doi: 10.1109/TNSM.2022.3173398.
2. L. Fernandez and G. Karlsson, "Black-Box Fuzzing for Security in Managed Networks: An Outline," in IEEE Networking Letters, vol. 5, no. 4, pp. 241-244, Dec. 2023, doi: 10.1109/LNET.2023.3286443.
3. Shackelford, Scott, "Rethinking Active Defense: A Comparative Analysis of Proactive Cybersecurity Policymaking," University of Pennsylvania Journal of International Law, vol. 41, no. 2, pp. 377, 2019-2020.
4. M. Şenol, "Cyber Security and Defense: Proactive Defense and Deterrence," 2022 3rd International Informatics and Software Engineering Conference (IISEC), Ankara, Turkey, 2022, pp. 1-6, doi: 10.1109/IISEC56263.2022.9998314.
5. Z. Cheng, Y. Guo, X. Li and J. Hu, "Based on Generative Adversarial Networks Seed Generation Method for Fuzzing," 2024 3rd International Conference on Big Data, Information and Computer Network (BDICN), Sanya, China, 2024, pp. 97-101, doi: 10.1109/BDICN62775.2024.00025.
6. E. Yang, "Fuzz testing & software composition analysis in software engineering," 2018 International Symposium on VLSI Design, Automation and Test (VLSI-DAT), Hsinchu, Taiwan, 2018, pp. 1-3, doi: 10.1109/VLSI-DAT.2018.8373240.
7. Casas, Jose Carlos, “Continuous Active Defense for Digital Services”, United States Patent US20240022581A1, Jan. 18, 2024.
8. Humphrey, Dickon Murray, “Cyber Threat Defense System and Method”, United States Patent US20210273960A1, Aug. 20, 2024
9. Meijer, Erik, “Fuzz Testing of Asynchronous Program Code”, United States Patent US20120089868A1, Apr. 20, 2015
10. BlackDuck, AppSec. “Defensics Fuzz Testing Tools & Services | Black Duck.” blackduck.com. www.blackduck.com/fuzz-testing.html. (accessed Feb 10. 2025)
11. Security, Beyond. “Dynamic Application Security Testing Tool (DAST) | BeSTORM.” beyondsecurity.com. www.beyondsecurity.com (accessed Feb. 9, 2025)
12. PortSwigger. “Burp Suite Professional.” portswigger.net. portswigger.net/burp/pro (accessed Feb. 9, 2025)
13. srg-imperial. “GitHub - Srg-Imperial/SnapFuzz.” github.com. github.com/srg-imperial/SnapFuzz. (accessed Feb. 9, 2025)
14. honggfuzz, Google. “Honggfuzz.” github.com. github.com/google/honggfuzz. (accessed Feb. 9, 2025)
15. PeterWeiJust. “GitHub – PeterWeiJust/SGANFuzz.” github.com. github.com/PeterWeiJust/SGANFuzz. (accessed Feb. 9, 2025)