REINFORCEMENT LEARNING-BASED SECURITY VULNERABILITY DETECTION FOR MICROSERVICES

Kasun Maduranga

(219361N)

Degree of Master of Science

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

July 2025

REINFORCEMENT LEARNING-BASED SECURITY VULNERABILITY DETECTION FOR MICROSERVICES

Kasun Maduranga

(219361N)

MSc thesis submitted in partial fulfilment of the requirement for the degree Master of Science.

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

July 2025

DECLARATION

I declare that this is my own work, and this MSc Research Project Report does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief, it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to the University of Moratuwa the non-exclusive right to reproduce and distribute my thesis/dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works.

……. 02/07/2025

Kasun Maduranga Date

The above candidate has carried out research for the Masters' thesis under my supervision.

Name of the supervisor: Prof. Indika Perera

……. 02/07/2025

Prof. Indika Perera Date

ACKNOWLEDGEMENT

I like to express my sincere gratitude to my thesis supervisor, Professor Indika Perera, for his unwavering support and essential direction during my study. His expertise in the field and encouragement provided was vital to the successful finalization of this work. I would like to mention the remarkable flexibility that Professor Indika Perera and the Computer Science and Engineering department provided for the completion of this research.

I extend my heartfelt thanks to the Google Cloud Platform developers for creating the Online Boutique microservice application. I appreciate the contributions they have made to the industry and academic community by releasing this application as an open-source project and maintaining it actively. It made the implementation of the proposed methodology feasible and convenient.

Lastly, I like to convey my appreciation towards my parents and my wife for their patience and continuous support. They always kept me motivated to complete this research successfully.

ABSTRACT

Microservices architecture (MSA) is the main architectural model for contemporary software systems due to its scalability, flexibility, and maintainability. Yet, there are major security concerns that emerge from the distributed and dynamic nature of microservices. Most of these risks and vulnerabilities are difficult to identify due to the complexity and the evolving nature of microservice based systems. Due to this reason, conventional security vulnerability detection methods designed for monolithic systems are inadequate and ineffective in microservice based systems. To address this shortcoming, this thesis investigates the use of Reinforcement Learning (RL) and offers a Proximal Policy Optimization (PPO)-based framework as an automated and adaptive tool for finding security vulnerabilities in microservices. “Online Boutique” application, which built on microservices architecture is utilized as a testbed for assessing the performance of the developed RL framework in finding security vulnerabilities. The PPO agent learns to interact with the system, simulate attacks, and find real-time security vulnerabilities. The study intends to mimic DoS attack situations targeting the Online Boutique application. The suggested approach is a scalable, consistent security testing tool able to evolve to identify developing threats and adapt to new security vulnerabilities. This study proposes a promising substitute for conventional manual testing for DoS attacks. The results illustrate the efficacy of the PPO framework in detecting vulnerabilities in the microservices environment, with implications for improving the security of microservices-based applications.

Keywords: Microservices Architecture, Reinforcement Learning, Proximal Policy Optimization, Vulnerability Detection, Security Testing, Automated Security, Penetration Testing, DoS Simulation, Machine Learning in Cybersecurity.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

MSA	Microservice Architecture
RL	Reinforcement Learning
PPO	Proximal Policy Optimization
DoS	Denial of Service
SQL	Structured Query Language
URL	Uniform Resource Locator
CPU	Central Processing Unit
API	Application Programming Interface
TLS	Transport Layer Security
DQN	Deep Q-Networks
IDS	Intrusion Detection Systems
TRPO	Trust Region Policy Optimization
CI	Continuous Integration
CD	Continuous Deployment
XSS	Cross-Site Scripting
A3C	Asynchronous Advantage Actor-Critic
RAM	Random Access Memory
GB	Giga Byte
MB	Mega Byte
KB	Kilo Byte
BIOS	Basic Input/Output System
UEFI	Unified Extensible Firmware Interface
LTS	Long Term Support
WSL	Windows Subsystem for Linux
CLI	Command Line Interface

IP	Internet Protocol
HTTP	Hyper Text Transfer Protocol
ID	Identifier
JSON	JavaScript Object Notation
QPS	Queries Per Second
TCP	Transmission Control Protocol
GAE	Generalized Advantage Estimator
UI	User Interface

1. INTRODUCTION

1.1. Background

Microservices Architecture Overview and Its Growing Use in Modern Software Systems

Microservices architecture (MSA) refers to the development of an application of a system as a composite of loosely jointed, autonomously deployable services. In every microservices, each business unit is assigned a specific service which they can develop, deploy, and scale independently. This is different than traditional monolithic architecture where all the components function as one seamlessly integrated system—MSA is quite the opposite of that [1].

Due to its growing popularity, modern software systems can take advantage of scalability, flexibility, and maintainability, including large-scale applications that necessitate high uptime, frequent updates, and the ability to scale [2]. MSA's advantage is allowing organizations to deploy components of a system individually which leads to faster development cycles with more robust and scalable solutions.

Since microservices are composed of many independent services, they provide the developers with the flexibility of choosing different technologies for each of them. This helps to improve productivity as well as the innovativeness of the design. At the same time, as the MSA is decentralized in the design, it offers better fault tolerance and system resilience, the reason is that a failure in one service does not compromise the entire system. Even though this distributed nature of microservices has lots of benefits, on the other hand it introduces significant amounts of complexity in terms of system design [3], management, and as well as system security.

Security Challenges Associated with Microservices

Unlike the traditional monolithic system, where there's a common database, common communication protocol and a fixed dependency, the distributed and decentralized nature of the MSA allows each microservice of the system to have their own database, communication protocols, and dependencies. This leads to several security challenges as this decentralization increases the attack surface for potential threat and vulnerabilities [4]. Similarly, this decentralized nature makes the implementation of effective security measure complicated.

Service-to-Service Communication: The communication between microservices often happens over networks, because of that there are risks associated with interception of data, unauthorized access, and man-in-the-middle attacks [5]. To avert unwanted access and manipulation of critical data, secure communication channels (e.g., encryption and authentication) usage becomes crucial.

Third-Party Dependencies: Although a microservice is generally developed to perform a single task. But during its operation it must depend heavily on external services and third-party components. This does introduce a lot of potential vulnerabilities as a vulnerable third-party library or API can compromise the entire system, even if the core microservices are secure.

Misconfigurations: When the number of services present the system increases, there's a high possibility that having misconfigurations. Improperly configured network permissions, service access controls and exposed endpoints are some examples of these misconfigurations. If these misconfigurations are not managed properly, they can lead to security breaches in the entire system [6].

Traditional security testing mechanisms such as static code analysis, traditional dynamic security analysis, manual penetration testing, and vulnerability scanning usually become inadequate to handle the dynamic and continuously evolving threats in the microservice system due to their complexity [7].

Introduction of Reinforcement Learning as a Promising Solution

Reinforcement Learning is now a useful method to tackle the issues of security test automation. Reinforcement Learning is a subdivision of machine learning in which a program learns decision-making by interacting in a world and gains feedback via rewards or penalties [8]. The program works to raise its total reward via exploring and exploiting the environment.

For microservices security, RL aids in the simulation of actual attack cases and spots weaknesses with automation and adaptation [9]. With RL methods, a program learns by adapting to new threats, this creates a dynamic solution for security vulnerability detection. In contrast to traditional testing methods, RL can identify the vulnerabilities without human intervention.

1.2. Problem Definition

Explanation of the Security Testing Problems in Microservices

At present most of the modern systems are developed using microservices [10]. Most of the traditional manual security vulnerability testing mechanisms like penetration testing and vulnerability scanning are no longer useful or productive as they are time-consuming and resource intensive. At the same time, they are unable to keep up with the dynamic nature of microservice environments as the microservices are susceptible to frequent updates and configuration changes [11].

On the other hand, most of the current security vulnerability detection systems usually pay more attention to the known vulnerability types and attack vectors. This creates a possibility that some of the newly emerging threats or vulnerabilities remain unnoticed in microservice systems [12]. Lack of support for scalability is also a concern when trying to make use of traditional security vulnerability detection methods in microservice architecture.

This shows that there is a need for automated, dynamic, and scalable security vulnerability detection methodology for microservices. This methodology should have the capability to evolve together with the microservice system continuously adapting to new threats while covering the entire system efficiently.

The Gap in Current Research

Even though there's a huge potential of utilizing **Reinforcement Learning** in the field of automating the security testing, not many research explorations have been done to identify its applicability towards the microservices environments. The focus of most prior research has been towards the applicability of RL in cybersecurity for isolated systems or single -service applications [13]. So, the complexities and challenges in identifying vulnerabilities of distributed systems like microservices remain as a largely unexplored area. Hence there is a definite gap in applying RL for the real-time detection of vulnerabilities and for the simulation of attack vectors within microservices architectures.

This study seeks to fill this gap in existing literature by focusing on RL automated security vulnerability detection and to provide a more efficient, scalable, and adaptive solution to identify vulnerabilities in microservice based systems.

1.3. Research Objectives

The primary purpose of this research is to automate the detection of vulnerabilities in microservices using Reinforcement Learning. In this research, **Proximal Policy Optimization (PPO)** [14] is used as the primary reinforcement learning algorithm.

The principal aims of this research are as follows:

- Develop a reinforcement learning based framework that can identify security vulnerabilities in a microservices environment.
- Create DoS attack vectors to launch attacks on microservices.
- Use RL to simulate realistic attack scenarios on the Online Boutique app and determine the available security vulnerabilities based on its responses to these attacks.
- Demonstrate how RL can be utilized as a more efficient and adaptive alternative to traditional manual penetration testing.

1.4. Significance of the Research

Scope of the research

As previously described in the Problem Definition section, the use of MSA has added extra complexity towards the detection of vulnerabilities in the system. It is evident that the security vulnerability detection of a microservice-based system needs to be automated. With the introduction of RL into the security testing process, this study targets to provide a more dynamic and scalable approach for security vulnerability detection in MSA. This approach can simultaneously provide real-time security vulnerability identification and constant adaptation to new threats. This is crucial for deploying. As this allows microservices-based applications to maintain alignment with the rapid development of security threats and protect against emerging risks, this is a very crucial thing for organizations that deploy such microservice based systems.

The importance of this research extends beyond academic contributions. Outcomes of this can be applied in the development of more secure, automated security testing frameworks for microservices. It will potentially help to transform how security is approached in the software industry.

1.5. Structure of the Thesis

The thesis is organized as outlined below:

- **Chapter 1: Introduction**: The research problems, background, and objectives are explained in this chapter. It outlines microservices architecture and the security challenges it presents. The application of reinforcement learning for automating security vulnerability identification within microservices is also discussed in this chapter.

- **Chapter 2: Literature Review**: Current research on microservices in terms security, vulnerability detection methodologies, reinforcement learning, and the utilization of reinforcement learning in cybersecurity are reviewed in this chapter. It identifies gaps in the existing research and substantiates the integration of reinforcement learning for the automation of security testing in microservices.

- **Chapter 3: Methodology**: The approach used to construct the RL-based security testing framework is described in this chapter. It encompasses specifics regarding the design of the PPO agent, the configuration of the environment (Online Boutique), and the simulation of attack vectors. It also goes into detail about the evaluation criteria used to determine the framework's effectiveness.

- **Chapter 4: Implementation**: This chapter details the system architecture. These covers setting up the PPO agent and simulating the attack. It also outlines the technical steps taken to build the testing framework and explains the way it integrates with the Online Boutique application.

- **Chapter 5: Observations, Results and Analysis**: The outcomes of the assessment of the PPO-based system are shown in this chapter. This covers the agent's performance specifically in catching vulnerabilities. The chapter also contrasts the RL-based framework with manual penetration testing in terms of accuracy, efficiency, and scalability.

- **Chapter 6: Discussion and Conclusion**: The results of this study are summarized in this chapter. It also considers the consequences for practice and makes

recommendations for future research. The possibility of expanding the framework to additional domains and applications is also mentioned here.

2. LITERATURE REVIEW

This section covers the related existing work in the following areas.

- Benefits and associated security challenges of microservice architecture.
- Utilization of reinforcement learning in cybersecurity and security vulnerability detection.
- Application of Proximal Policy Optimization as a reinforcement learning algorithm.

2.1. Microservice Architecture

In Microservice architecture a single application is divided into an aggregation of small services. Every one of these services is built as an autonomous entity that manages its own responsibilities. These services are loosely coupled, and **Application Programming Interfaces** (**API**) are used when interacting with each other.

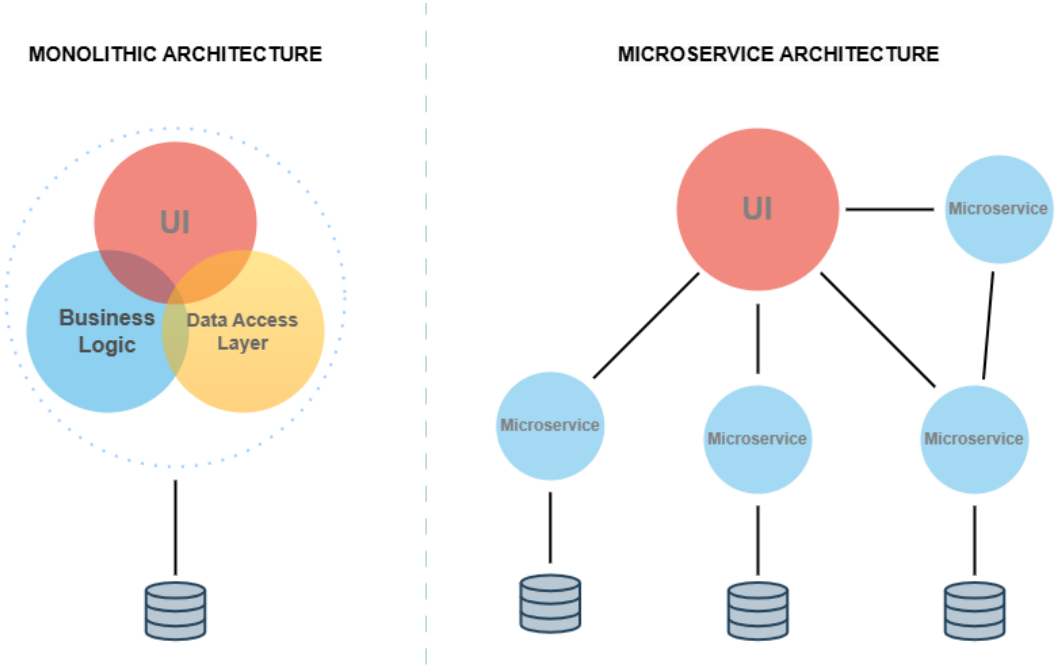

Figure 2.1 Comparison between Monolithic architecture and Microservice architecture

2.1.1. Characteristics and Advantages of Microservices

Microservices possess numerous advantages:

- **Scalability**: Microservices can be scaled independently as their functionality is independent from each other. This helps to manage resource allocation and cost in an optimum manner [15].

- **Flexibility**: Microservices offer developers the option to use various technologies for different services. This aids in selecting the most suitable technology for a particular service or a task [16].

- **Fault Tolerance**: Microservices function independently. Therefore, failure in one service does not necessarily affect other services. This makes a microservice-based system more resilient towards faults [17].

Even though microservices offer multiple benefits related to scalability and flexibility, compared with the traditional systems microservices introduce additional complexity when it comes to inter-service communication and system security.

2.1.2. Security Challenges in Microservice Architectures

The following are some of the common security challenges that microservices encounter due to their decentralized and distributed nature.

- **Service-to-Service Communication**: When microservices interact with each other, communication usually happens via a network. This creates a potential vulnerability of data interception and unauthorized access. Therefore, it's essential to secure this inter-microservice communication with Transport Layer Security (TLS) encryption [18] and it's challenging to manage these encryptions in a large distributed system.

- **Dependency on Third-Party Libraries**: For the successful functionality of microservices, they usually depend on multiple external dependencies. Managing these external dependencies properly is crucial for the security of overall system. Any vulnerability that is present in these 3rd party libraries can be exploited by attackers.

- **Configuration Management**: Maintaining consistent security configurations across all the services present in the system is challenging. Failing to prevent misconfigurations like improper access controls can lead to potential breaches of system security [19].

- **Service Interactions**: During their operation, microservices do interact with several other microservice. As a result of that it is difficult to difficult to prevent the propagation of vulnerability from one microservice to another [20]. This leads to a magnification of the potential attack surface.

These are some of the unique challenges which are inherent to microservices and because of that security tools which were designed for monolithic architecture may become inadequate in identifying vulnerabilities of microservice-based systems.

2.2. Reinforcement Learning

Reinforcement learning is a type of machine learning where decision making is done by autonomous agents. An agent acquires knowledge to make decisions through his engagement with the environment. The agent performs actions and obtains feedback for those actions by means of rewards or penalties, then modifies own behavior over time to optimize cumulative rewards.

2.2.1. Introduction to Reinforcement Learning

The fundamental elements of reinforcement learning problems are:

- **Agent**: The one who learns or makes decisions.
- **Environment**: The setting in which the agent functions.
- **State**: An illustration of the environment's present condition.
- **Action**: A collection of decisions which are available for the agent to make at a given state.
- **Reward**: A scalar value the agent receives when action is performed based on the outcome of it.

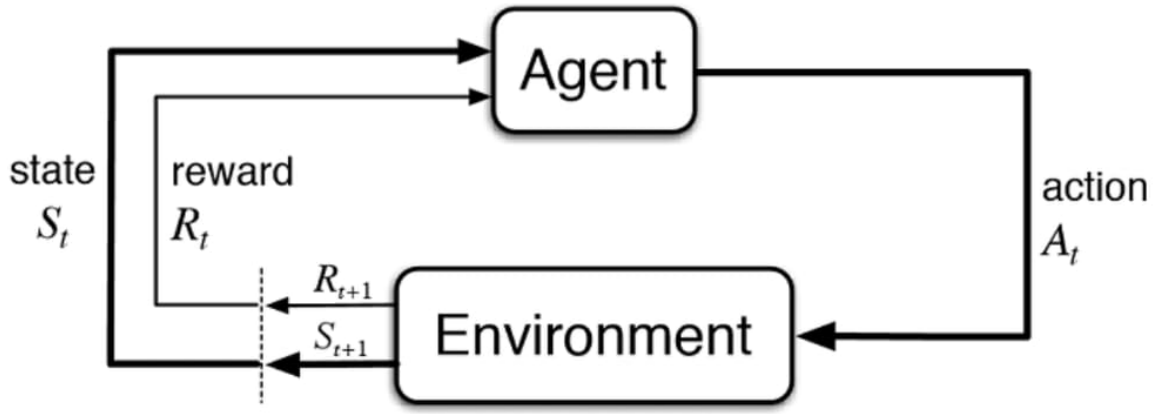

Figure 2.2 An overview of a RL algorithm.

2.2.2. RL Algorithms and Techniques

The reinforcement learning algorithms that are employed most frequently are as follows:

- **Q-Learning**: A model-free, value-based RL algorithm which finds the best set of actions depending upon the agent's current state. The agent follows trial and error approach to identify the actions that that result in positive outcomes. Q-learning has a slow learning rate as it needs to explore the whole state space and is usually inefficient in complex environments. Its actions are limited to discrete values hence most of the time it's less suitable for real-world applications [21].
- **Deep Q-Networks (DQN)**: An enhanced variant of the Q-learning technique that uses a deep neural network to approximate the state-value function in lieu of a Q-table. This makes it a better choice for the infinitely large state spaces, but it takes longer to train when compared with the Q-learning [22].
- **PPO**: PPO is an on-policy algorithm which uses first-order optimization function (the clip function) to constrain the overly large policy updates while achieving the biggest possible step on the policy. PPO maintains a balance between exploration & exploitation and works well both discreate and continuous action spaces. It's relatively less computationally complex and offers better stability and sample efficiency [23].

2.2.3. Applications of RL in Cybersecurity

- **Intrusion Detection Systems (IDS)**: An IDS is a device or software which is used to identify known malicious activity in a network by monitoring its traffic. RL has been used to develop adaptive IDS that are learning to identify novel attack patterns while continuously interacting with the environment [24].

- **Automated Penetration Testing**: Penetration testing is the process of identifying vulnerabilities of a system by purposefully creating attacks. RL has been used to simulate and automate penetration testing [25]. It has been found that the use of RL helps to achieve more comprehensive security testing by generating a wide range of attack scenarios that human testers might miss.

- **Security Policy Optimization**: Security policies of a system need to be standardized and managed properly to maintain overall security. RL has been used for optimizing security policies by learning the most effective configurations through interactions with the system [26].

RL showcases great potential in the ability to continuously learn from attacks and adapt security strategies in real-time. This is a major advantage that RL possesses in the context of microservices, when compared with the traditional security methods.

2.3. Security Vulnerability Detection in Microservices

2.3.1. Traditional Security Approaches in Microservices

Traditional security approaches, such as manual penetration testing, static code analysis, traditional dynamic security analysis, and vulnerability scanning, are often ineffective in microservices environments due to the rapid and dynamic nature of microservice systems. Static analysis tools may not capture runtime vulnerabilities. Static analysis in a microservice based system must analyze the code of an individual service in isolation and it misses the potential security vulnerabilities which arise due to the interaction between the services. Traditional dynamic security analysis tools rely on predefined attack scripts and patterns hence they fail to capture novel security vulnerabilities in an evolving system. Traditional dynamic security analysis also has limitations in terms of scalability and continuous monitoring. Similarly, manual penetration testing cannot be scaled efficiently with the increasing number of services in a microservices ecosystem.

2.3.2. Automated Security Vulnerability Detection and Mitigation

Automated tools are increasingly used to address security in microservices. These tools can continuously monitor vulnerabilities, analyze traffic patterns, and identify misconfigurations in real-time [27]. For example, dynamic analysis tools can simulate attacks to security identify vulnerabilities that may be exploited during runtime, while orchestration tools can be used to automate the deployment of security patches across multiple services.

However, automated tools still face limitations in adapting to new vulnerabilities that arise from the interaction of services, especially in dynamic environments.

2.3.3. Limitations of Current Approaches and Need for Innovation

Despite the advancement in automated tools, the dynamic, distributed nature of microservices still makes it difficult to continuously detect and mitigate vulnerabilities. Many current approaches struggle with scalability, adaptability, and real-time updates, leading to gaps in security coverage [28].

There is a need for intelligent, adaptive security solutions that can continuously learn from attacks and evolve to mitigate new threats as they emerge, which is where reinforcement learning offers significant promises.

2.4. PPO in Security Testing

2.4.1. Introduction to PPO

PPO is a method for optimizing policies that maintains a balance between performance and computational efficiency. PPO is designed to minimize large, unstable policy updates by imposing a limitation on the policy update step. PPO is more stable and simpler to implement than other policy gradient approaches, like **Trust Region Policy Optimization (TRPO)** [29].

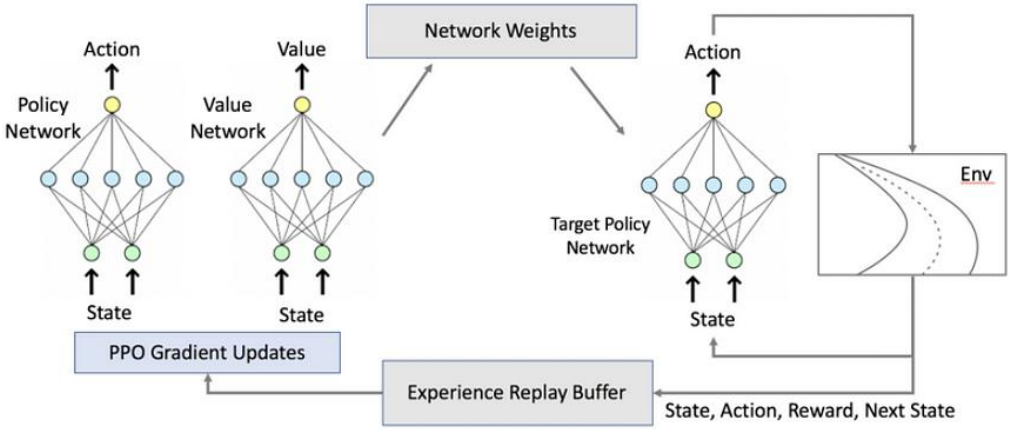

Figure 2.3 In illustration of PPO algorithm.

2.4.2. Application of PPO in Cybersecurity

PPO has been applied to several cybersecurity tasks:

- **Automated Penetration Testing**: PPO has been used to simulate attack strategies and optimize the detection and mitigation of vulnerabilities, learning from interactions with a simulated environment.
- **Intrusion Detection**: PPO has been used to dynamically adjust detection thresholds based on ongoing attack simulations, enabling real-time responses to evolving threats [30].

2.4.3. Benefits and Challenges of PPO for Security Vulnerability Detection

PPO offers several benefits for security vulnerability detection:

- **Scalability**: PPO can handle complex environments, making it ideal for large-scale microservice ecosystems [31].
- **Adaptability**: PPO can continuously learn and adapt to new attack strategies as they emerge, providing real-time updates to security measures.
- **Stability**: PPO's stability ensures reliable updates to security policies without causing system disruptions.

However, challenges include the computational resources required for training and the need for an adequately extensive and varied dataset of attacks to train the agent effectively.

3. METHODOLOGY

3.1. Research Design

3.1.1. Overview of the Research Approach

There are multiple ways of identifying security vulnerabilities of microservices and ensuring security. Penetration testing is one such methodology where real-world attacks are simulated to identify the exploitable vulnerabilities of microservices and the infrastructure which the microservices are associated with. Penetration testing paves the way to uncover a lot of weaknesses of a microservice system to the system owner. Even though penetration testing is quite useful in terms of the results it generates, there are several limitations and drawbacks associated with it. Given below are few such limitations.

- **Limited scope:** Most of the time penetration testing is conducted only on a few components of the microservice system due to limited resources. Hence the whole system or all the components of the microservice system are not tested thoroughly. As a result of that some of the vulnerabilities that are present in the system may not be detected.
- **Limited by Human Factor:** Penetration tests are designed by penetration testers. The methodologies followed by one penetration tester can vary from the other resulting inconsistent results [32]. At the same time some of the vulnerabilities may go undetected due to oversight or the lack of knowledge and other potential mistakes of the penetration tester.
- **Time and Resource constraints:** Conducting comprehensive penetration testing on a microservice architecture takes considerable time and effort. Oftentimes the time available for testing is shortened due to business pressures and tight deadlines. Shorter testing periods lead to the creation of penetration tests which fail to cover all the aspects of the system leaving potential vulnerabilities undetected [33].

Scalability of manual penetration testing becomes challenging as a typical microservice system consists of hundreds of microservices and each of them may have different configurations.

Static code analysis is another way of identifying vulnerabilities present in microservices where it looks for predefined set of vulnerabilities. While this is useful in identifying flaws in microservices, it is not so useful in identifying possible threats from new attack vectors. Distributed and loosely coupled nature of the microservices allows microservices in a system to be designed in different languages and frameworks. Therefore, a static analysis tool which

is effective for a particular language may be able to cover the holistic view on a microservice system. As a result of that the security vulnerability detection capability of static code analysis is usually limited to standalone services of the system, and it misses the vulnerabilities that come up due to the interaction between services.

Dynamic analysis traditionally focuses on testing individual services or endpoints from the outside, but microservices often rely on complex inter-service communications via APIs, messaging queues or event-driven patterns. Traditional dynamic security analysis may not capture the vulnerabilities that arise from the interaction between services (e.g., insecure communication, unauthorized access, data leaks between services). Dynamic analysis generally performs a set of predefined tests on running systems with the assumption that system is more or less fixed during the test. However, since microservices are distributed and constantly evolving, the services can be added, removed or updated at any time. As the dynamic analysis tools are not adaptive to these dynamic changes in the system, it requires manual effort to reconfigure the dynamic analysis scheme.

In most cases microservice-based systems opt for Continuous Integration and Continuous Deployment (CI/CD) based development practices. This results in the addition of new services or updates to the system multiple times a day. As a result of that the microservice-based system often keeps on evolving daily. Conventional manual penetration testing, static code analysis and traditional dynamic security analysis cannot keep up with this rate as they usually have limitations in terms of their applications owing to limitations of time, resources and adaptability.

The methodology presented in this study targets to overcome these limitations by automating the process of security vulnerability detection in MSA with the help of RL. This methodology leverages dynamic analysis principles, where the system is continuously tested in real-time through simulated attacks.

The key innovation of this approach lies in the use of RL in implementing RL agent which interacts with the running microservice system by launching attacks on microservices. The way that the system responds to these attacks allows the RL agent to learn and identify security vulnerabilities present in the system.

This methodology uses PPO as the RL algorithm, and it provides the way implement an adaptive, data-driven approach to continuously identify security vulnerabilities of MSA.

This algorithm is mapped to proposed methodology as follows.

1. **Reinforcement Learning Environment:** An agent was created to generate attacks on microservices and the microservice system is modeled as an environment where this agent can interact with the components the microservice system. This environment consists of diverse types of microservices, the interconnections between the microservices and potential vulnerabilities.
2. **States, Actions, and Rewards:**
    - **States:** Current configuration and the security of the system are represented by the states. These include things such as service health, exposed endpoints and access control policies.
    - **Actions:** Potential attack vectors that the Reinforcement learning agent can explore are represented by the actions. These encompass simulated assaults like denial-of-service (DoS) attacks.
    - **Rewards:** The incentives or the recognition that the agent receives based on the effectiveness of detecting vulnerabilities are represented with the rewards. The agent receives a positive reward when it successfully detects vulnerability while it receives a negative reward when it is unable to detect a vulnerability from the launched attack.
3. **Policy Optimization:** The reinforcement learning agent based on PPO engages with the reinforcement learning environment by choosing different actions (attacks) based on the policy from which it has learned. Meanwhile the agent is continuously updated by the PPO algorithm to minimize the large and unstable policy updates. This ensures stability during the learning process. This recurrent procedure enables the agent to enhance its vulnerability detection capabilities over time.
4. **Continuous Learning:** Microservice systems evolve constantly with the addition of new services and updating of existing services. Hence, new vulnerabilities get added into the system continuously. The reinforcement learning agent deployed in the system can continuously learn and adapt in real-time with the system, this dynamic learning ensures that the agent's ability to uncover new vulnerabilities even as the system evolves.

3.1.2. Justification for Using PPO in Security Vulnerability Detection

Proximal Policy Optimization offers several innovative features that solve the difficulties in modern microservices by automating security vulnerability detection through attack injections.

- **Adaptability to New Threats:** In traditional dynamic security vulnerability detection approaches the process relies on a fixed set of attack patterns. So, it has limited capability to adapt to new threats. Conversely, the PPO based agent follows the approach of continuous learning via the interaction between it and the system. Therefore, the agent can adapt to the environment as the system evolves. As a result of that PPO agents can detect previously unknown vulnerabilities.
- **Real-time Detection:** Unlike the traditional security vulnerability mechanisms which are performed in a periodic manner, PPO based security vulnerability detection can be performed as part of microservice lifecycle. This allows system administrators to identify the security vulnerabilities in real-time by injecting attacks during development or as a part of deployment process.
- **Scalability:** It is somewhat challenging to scale traditional manual testing schemes as the microservice systems grow in terms of size and complexity, but PPO based agents can handle the increased complexity by simultaneously launching multiple attack vectors over multiple services with the help its dynamic learning ability which allows it to identify the most effective attack strategies.
- **Efficiency:** Manual penetration testing is a laborious technique that demands significant effort. Most of the time this process yields limited coverage. On the contrary, the PPO based agent helps to minimize the time and other resources required for testing as it automates the process of identifying the vulnerabilities. Ath the same time it ensures comprehensive coverage as it keeps on evaluating the environment continuously as a part of its learning process.
- **Identification of Subtle Security Vulnerabilities:** PPO based agent has the ability of uncovering subtle vulnerabilities in the system such as logical flaws during microservice-to-microservice interactions and configuration weaknesses as it keeps on injecting a variety of attacks. In traditional security vulnerability testing, these vulnerabilities can remain undetected as they are not apparent.

Proximal Policy Optimization is selected as the proposed methodology as the reinforcement learning algorithm to implement the agent due to its stability, efficiency and suitability for

handling complex environments like microservice security vulnerability testing. Similarly, PPO is well suited for tasks that require continuous learning and adaptation of the agent in new and evolving environments. Dynamic nature of microservices creates one such environment. Stability during the training process is one of the key advantages that PPO possesses over the RL algorithms. Even though the traditional policy optimization methods like TRPO are quite effective in terms of the training stability, they require substantial amounts of data and long training times. The **clipped objective** introduced in the PPO helps to constrain the policy update size preventing excessively large policy updates. This allows PPO to achieve more practical and efficient solutions while maintaining stability in training.

Similarly, PPO aims to be more **sample-efficient** than most RL algorithms by achieving reasonable performance after few interactions with the environment. In simpler RL methods such as **Q-Learning**, agents need to do a lot of trial and error within the environment before they can implement an optimal policy, which is expensive and takes a while to do, especially for advanced systems like microservices. PPO uses surrogate loss functions to control policy update optimizations, allowing it to make small, stable, and less informative updates. Efficiency in sample usage is very important when the agent is trying to uncover vulnerabilities in a computationally efficient manner in a real-time dynamic environment like microservices.

The action space of the task of microservice security vulnerability analysis can become extremely **high-dimensional**. There are numerous different attack vectors that require testing, and configuration changes, attack injections, and security procedures must be adapted in real-time by the RL agent. Traditional algorithms in reinforcement learning like Q-Learning are more appropriate for environments with discrete action choices, and they do not scale well to continuous or large action scope environments. PPO is often a better choice for **high-dimensional continuous action spaces** because it uses **policy** gradient approaches instead of an action-value approach. With PPO, the agent defines an action as a probability distribution over available actions and the policy is adjusted in response to defined rewards. This makes it effortless to deal with many possible attack strategies and configurations that can be taken in microservices environments.

Given below is a comparison between PPO vs Deep Q-Learning (DQN) and Asynchronous Advantage Actor-Critic (A3C) [34] with respect to the requirements of this research.

Table 3.1 Comparison of RL algorithms

Aspect	**PPO**	**DQN**	**A3C**
Action Space	Continuous, high-dimensional action space	Discrete action space	Continuous, but with higher complexity due to multiple agents
Stability	High stability with clipped objective function	Less stable, prone to oscillations	Less stable, especially in large environments
Sample Efficiency	High, optimized for efficient learning	Low, requires large amounts of exploration	Moderate, but requires multiple agents for faster learning
Suitability for Real-Time Applications	Exceptionally appropriate for real-time, dynamic environments	Less appropriate for real-time, dynamic environments	Moderate, but computationally expensive
Computational Overhead	Low	Low	High, requires multiple parallel agents

When compared with the other popular RL algorithms that are relevant for this research area, PPO stands out the most **stable**, **sample-efficient** and **computationally less complex** one for the analysis of vulnerabilities in microservice architecture.

3.2. System Setup and Environment Configuration

Online Boutique (also known as Hipster Shop**)** application is used as the microservice system in this research and it configured and deployed to serve as the testbed for this security vulnerability analysis. This section explains the details of the Online Boutique application and how it is deployed for the purpose of this research.

3.2.1. Deploying the Online Boutique Application

Online Boutique is an open-source sample e-commerce application which was originally developed by Google. Google typically employs this application to exhibit the utilization of technologies such as Kubernetes, gRPC, OpenCensus, Stackdriver, and Istio. This microservice based application is designed to mimic a real-world e-commerce platform, and it contains components such as **product catalog**, **shopping cart**, **checkout service**, and **recommendation service**.

For this research the microservices of the Online Boutique application are built and deployed in Kubernetes cluster with a single node which runs on the local development machine. This process involves several steps, and they are listed below in detail.

1. System Requirements

Given below are the system requirements that need to be fulfilled to deploy Online Boutique successfully in a local machine.

Table 3.2 System Requirements for Online Boutique Local Deployment

Category	**Requirement**	**Details/Version**
Hardware	CPU	at least 4 cores
	RAM	8 GB minimum (16GB recommended)
	Storage	30GB available disk space
	BIOS/UEFI	Virtualization support enabled

Software	Operating System	Ubuntu 20.04 LTS or Windows 10/11 with WSL2
	Containerization	Docker Engine 20.10.x or newer
	Kubernetes CLI	kubectl v1.23 or newer
	Version Control	Git client
	Package Management	Helm

2. Minikube Cluster Setup

Minikube can be used to create one node Kubernetes cluster on the local machine. This cluster is used as the environment to host the Online Boutique application.

Minikube cluster is configured with 6 GB of memory and 6 CPU cores to enable sufficient resources for microservices and RL agent deployments.

```
PS C:\Dev\MSc\project\microservices-demo> minikube start --memory=6144 --cpus=6 --driver=docker
* minikube v1.35.0 on Microsoft Windows 11 Pro 10.0.26100.3476 Build 26100.3476
* Using the docker driver based on user configuration
* Using Docker Desktop driver with root privileges
* Starting "minikube" primary control-plane node in "minikube" cluster
* Pulling base image v0.0.46 ...
    > gcr.io/k8s-minikube/kicbase...:  500.31 MiB / 500.31 MiB  100.00% 64.35 M
* Creating docker container (CPUs=6, Memory=6144MB) ...
! Failing to connect to https://registry.k8s.io/ from inside the minikube container
* To pull new external images, you may need to configure a proxy: https://minikube.sigs.k8s.io/docs/reference/networking/proxy/
* Preparing Kubernetes v1.32.0 on Docker 27.4.1 ...
  - Generating certificates and keys ...
  - Booting up control plane ...
  - Configuring RBAC rules ...
* Configuring bridge CNI (Container Networking Interface) ...
* Verifying Kubernetes components...
  - Using image gcr.io/k8s-minikube/storage-provisioner:v5
* Enabled addons: storage-provisioner, default-storageclass
* Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
PS C:\Dev\MSc\project\microservices-demo>
```

Figure 3.1 Starting of the Minikube cluster.

Successful startup of the Minikube cluster can be verified as follows.

```
PS C:\Dev\MSc\project\microservices-demo> minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured

PS C:\Dev\MSc\project\microservices-demo> kubectl get nodes
NAME       STATUS   ROLES           AGE     VERSION
minikube   Ready    control-plane   5m32s   v1.32.0
```

Figure 3.2 Verifying Minikube's startup process.

3. Downloading and Preparing the Online Boutique Application

As the first step in deploying the Online Boutique, it needs to be cloned from its GitHub repository (https://github.com/GoogleCloudPlatform/microservices-demo).

```
PS C:\Dev\MSc\project> git clone https://github.com/GoogleCloudPlatform/microservices-demo.git
Cloning into 'microservices-demo'...
remote: Enumerating objects: 19154, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 19154 (delta 2), reused 0 (delta 0), pack-reused 19148 (from 2)
Receiving objects: 100% (19154/19154), 34.65 MiB | 162.00 KiB/s, done.
Resolving deltas: 100% (14725/14725), done.
PS C:\Dev\MSc\project> cd microservices-demo
PS C:\Dev\MSc\project\microservices-demo>
```

Figure 3.3 Cloning of the Online Boutique source code.

Once the cloning is finished, the Online Boutique application can be deployed using kubectl.

```
PS C:\Dev\MSc\project\microservices-demo\release> kubectl apply -f .\kubernetes-manifests.yaml
deployment.apps/emailservice created
service/emailservice created
serviceaccount/emailservice created
deployment.apps/checkoutservice created
service/checkoutservice created
serviceaccount/checkoutservice created
deployment.apps/recommendationservice created
service/recommendationservice created
serviceaccount/recommendationservice created
deployment.apps/frontend created
service/frontend created
service/frontend-external created
serviceaccount/frontend created
deployment.apps/paymentservice created
service/paymentservice created
serviceaccount/paymentservice created
deployment.apps/productcatalogservice created
service/productcatalogservice created
serviceaccount/productcatalogservice created
deployment.apps/cartservice created
service/cartservice created
serviceaccount/cartservice created
deployment.apps/redis-cart created
service/redis-cart created
deployment.apps/loadgenerator created
serviceaccount/loadgenerator created
deployment.apps/currencyservice created
service/currencyservice created
serviceaccount/currencyservice created
deployment.apps/shippingservice created
service/shippingservice created
serviceaccount/shippingservice created
deployment.apps/adservice created
service/adservice created
serviceaccount/adservice created
```

Figure 3.4 Deployment of the Online Boutique application.

It takes a few minutes to complete the deployment process and once it's deployed properly, deployed services can be verified as follows.

```
PS C:\Dev\MSc\project\microservices-demo\release> kubectl get pods
>>
NAME                                     READY   STATUS    RESTARTS   AGE
adservice-8568877bf9-ddns9               1/1     Running   0          5m21s
cartservice-f84bf7dd4-q8png              1/1     Running   0          5m22s
checkoutservice-5d9894c787-gh6z7         1/1     Running   0          5m23s
currencyservice-84459c6759-bqtlj         1/1     Running   0          5m21s
emailservice-6fb4dd89fc-k22b2            1/1     Running   0          5m23s
frontend-754cdbf884-xwnqw                1/1     Running   0          5m22s
loadgenerator-696d89b74f-h7dr4           1/1     Running   0          5m22s
paymentservice-5575668b5c-f2fvw          1/1     Running   0          5m22s
productcatalogservice-59cf6fd7b5-vdgqj   1/1     Running   0          5m22s
recommendationservice-589895488f-tfrpz   1/1     Running   0          5m22s
redis-cart-c4fc658fb-747wg               1/1     Running   0          5m22s
shippingservice-fb4c9695c-pspdv          1/1     Running   0          5m21s
```

Figure 3.5 Status of the pods holding the services.

Status of the Minikube cluster can also be observed with the help of the Minikube's dashboard.

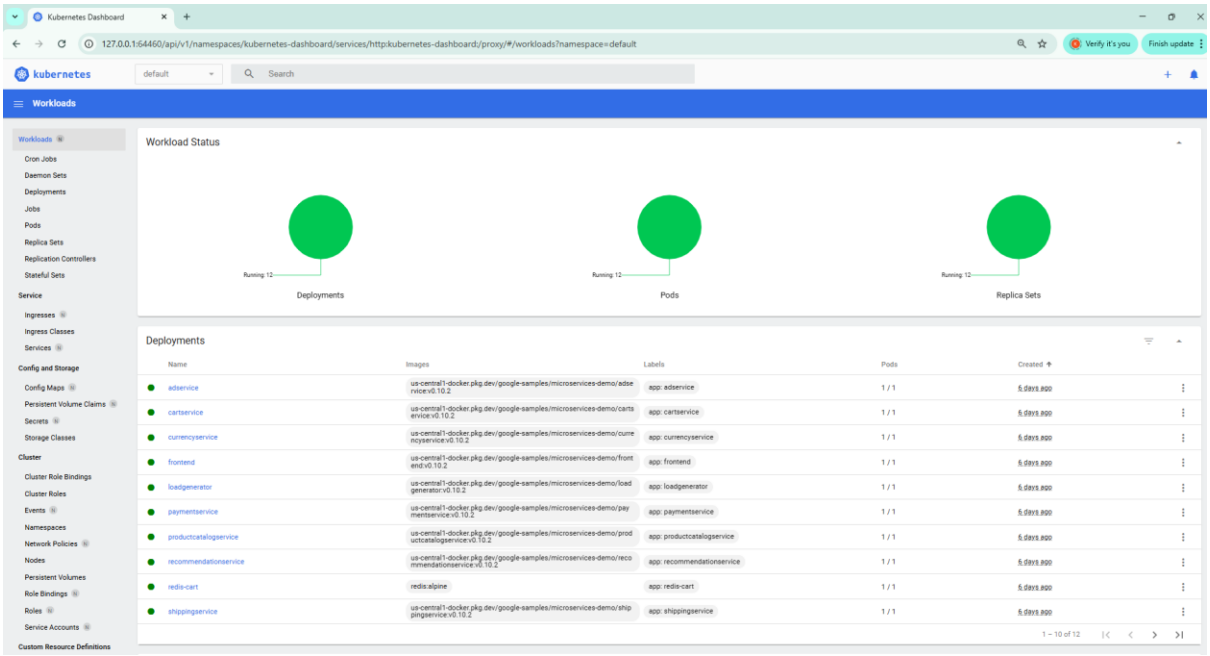

Figure 3.6 Minikube dashboard.

Online Boutique can be accessed locally by exposing the “frontend” service via an external IP address. Minikube’s built-in service exposure method can be used for this purpose.

```
PS C:\Dev\MSc\project\microservices-demo\release> minikube service frontend --url
* service default/frontend has no node port
! Services [default/frontend] have type "ClusterIP" not meant to be exposed, however for local development minikube allows you to access this !
http://127.0.0.1:56997
! Because you are using a Docker driver on windows, the terminal needs to be open to run it.
```

Figure 3.7 Exposing the URL for frontend service.

This command exposes the service and returns the URL to access the application. This URL is used to access the application via the web browser. Once it’s accessed, the deployed Online Boutique application appears as follows.

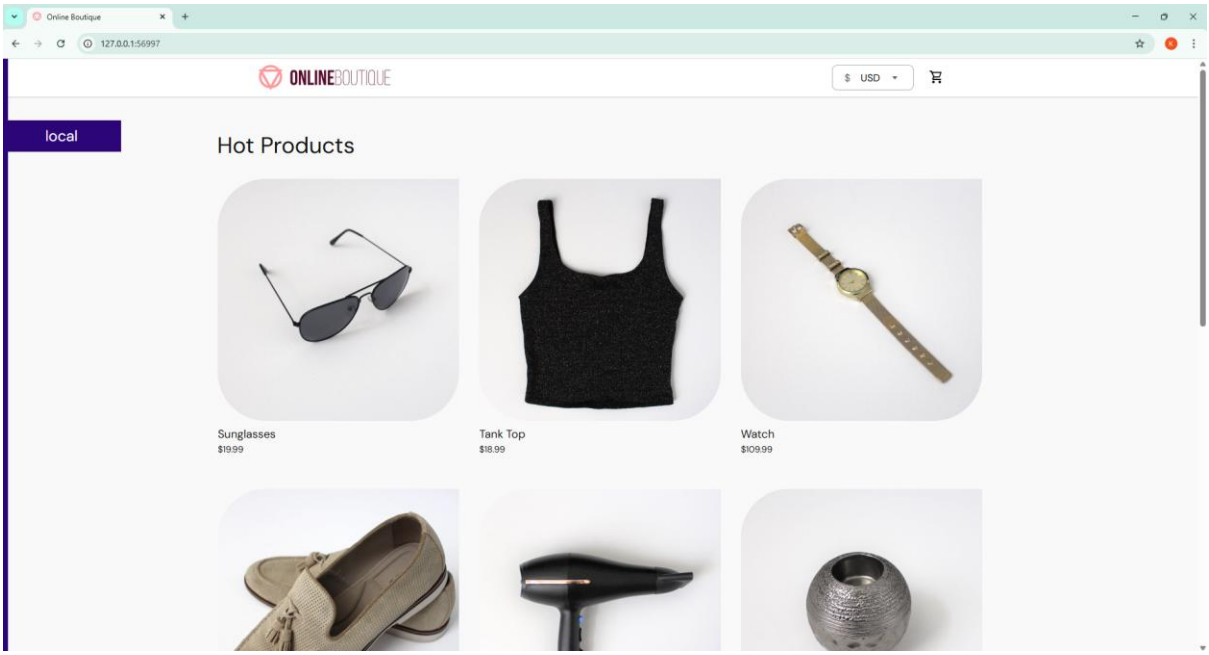

Figure 3.8 Online Boutique's Home page.

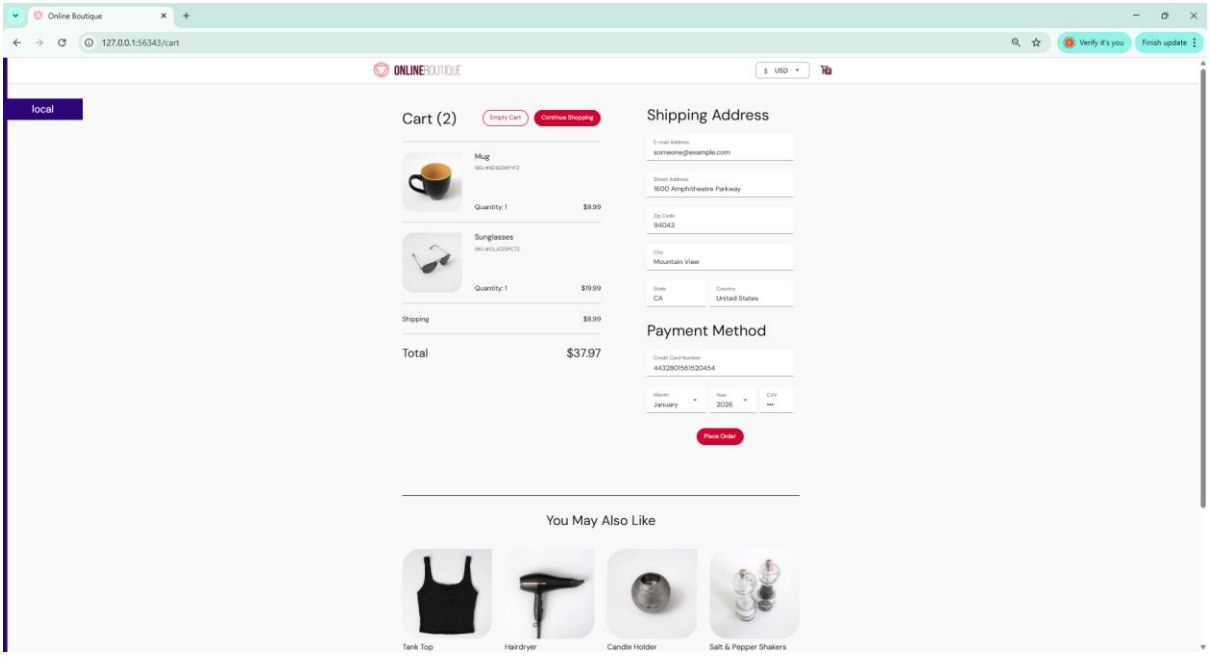

Figure 3.9 Online Boutique's cart page.

3.2.2. Microservices Architecture Setup

Online Boutique is an application which consists of multiple independent microservices. The communication among the microservices performed via well-defined APIs. Below diagram shows how each microservice interacts with the other during their operation.

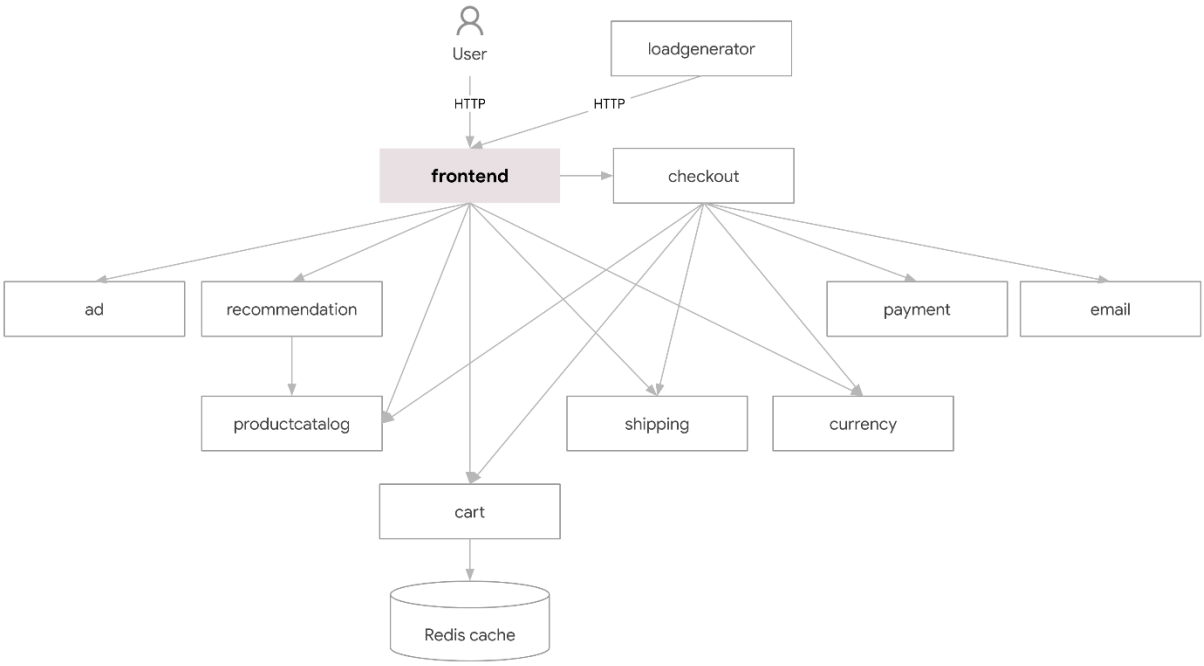

Figure 3.10 Architecture of the Online Boutique application.

Each of the microservice present in the Online Boutique application performs a unique task while interacting with the other services. Below table summarizes the individual functionality of the microservice and the technology stack used to develop these services.

Table 3.3 Service Description of Online Boutique Application

Service	**Language**	**Description**
frontend service	Go	Used to expose an HTTP server to host the website. It produces session IDs automatically for all users and doesn't need users to sign up or login.
cart service	C#	Used to store the items of the user's shopping cart in the in Redis and to retrieve it.
productcatalog service	Go	Used to offer a catalog of items from a JSON file, together with the capability to search for and retrieve specific products.

currency service	Node.js	Used to convert one currency to another. It uses real currency data obtained from the European Central Bank and has the highest Queries Per Second (QPS).
payment service	Node.js	Used to charge the given amount from the credit card info provided and return a transaction ID.
shipping service	Go	Used to provide estimates for shipping costs depending on the products added to the shopping cart and to dispatches items to the specified address.
email service	Python	Used to send the order confirmation email to the users.
checkout service	Go	Used to retrieve the user cart, prepare order and orchestrate the payment, complete shipping with email notification.
recommendation service	Python	Used to create recommendations for alternative items derived from the items present in the cart.
ad service	Java	Used to provide textual advertisements derived from specified contextual keywords.
loadgenerator service	Python/Locust	Used to send requests continuously emulating authentic consumer buying experiences in the frontend.

3.3. Development of the PPO-Based Security Testing Framework

The development of the PPO-based Security Testing Framework focuses on leveraging Proximal Policy Optimization to automate security vulnerability detection through simulated attacks within a microservices environment. This section will detail the custom RL

environment, the process of defining **states**, **actions**, and **rewards**, and the architecture of the **PPO agent** used for continuous learning and security vulnerability detection.

3.3.1. Custom RL Environment for Attack Simulation

A custom RL environment is essential for simulating real-world attack scenarios to test the vulnerabilities of the Online Boutique application. The RL environment acts as a simulator where the PPO agent interacts with the application, taking actions (e.g., launching attacks) and receiving feedback (rewards or penalties). This environment simulates the interaction of microservices and the process of identifying vulnerabilities via realistic attack vectors.

Attack Simulation:

The environment is set up to mimic several common types of attack vectors. This study focuses mainly on the development of **Denial of Service (DoS)** attacks.

DoS attacks try to render the system unreachable by flooding the program with too much traffic, therefore incapacitating the software. Here the RL-based agent creates a spike in requests or a denial-of-service attack by producing network traffic directed at the application's services. The reward system, which monitors the system's failure or resilience under pressure, offers incentives for efficient service failure identification.

Environmental Feedback:

The RL environment continuously tracks system performance metrics during these attack simulations. The following details are obtained from the environment to understand the impact of the launched attack.

- **Service availability**: Whether the service remains responsive during attacks.
- **Error logging**: Whether the system logs errors or exceptions caused by attacks.
- **Response time**: Whether attack attempts cause significant delays in system responses.

The **environment** is designed to simulate multiple attack vectors, and the **PPO agent** observes the system's state and reacts accordingly, either by launching an attack or adapting based on feedback from previous actions.

3.3.2. Defining States, Actions, and Rewards

The state, action, and reward constitute the fundamental elements of reinforcement learning that direct the agent's learning process. These components need to be carefully defined to align with the goals of detecting vulnerabilities in microservices.

States (Security Posture):

The state represents the security status of the Online Boutique application at any given moment. It encapsulates the current condition of the system and all relevant security-related aspects. The states are designed to equip the PPO agent with information related to the health of the application, including:

- **Service status**: Whether each microservice (e.g., product catalog, cart service, payment service) is running, and whether it is exposed to vulnerabilities (e.g., misconfigurations or weak authentication).
- **Input validation status**: Whether input fields (like search boxes or user registration forms) are properly sanitized or if they are vulnerable to injection attacks.
- **Access control configuration**: Whether authentication and authorization mechanisms are properly configured or if they are vulnerable to attacks (e.g. access controls, token validation).
- **Attack surface**: Whether the system is exposed to various attack vectors (i.e. number of exposed endpoints, insecure APIs, or other potential attack surfaces).

Each of these states possess numerical values to denote current condition of various components of the system.

Actions (Attack Strategies):

The **actions** refer to the possible decisions the PPO agent can make to interact with the environment. When an agent performs an action, it directly affects the environment's state. PPO agent's actions can either be related to the simulation of attack vectors or making observations on the environment.

For example, one such action of the PPO agent is sending traffic floods where the agent launches requests to simulate a DoS attack and observe whether the system can handle the load.

Each action of the PPO agent creates an influence on the state of the system. As a result of that it may either trigger a security vulnerability detection mechanism or exploit an existing vulnerability.

Rewards (Success or Failure of Detection):

PPO agents are guided towards optimal behavior with the help of the reward system. Rewarding is contingent upon the outcomes of the action. The framework of it is defined as follows:

- **Positive Reward**: The agent is rewarded with a positive reward if it is successful in detecting a security vulnerability from the action done. The value of the reward is proportional to the severity of the vulnerability detected (e.g. higher reward is offered if critical vulnerabilities are found).
- **Negative Reward**: If the agent fails to identify a security vulnerability or the attack does not lead to a successful exploitation, it receives a negative reward or zero reward. This allows the agent to learn to avoid ineffective attacks and focus on actions that lead to vulnerability detection.
- **Penalties**: In case the agent generates false positive (i.e. incorrectly identifying a security vulnerability that doesn't exist) or introducing false negatives (i.e. failing to detect an existing vulnerability) will be penalized for it.

This rewarding system helps to keep the agent learning from the engagements with its environment while maximizing the vulnerability detection rates.

3.3.3. PPO Agent Architecture and Implementation

The following section explains architecture and the implementation of the PPO agent.

PPO Agent Architecture

The architecture of the PPO agent consists of two primary elements:

- Policy Network
- Value Network

Policy Network

The policy network is accountable for determining which action (attack) to take given the current state of the system. It maps the observed state (security posture) to a probability distribution over possible actions (e.g. attacking with SQL injection or sending network traffic).

The network is typically a **deep neural network (DNN)** that comprises several layers:

- **Input Layer**: The security posture (state) is fed as input. This might include service status, error logs, and vulnerability indicators.
- **Hidden Layers**: Several fully connected layers that process the input state, extract features, and make decisions.
- **Output Layer**: A softmax layer that generates the probability distribution across.

Value Network

The value network estimates the **expected future rewards** (i.e., the value of existing within a particular state). The value network helps the agent understand how good a state is and improves the stability of policy updates.

Like the policy network, the value network is also a neural network but with a different output: it outputs a scalar quantity denoting the expected cumulative reward of the current state.

3.4. Training the PPO Agent

PPO agent training is one of the critical aspects of this framework. The agent learns to detect vulnerabilities in the Online Boutique application by simulating real-world attack scenarios while interacting with the custom RL environment. The training process and the PPO algorithm are designed to ensure that the agent can effectively identify vulnerabilities while continuously improving over time.

3.4.1. Training Process and Algorithms

The PPO algorithm is employed to train the agent, enabling it to continuously improve its ability to detect vulnerabilities by simulating attacks. The training process consists of several phases: environmental interaction, reward collection, policy optimization, and agent evaluation.

1. Environment Interaction:

Initial Setup: The training starts with the PPO agent interacting with the custom RL environment (which simulates the Online Boutique application and its various attack vectors). The state of the environment reflects the security posture of the system, including the status of microservices, input validation status, and error logs.

Action Selection: At every timestep, the PPO agent chooses an action based on the current policy. The actions in this context are simulated attack strategies.

State Transition: Every action carried out by the agent causes a transition to a new state. For example, if the agent performs a DoS attack, the state changes to reflect the security vulnerability or lack of vulnerability in the targeted service.

2. Reward Calculation:

Reward Function: After each action, the environment provides feedback in the form of a reward. The reward is determined by the success or failure of the simulated attack. For example, if a DoS attack successfully crashes the service or results in higher resource utilization, then the agent receives a positive reward. If the system is resilient and the attack does not exploit any vulnerabilities, the agent receives a negative or zero reward.

Feedback Loop: The rewards guide the PPO agent towards actions that lead to successful vulnerability detection. The feedback loop involves adjusting the agent's policy so that it favors actions that have led to successful detection in the past.

3. Policy Optimization:

Objective Function: PPO optimizes the policy by maximizing the expected cumulative reward. It constitutes a **clipped surrogate objective function** that restricts substantially large policy updates, ensuring stable training and avoiding destabilizing changes to the agent's behavior.

Policy Update: During training, the policy network is updated using mini batches of data. PPO calculates the advantage of execution of an action at a given state and adjusts the policy network's weights accordingly. The policy is optimized using an **Actor-Critic method** [35],

where the **actor** modifies the policy in accordance with the chosen actions and rewards, while the **critic** assesses the actions' value (expected future reward).

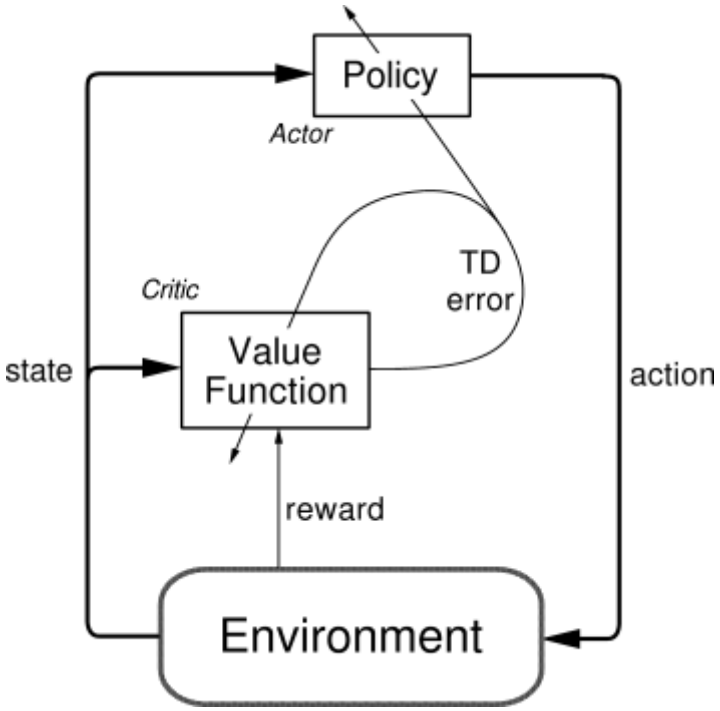

Figure 3.11 The overview of Actor-Critic method.

4. Training Evaluation:

Validation: After several episodes of interaction, performance of the agent is evaluated using metrics such as attack detection accuracy, false positives, and false negatives. This allows us to evaluate the agent's learning efficacy to identify vulnerabilities and whether it is improving over time.

3.4.2. Hyperparameters and Tuning

The training of the PPO agent requires the careful selection and tuning of various hyperparameters to optimize the learning process and ensure stable performance. These hyperparameters influence the learning rate, exploration strategies, and overall efficiency of the agent. Below are the key hyperparameters used in training the PPO agent:

1. Learning Rate

The learning rate regulates the extent to which the agent's policy and value network are modified at each iteration. An elevated learning rate may lead the agent to bypass excellent answers, whereas a diminished learning rate might impede the learning process.

2. Batch Size

The batch size specifies the quantity of experiences (state-action-reward transitions) gathered prior to the model's update. A larger batch size provides more data for learning, but it can be computationally expensive.

3. Clipping Parameter

The clipping parameter is an essential element of the **surrogate objective function** in PPO [36]. It guarantees that the policy updates remain moderate by constraining the likelihood ratio between the previous and current policies. This inhibits the agent from making substantial movements that could compromise training stability.

4. Value Function Loss Coefficient

The value function loss coefficient establishes the significance of the **value network**'s loss in the policy update process. This balances the importance of the **policy loss** (which tries to maximize rewards) and the **value loss** (which helps to estimate the value of states accurately).

5. Entropy Coefficient

The entropy coefficient regulates the agent's exploring behavior by promoting unpredictability in action choices. A higher entropy value promotes more exploration, which is useful in complex environments.

6. Discount Factor

The discount factor assesses the valuation of future benefits in relation to present rewards. An elevated discount factor prioritizes long-term results.

The hyperparameters are tuned iteratively through experimentation, with performance metrics guiding adjustments. Techniques like early stopping and learning rate schedules can also be employed to prevent overfitting and ensure faster convergence.

4. IMPLEMENTATION

4.1. System Architecture and Design

The core of the system is the PPO-based reinforcement learning agent that engages with the Online Boutique microservice environment through specialized attack modules. Here the agent receives state information from both the attack execution results and the service monitoring component, allowing it to learn effective attack strategies over time. This section explains how the PPO agent integrates into the Online Boutique application and the overall design.

Below is a diagram which illustrates the high-level architecture of the PPO-based security vulnerability detection framework.

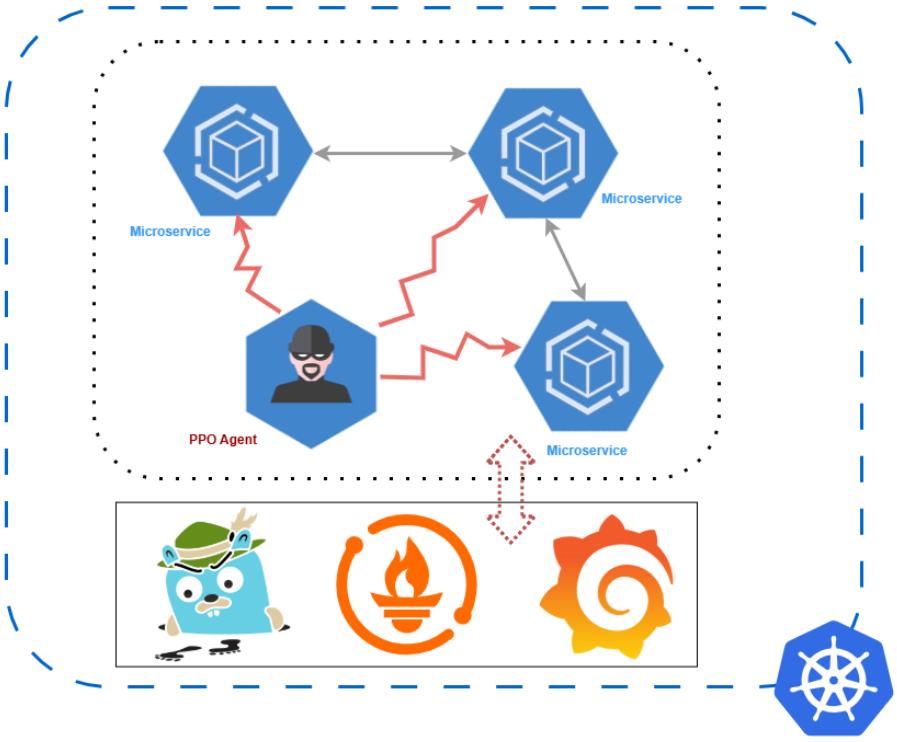

Figure 4.1 Architecture of the PPO-based security vulnerability detection framework.

The architecture of the detection framework consists of the following components.

- **PPO Agent**: Launches attack scenarios in the RL environment and adjusts its behavior based on rewards from attack success or failure.
- **RL Environment**: Interacts with the Online Boutique microservices to simulate real-world attack scenarios.
- **Microservices (Online Boutique)**: Includes the frontend, cart, product catalog and checkout services, orchestrated by Kubernetes, interacting with each other through APIs.
- **Monitoring Layer**: Consists of Prometheus, Jaeger and Grafana, captures key system metrics and logs attack simulations, providing data that helps evaluate the effectiveness of vulnerability detection.

The PPO-agent maintains a service discovery module that interfaces directly with the Kubernetes API to identify available microservices, their endpoints, and connection information. This enables dynamic targeting without requiring manual configuration for each service under test.

Data flows from the service discovery module to the attack agent, which selects appropriate parameters and executes attacks through specialized attack vector implementations. The monitoring module continuously communities with the Prometheus monitoring framework to gather service health during attacks, providing real-time feedback used to calculate rewards for the RL agent. All attack results and performance metrics are stored in a persistent database to enable continuous learning across sessions.

The PPO-agent is designed to operate within the Minikube cluster as a service, with appropriate credential configuration for API access.

4.2. PPO Model Implementation

PyTorch machine learning library is used as the software framework to implement PPO algorithms and the agent. As discussed in the previous sections, PPO is selected for its sample efficiency and stability. These are crucial for a system where each action involves relatively expensive attack execution.

4.2.1. State Space

State space of this implementation consists of six normalized continuous features.

Table 4.1 State Space Table

State Feature	**Description**	**Normalization / Value Range**
Baseline Response Time	Normalized response time of the service	0 - 1 (response time / 10s)
Baseline Service Availability	Availability of the service	0 - 1 (percentage / 100)
Target Service Type	Type of microservice being attacked	0 - 1 (normalized index)
Previous Attack Success	Whether the last attack was successful	{0, 1}
Previous Attack Intensity	Intensity of the last attack	0 - 1 (intensity / 1000)
Previous Attack Duration	Duration of the last attack	0 - 1 (duration / 60s)

This state representation captures both the current vulnerability context and historical attack information, enabling the agent to modify its strategy based on previous results.

4.2.2. Action Space

The action space comprises a combination of three distinct choice dimensions.

Table 4.2 Action Space Table

Action Feature	**Description**	**Values / Normalization**
Attack Type	Type of attack strategy	{0: HTTP Flood, 1: TCP Flood, 2: Slowloris, 3: Combined} (Categorical, encoded as index)
Intensity Level	Strength of the attack	{50, 100, 200, 350, 500, 750, 1000} (Discrete levels)
Duration	Duration of the attack	{10, 20, 30, 45, 60} seconds (Discrete levels)

These dimensions create a total action space of 140 possible attack configurations. This multi-dimensional space is flattened into a single discrete action space for the policy network.

4.2.3. Neural Network Architecture

The PPO implementation uses an actor-critic architecture with:

- **Shared feature extractor**: Two fully connected layers (64 and 32 neurons) with tanh activation
- **Actor head**: A policy layer outputting action probability through softmax activation
- **Critic head**: A value function estimator providing state-value predictions.

This architecture balances representational capacity with computational efficiency, allowing the agent to learn complex attack strategies while maintaining fast inference during operation.

4.2.4. PPO Implementation Details

PPO implementation uses the following hyperparameters.

Table 4.3 Hyperparameter table

Hyperparameter	**Description**	**Value**
Learning Rate	Size of a step for modifying policy weights	0.002
Discount Factor (γ)	Future reward consideration	0.99
GAE Parameter (λ)	Smoothing factor for advantage estimation	0.95
Clip Range (ε)	PPO clipping range to mitigate substantial updates	0.2
Value Function Coefficient	Weight of the value function loss in PPO loss	0.5
Entropy coefficient	Facilitates exploration by introducing entropy to loss	0.01
Mini-batch Size	Quantity of episodes per training batch	5 episodes
Training Epochs per Update	Number of times each batch is used for training	4

These parameters were selected based on empirical testing to balance exploration, stability, and learning speed for this security vulnerability detection task.

4.2.5. Attack Vector Implementation

PPO agent is designed to implement four distinct attack vectors, each targeting different vulnerability surfaces in microservice architectures:

4.2.5.1.HTTP Flood

The HTTP Flood attack generates high volumes of legitimate-looking HTTP requests to overwhelm application-layer resources. The implementation of this attack type is done in a way that it:

- dynamically selects endpoints from service configuration
- randomizes HTTP headers and parameters to bypass caching
- uses connection pooling for efficient request generation
- implements adaptive rate limiting based on intensity parameter
- simulates distributed sources using randomized X-Forwarded-For headers

The attack is particularly effective against stateful services with complex request processing logic, such as the frontend and checkout services.

4.2.5.2.TCP Flood

The TCP Flood attack targets the network layer by establishing large number of TCP connections without completing application-layer transactions. In this implementation the TCP flood, it:

- creates raw socket connections to service endpoints
- maintains connection state to prevent premature timeouts
- implements exponential backoff for failed connections
- uses adaptive timeout values based on observed response patterns
- employs connection reuse strategies for maximum efficiency

This attack vector is the most effective against services with limited connection pools, such as the checkout and shipping services.

4.2.5.3. Slowloris

The Slowloris attack exploits server-side connection handling by establishing connections and keeping them open with minimal data transfer. In the implementation of this attack it:

- opens legitimate HTTP connections with partial request headers
- periodically sends small header fragments to prevent timeout
- monitors connection health and replaces closed connections

- implements randomized timing patterns to avoid detection
- uses adaptive header selection based on observed server behavior

This attack is particularly effective against services using synchronous processing models with thread-per-connection architectures.

4.2.5.4.Combined Attack

The combined attack orchestrates all three attack vectors simultaneously with weighted intensity distribution, allowing more complex vulnerability testing. This implementation:

- allocates resources proportionally based on the intensity parameter (60% HTTP, 20% TCP, 20% Slowloris)
- synchronizes attack timing across vectors
- implements global rate limiting to prevent client-side resource exhaustion
- uses shared connection pools where appropriate
- provides unified metrics collection across all attack types

Managing resource contention between attack threads, ensuring accurate timing, and preventing the attacking system from self-limiting due to resource exhaustion were some of the challenges that faced during implementation.

4.2.6. Reward Function Design

The reward function is critical for guiding the RL agent toward effective attack strategies. A multi-component reward function that balances attack effectiveness with resource efficiency is designed for the reward function.

The primary reward component is based on attack success, derived from the confidence score returned by the analysis algorithm.

$$\mathrm{R}_{\mathrm{success}} = \text{success\_indicator} \times \left(\frac{confident\ score}{100}\right)$$

Where success_indicator is 1 if the attack caused significant service degradation (confidence > 40%) and 0 otherwise.

Efficiency is incorporated through a multiplicative factor.

$$efficiency\_factor = 1.0 - \left(\frac{duration}{60.0}\right) \times 0.5 - \left(\frac{intensity}{1000.0}\right) \times 0.5$$

This factor varies from 0.1 to 1.0, with elevated values corresponding to shorter, lower-intensity assaults.

The final reward is calculated using the below equation.

$$R = R_{success} \times (1.0 + efficiency\_factor)$$

The following are ensured by this formula.

1. Failed attacks receive zero reward regardless of efficiency.
2. Successful attacks with lower resource usage receive higher rewards.
3. The maximum reward possible is 2.0 (perfect success with minimum resources).

During PPO updates rewards are normalized within each batch to address reward scale stability. This helps to maintain consistent learning rates despite varying absolute reward values across different services.

4.2.7. Training Methodology

This framework follows an online training methodology where the agent learns continuously from actual attack execution rather than simulated environments. This approach ensures that the learned policy reflects real-world vulnerability patterns.

4.2.7.1.Experience Collection

The training process follows the steps below.

1. The agent selects attack parameters based on current policy.
2. The attack is executed against the target service.
3. Monitoring data is collected throughout the attack duration.
4. Success analysis determines the effectiveness of the attack.
5. Rewards are calculated and stored with the experience tuple.
6. After collecting five episodes, a policy update is performed.

This approach balances the need for diverse experience with the computational cost of policy updates.

4.2.7.2.Exploration Strategy

An adaptive ε-greedy strategy is implemented to encourage exploration of the large action space.

- Initial exploration rate $\varepsilon = 0.3$ (30% random actions)
- Decay schedule:

$$\varepsilon = max(0.05, \quad \varepsilon_{initial} \times decay\_factor^{episode})$$

- The decay factor is tuned to reach the minimum value after approximately 50 episodes.

This strategy ensures thorough exploration of different attack vectors and parameters during early training while gradually shifting toward exploitation of effective strategies.

4.2.7.3.Transfer Learning

A simple transfer learning approach is implemented to accelerate learning across different services.

- The same policy network is used across all services.
- The service type is encoded in the state representation.
- Experiences from all services contribute to the same experience buffer.
- The agent can leverage strategies learned from one service when attacking similar services.

This approach allows the agent to discover general vulnerability patterns applicable across multiple microservices while still learning service-specific weaknesses.

4.2.7.4.Convergence Criteria

The training process is considered to be converged when:

1. The average reward per episode stabilizes (moving average change < 5% over 20 episodes).
2. The attack success rate exceeds 80% for high value targets.
3. The policy entropy decreases below a threshold of 0.5, indicating confidence in selected actions.

In practice, it was observed that the initial convergence happens after 50-100 attack episodes, with continued refinement over longer training periods.

5. OBSERVATIONS, RESULTS AND ANALYSIS

This section delineates the results and discussion of the PPO-based security vulnerability detection framework implemented to identify the vulnerabilities on the Online Boutique application.

Prometheus monitoring application is deployed inside the Minikube cluster to monitor the states of the services within the cluster. It is also utilized within the PPO-agent to capture current state of the environment to decide the next actions.

```
PS C:\Users\kazun> kubectl get services -n monitoring
NAME                                      TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)                      AGE
alertmanager-operated                     ClusterIP   None            <none>        9093/TCP,9094/TCP,9094/UDP   23h
prometheus-grafana                        ClusterIP   10.101.20.191   <none>        80/TCP                       23h
prometheus-kube-prometheus-alertmanager   ClusterIP   10.111.117.94   <none>        9093/TCP,8080/TCP            23h
prometheus-kube-prometheus-operator       ClusterIP   10.102.89.187   <none>        443/TCP                      23h
prometheus-kube-prometheus-prometheus     ClusterIP   10.103.120.22   <none>        9090/TCP,8080/TCP            23h
prometheus-kube-state-metrics             ClusterIP   10.111.3.9      <none>        8080/TCP                     23h
prometheus-operated                       ClusterIP   None            <none>        9090/TCP                     23h
prometheus-prometheus-node-exporter       ClusterIP   10.101.80.117   <none>        9100/TCP                     23h
```

Figure 5.1 Prometheus services deployed in the cluster.

The following dashboard was created using Grafana to observe the results and the impact created by the attacks generated by the PPO-based agent in real-time. Prometheus was used as the data source for the Grafana.

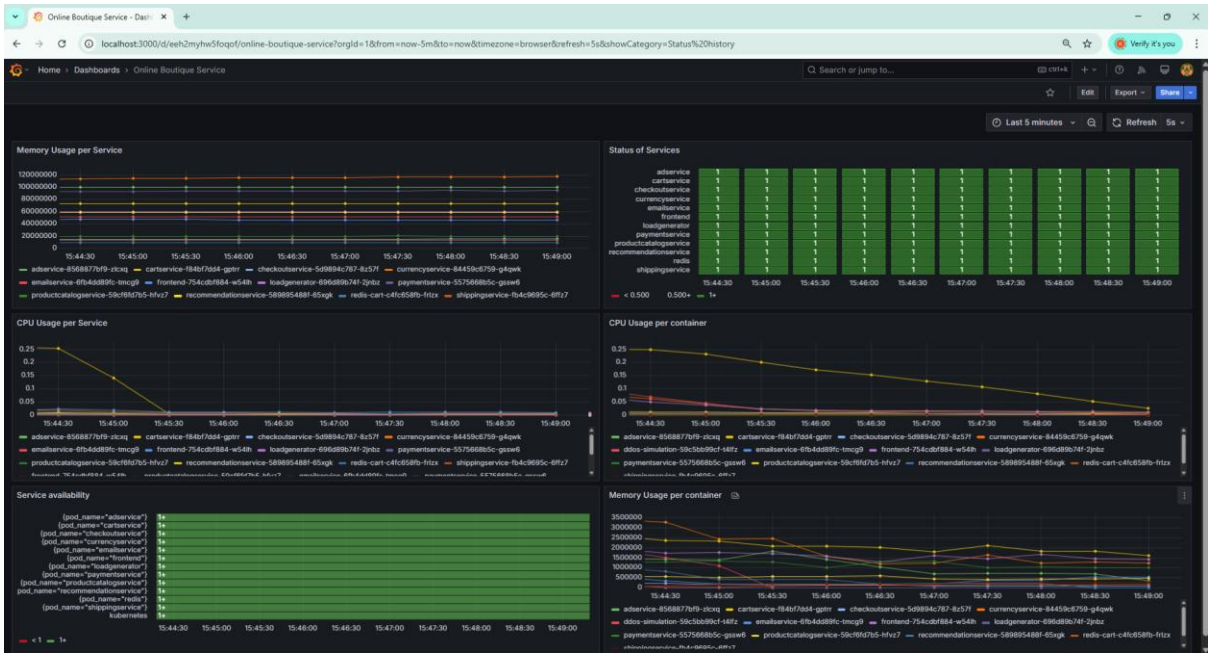

Figure 5.2 Grafana dashboard to monitor the status of the services.

PPO-based agent is deployed on the Minikube cluster identify vulnerabilities by launching attacks.

```
online-boutique   productcatalogservice-59cf6fd7b5-hfvz7   1/1   Running   1 (15h ago)   17h
online-boutique   recommendationservice-589895488f-65xgk   1/1   Running   1 (15h ago)   17h
online-boutique   redis-cart-c4fc658fb-frlzx               1/1   Running   1 (15h ago)   17h
online-boutique   shippingservice-fb4c9695c-6ffz7          1/1   Running   1 (15h ago)   17h
ppo-attack        ppo-ddos-attack-578c67c67c-fbq9z         1/1   Running   0             11s
```

Figure 5.3 PPO-based agent deployed in the Minikube cluster.

5.1. Observations for Individual Attacks

Individual independent attacks were performed on several services while monitoring the resource utilization metrics via Grafana to identify the PPO-based agent's ability to generate DoS attacks and identify potential vulnerabilities. The following graphs which were generated with the help of Grafana indicate how this service's CPU utilization and memory consumption varies over the time when PPO-based agent targets these services with DoS attacks.

Frontend Service (frontend**)**

This is the main user-facing application of the Online Boutique. It serves the frontend UI to users. Below graphs indicate how the CPU and memory usages were varied during the PPO agent's attack took place on the frontend service.

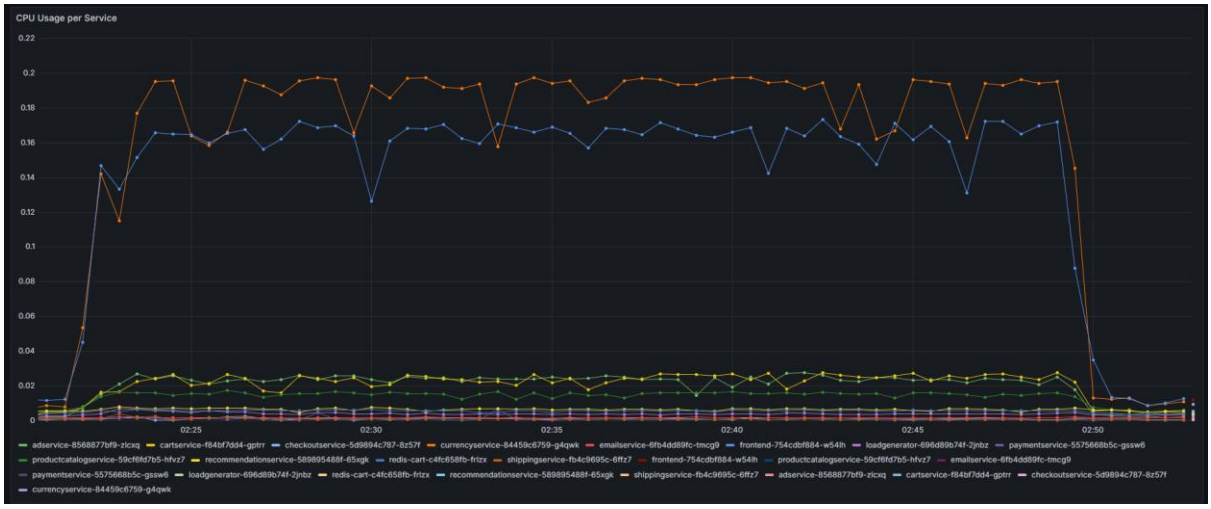

Figure 5.4 CPU utilization of the service during attack on the frontend serve.

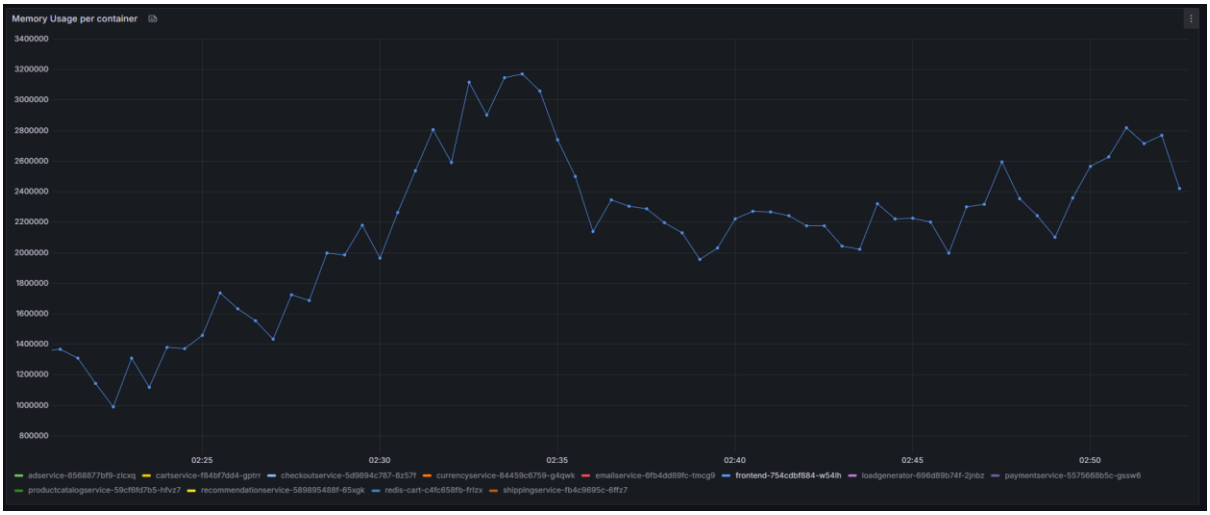

Figure 5.5 frontend service memory consumption during the attack.

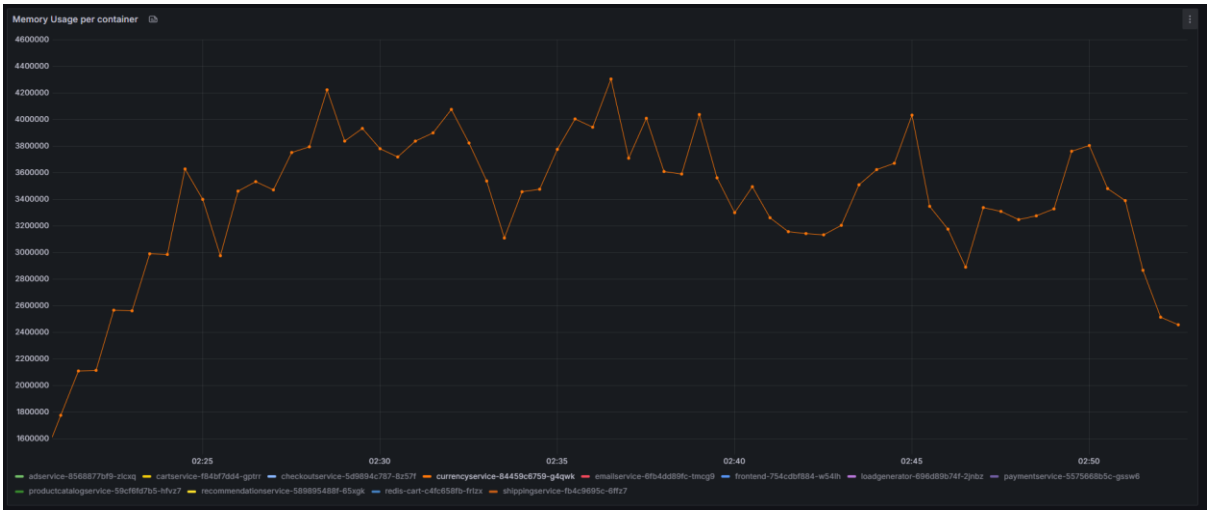

Figure 5.6 currencyservice memory consumption during the attack on frontend service.

As per the above graph, it's observed that when a DoS attack happens on the frontend service, it makes the frontend service consume huge amount of memory and CPU resources. Similarly, it's evident that it triggers a similar impact on the currencyservice as well. Also, the impact propagates to other downstream services as well.

Cart Service (cartservice)

This service takes care of handling the user's shopping cart. The CPU and memory utilization when the cartservice is under the DoS attack as shown in the below graphs.

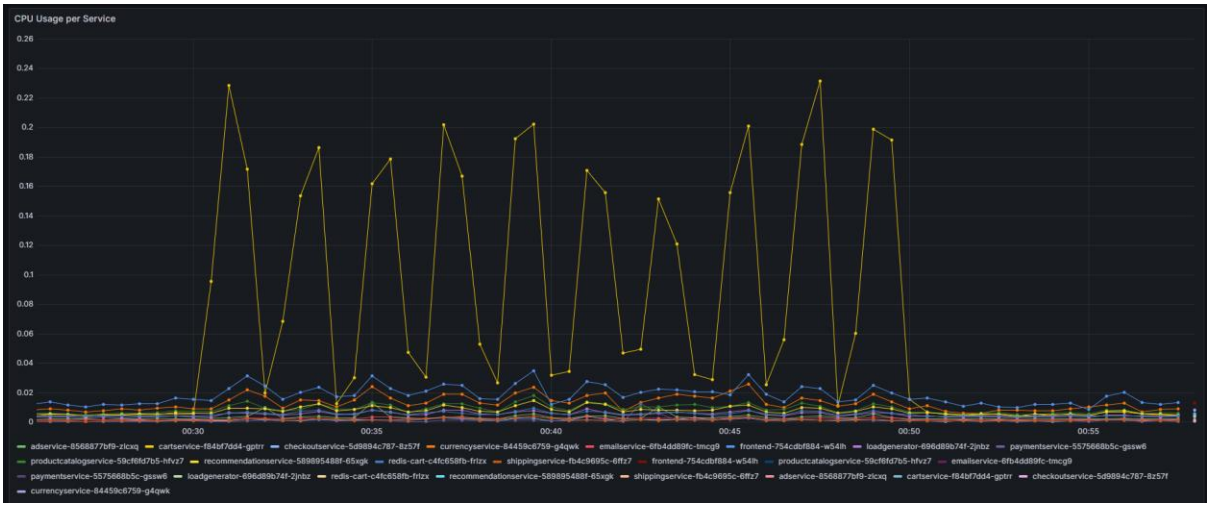

Figure 5.7 CPU utilization when cartservice is under attack.

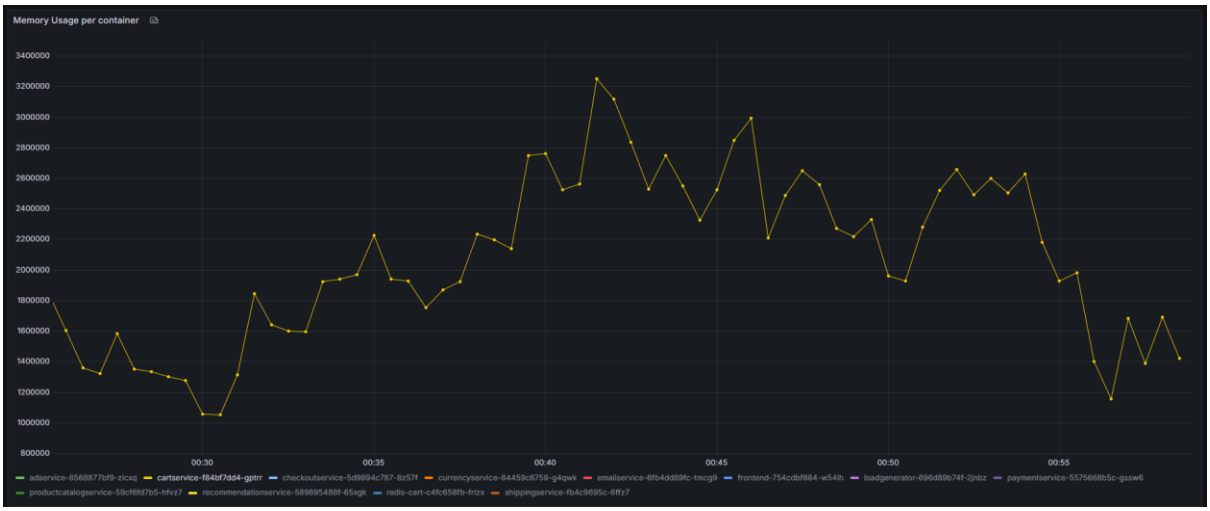

Figure 5.8 Memory consumption of the cartservice when it is under attack.

The impact of the attacks has caused huge fluctuations on the CPU utilization of the cartservice. But the impact has not propagated to the other services of the application.

Checkout Service (checkoutservice)

Checkout service oversees the checkout process of a user. Below shows how resources were impacted during the attack.

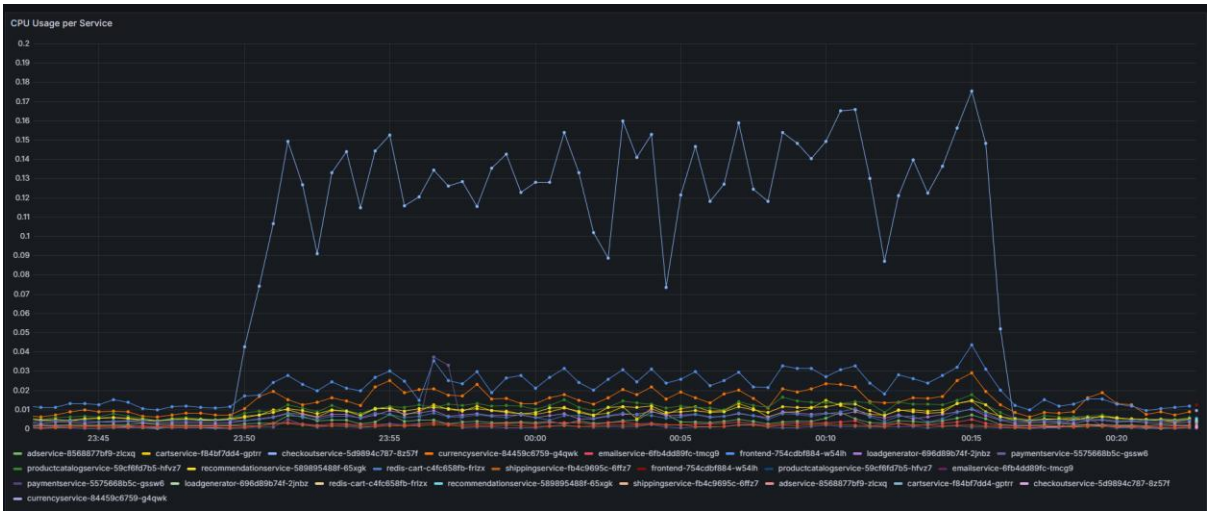

Figure 5.9 CPU utilization during attack on checkout service.

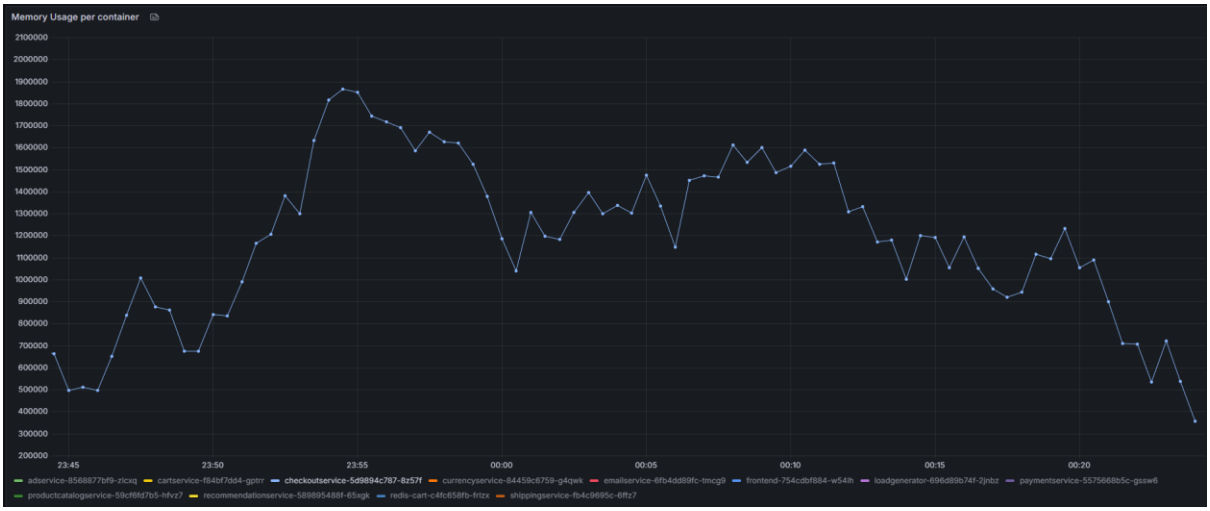

Figure 5.10 Memory usage of the checkout service during the attack.

PPO agent has been able to generate successful DoS attacks on the checkout service resulting in higher CPU and memory utilization. It was observed that there was significant impact on responsiveness of the checkout service from user interface side as well during the attack. It can be seen that the impact has been propagated on services like currencyservice and frontend services.

Product Catalog Service (productcatalogservice)

This service manages the catalog of products available for purchase. The impact on the resources is shown in the graphs below.

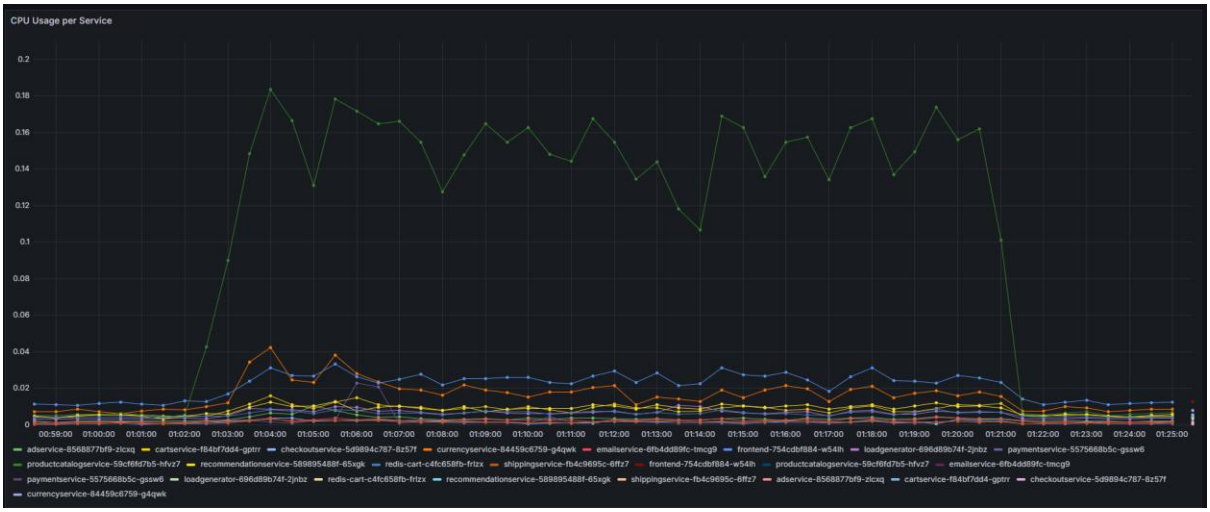

Figure 5.11 CPU utilization during the attack on productcatalog service.

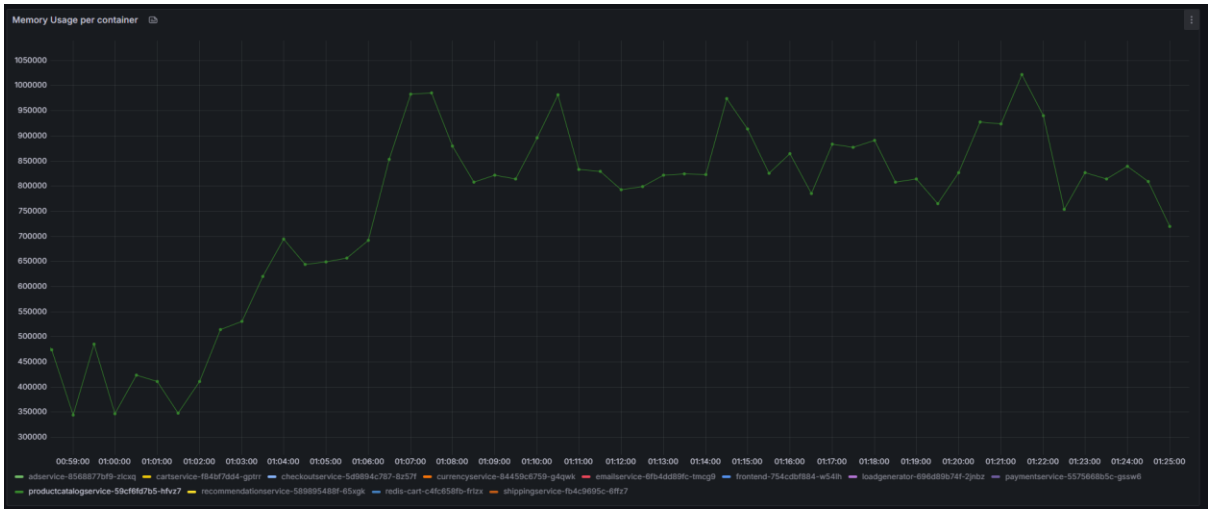

Figure 5.12 productcatalog service memory usage when it is under attack.

It is clear that when the productcatalog was targeted for the attack, the service consumed significant amounts of CPU and memory resources.

Below table summarizes the observations made in terms of resource utilization.

Table 5.1 Resource Utilization Before and During Attacks

Microservice	**CPU Usage (prior to attack)**	**CPU Usage (in the course of attack)**	**Memory Usage (prior to attack)**	**Memory Usage (in the course of attack)**
Frontend Service	6%	88%	14 MB	32 MB
Cart Service	3%	88%	82 MB	262 MB
Checkout Service	2.5%	87.5%	15 MB	56 MB
Product Catalog Service	2.5%	92%	24 MB	70 MB

5.2. Impact of Overall Attacks

Below table quantifies the impact of DoS attacks on each microservice in the Online Boutique application. This table provides a clear comparison of how different services respond under attack conditions, highlighting which components are most vulnerable to resource exhaustion attacks.

Metrics Explanation

- **CPU Usage (% change)**: Represents the percentage increase in CPU utilization during attack compared to baseline operation.
- **Memory Usage (% change)**: Shows the percentage increase in memory consumption during attacks. This metric helps identify services that may be susceptible to memory leaks or inefficient allocation under pressure.

- **Response Time (% change)**: Indicates how much slower the service becomes when under attack.

- **Request Success Rate (% change)**: Shows the reduction in successful requests during an attack. The negative values indicate decreases in reliability.

- **Pod Restarts**: Counts how many times Kubernetes needed to restart the service pods during the attack, indicating complete service failures.

Calculation Methodology

These metrics were calculated using the following approach:

1. **Baseline Measurements**: For each service, performance metrics were collected over a 10-minute period during normal operation. The mean values for CPU, memory, response time, and success rate were established as baselines.

2. **Attack Execution**: DoS attacks at intensity level 4 were executed against each service individually for 5 minutes using the PPO agent, which generated approximately 200 requests per second with 8KB payloads.

3. **Impact Calculation**:

   CPU and Memory changes were calculated using Prometheus metrics:

$$\frac{max\_during\_attack - baseline\_average}{baseline\_average} \times 100\%$$

   Response time changes were measured using distributed tracing data from Jaeger:

$$\frac{p95\_during\_attack - p95\_baseline}{p95\_baseline} \times 100\%$$

   Success rate changes were calculated by comparing the ratio of successful responses (HTTP 200-299) to total requests:

$$\frac{success\_rate\_during\_attack - baseline\_success\_rate}{baseline\_success\_rate} \times 100\%$$

4. **Pod Restart Detection**: Kubernetes events were monitored to track pod terminations and restarts that occurred during the attack period.

Table 5.2 Attack Impact Measurements

Service	**CPU Usage (% change)**	**Memory Usage (% change)**	**Response Time (% change)**	**Request Success Rate (% change)**	**Pod Restarts**
frontend	+187	+42	+325	-68	0
productcatalogservice	+156	+28	+243	-45	1
currencyservice	+192	+55	+278	-67	2
cartservice	+215	+65	+458	-92	3
checkoutservice	+145	+32	+267	-54	0
paymentservice	+178	+47	+312	-76	1
shippingservice	+75	+18	+125	-25	0
emailservice	+68	+12	+104	-18	0
recommendationservice	+123	+27	+198	-42	0
adservice	+88	+16	+132	-28	0

5.3. Impact Analysis

Above data reveals critical insights about the system resilience of the Online Boutique application.

1. **Vulnerable Components**: The cartservice shows the highest sensitivity to DoS attacks, with extreme degradation across all metrics.

2. **Resilient Services**: Services like emailservice and shippingservice showed relatively modest impacts, indicating better resource management or lower complexity.

3. **Failure Patterns**: The data shows that performance degradation typically manifests first in response time increases, followed by reduced success rates, and finally in pod restarts.

4. **Resource Consumption Profiles**: Services with higher CPU impact than memory impact (like frontend) likely has compute-bound bottlenecks, while services showing the opposite pattern (like cartservice) may have memory management issues.

6. DISCUSSION AND CONCLUSION

This section provides a concise overview of the principal results from the research and their practical relevance, an exploration of the limitations and challenges encountered, and suggestions for future research. Insights gained from this study contribute to the development of automated security testing frameworks using RL, specifically in the context of microservices architecture.

6.1. Discussion

The application of reinforcement learning techniques to security vulnerability detection in microservice architecture represents a novel approach to cybersecurity testing. This study focused specifically on DoS attacks against the Online Boutique microservices deployed in a Kubernetes environment, yielding several significant insights into both the effectiveness of RL-based security testing and the vulnerability profiles of microservice systems.

The Proximal Policy Optimization (PPO) algorithm demonstrated notable capability in identifying vulnerable components within the microservice architecture. As evidenced by the collected data, the RL agent progressively learned to target specific services that exhibited greater susceptibility to DoS attacks. The cartservice, for instance, showed significantly higher vulnerability with a 458% increase in response time during attacks and a 92% decrease in request success rate. This targeting behavior emerged naturally from the reward structure, where the agent received higher rewards for successful service disruptions.

The intensity-based approach to DoS attacks revealed important scaling effects in service resilience. Lower intensity attacks (levels 1-2) generally resulted in performance degradation without complete service failure, while higher intensities (levels 4-5) frequently triggered pod restarts and cascading failures. This graduated response provides valuable information for establishing realistic thresholds in monitoring systems and designing appropriate scaling policies.

Cross-service security vulnerability propagation presented particularly interesting patterns. When the frontend service was targeted, the impact frequently spread to downstream services, suggesting that proper isolation between services is critical for containing attack effects. Similarly, targeting the checkoutservice affected the currencyservice due to their dependency relationship. These propagation patterns underscore a fundamental security challenge in

microservice architectures: the interdependency of services can transform a vulnerability in one component into a system-wide issue.

The performance metrics collected during attacks offer a quantitative basis for prioritizing security improvements. CPU usage increases correlated strongly with response time degradation, highlighting compute-bound bottlenecks as primary vulnerability factors. Services with high memory usage changes during attacks (like cartservice at 65%) may suffer from inefficient memory management or resource allocation strategies that become particularly problematic under stress conditions.

The automated nature of the RL-based testing approach demonstrated several advantages over traditional security testing methods. The agent discovered vulnerable endpoints without prior knowledge of the application structure, simulating a real-world attacker's discovery process. Furthermore, it autonomously adapted its attack strategies based on observed outcomes, focusing effort on the most promising attack vectors and targets.

6.2. Study Limitations

Despite the encouraging results, this study experienced several limitations that should be recognized when interpreting the findings:

Limited Attack Vector: The most significant limitation was the exclusive focus on DoS attacks. While DoS represents a common threat to microservice architecture, it captures only one dimension of potential security vulnerability space. SQL injections, cross-site scripting (XSS), authentication bypass, and other attack vectors might reveal different vulnerability patterns across the same services.

Test Environment Constraints: The use of Minikube for local Kubernetes deployment, while practical for research purposes, may not fully represent the dynamics of production-scale deployments. Resource constraints in the test environment potentially influenced service behavior under attack, possibly exaggerating or understating actual vulnerability levels.

Simplification of Attack Mechanisms: The DoS attacks implemented focused primarily on request flooding and resource exhaustion. More sophisticated DoS techniques, such as application-layer attacks targeting specific vulnerabilities in the underlying frameworks, were not explored.

Reward Function Design: The reinforcement learning agent's effectiveness was heavily dependent on the reward function design. The current implementation rewarded service disruption but may not have adequately captured subtler forms of security compromise that would be valuable in a comprehensive security assessment.

Lack of Defensive Adaptation: The study did not include adaptive defensive measures, which would be present in real-world systems. Production environments would likely implement rate limiting, circuit breakers, and other protective mechanisms that might significantly alter the effectiveness of the tested attack vectors.

Single Application Architecture: Findings were based exclusively on the Online Boutique application, which, while representative of microservice architectural patterns, may exhibit specific vulnerabilities that do not generalize to all microservice systems.

6.3. Future Work

This paper suggests many intriguing directions for further investigation:

Expanded Attack Vector Coverage: Extending the reinforcement learning approach to include a broader range of attack vectors would provide a more comprehensive security vulnerability assessment. Implementing SQL injection, XSS, authentication bypass, and API fuzzing capabilities would enable the discovery of different vulnerability classes.

Adversarial Learning Framework: Developing a competitive learning environment where defensive agents evolve alongside offensive agents could yield insights into both security vulnerability detection and effective mitigation strategies. This could simulate the ongoing adversarial nature of cybersecurity more realistically.

Fine-grained Action Space: Refining the action space to include more specific attack parameters, such as payload structure, timing patterns, and header manipulation, could enable the discovery of more sophisticated security vulnerability exploits.

Integration with Continuous Security Testing: Incorporating the RL-based security vulnerability detection into CI/CD pipelines would enable continuous security assessment throughout the development lifecycle, potentially catching vulnerabilities before deployment.

Cross-architecture Validation: Testing the approach against different microservice architectures and application domains would help establish the generalizability of both the methodology and the vulnerability patterns observed.

Automated Mitigation Generation: Extending the system to automatically suggest or implement mitigations for discovered vulnerabilities would close the security testing loop, moving from detection to remediation.

Transfer Learning for Security Testing: Exploring whether an agent trained on one microservice architecture could transfer its learning to identify vulnerabilities more quickly in different architectures would have significant practical implications for security testing efficiency.

Real-time Attack Detection: Using the insights from this research to develop better detection systems that can identify attack patterns in their preliminary stages before significant service disruption occurs.

6.4. Conclusion

This research has demonstrated the feasibility and effectiveness of using reinforcement learning, specifically the PPO algorithm, for automated DoS security vulnerability detection in microservice architectures by leveraging dynamic analysis principles. The approach successfully identified service vulnerabilities, quantified their impact, and revealed propagation patterns across service dependencies.

The findings highlight several key insights for microservice security. First, service interdependencies create security vulnerability propagation paths that must be considered in security design. Second, different services exhibit varying resilience to DoS attacks, with stateful services like cartservice showing particular susceptibility. Third, the automated discovery capabilities of RL agents provide an effective simulation of adversarial behavior, enabling more thorough security testing.

The quantified impact measurements provide concrete evidence for prioritizing security improvements, with services showing response time increases exceeding 200% or success rate decreases greater than 50% representing critical vulnerabilities requiring immediate attention.

While this study focused exclusively on DoS attacks, the methodology establishes a foundation for more comprehensive automated security testing of microservice systems. The combination of reinforcement learning with a diverse array of attack vectors could significantly enhance the security posture of microservice architecture, addressing one of the primary challenges in modern distributed systems.

As organizations continue to adopt microservice architectures, automated and adaptive security testing approaches will become increasingly valuable. This research represents a step toward security testing methodologies that can keep pace with the complexity and rapid evolution of microservice systems.

REFERENCES

[1] L. De Lauretis, “From Monolithic Architecture to Microservices Architecture,” *IEEE Xplore*, Oct. 01, 2019. https://ieeexplore.ieee.org/abstract/document/8990350.

[2] V. Velepucha and P. Flores, "A survey on microservices architecture: Principles patterns and migration challenges", *IEEE Access*, vol. 11, pp. 88339-88358, 2023.

[3] Baškarada, S., Nguyen, V., & Koronios, A. (2018). Architecting Microservices: Practical Opportunities and Challenges. *Journal of Computer Information Systems*, *60*(5), 428–436. https://doi.org/10.1080/08874417.2018.1520056

[4] P. Billawa, A. B. Tukaram, N. E. D. Ferreyra, J.-P. Steghöfer, R. Scandariato, and G. Simhandl, “SOK: Security of Microservice Applications: A Practitioners’ perspective on challenges and best practices,” *Proceedings of the 17th International Conference on Availability, Reliability and Security*, pp. 1–10, Aug. 2022, doi: 10.1145/3538969.3538986.

[5] U. Zdun *et al.*, “Microservice Security Metrics for secure communication, identity management, and observability,” *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 1, pp. 1–34, May 2022, doi: 10.1145/3532183.

[6] B. Soundararajan, “Secure configuration Management for microservices architecture,” *International Journal of Multidisciplinary Research and Growth Evaluation*, vol. 1, no. 1, pp. 110–114, Jan. 2020, doi: 10.54660/.ijmrge.2020.1.1.110-114.

[7] Cerny, T.; Svacina, J.; Das, D.; Bushong, V.; Bures, M.; Tisnovsky, P.; Frajtak, K.; Shin, D.; Huang, J. “On code analysis opportunities and challenges for enterprise systems and microservices,” *IEEE Journals & Magazine | IEEE Xplore*, 2020. https://ieeexplore.ieee.org/abstract/document/9179733

[8] L. P. Kaelbling, M. L. Littman, and A. W. Moore, “Reinforcement Learning: a survey,” *Journal of Artificial Intelligence Research*, vol. 4, pp. 237–285, May 1996, doi: 10.1613/jair.301.

[9] S. H. Oh, J. Kim, J. H. Nah, and J. Park, “Employing deep reinforcement learning to Cyber-Attack simulation for enhancing cybersecurity,” *Electronics*, vol. 13, no. 3, p. 555, Jan. 2024, doi: 10.3390/electronics13030555.

[10] A. Ganje, “Microservices in organizations,” *Journal of Software Engineering and Applications*, vol. 18, no. 02, pp. 76–86, Jan. 2025, doi: 10.4236/jsea.2025.182005.

[11] A. Katal, P. Prasanna, R. Birla, and N. Kunal, “Evolution from Monolithic to Microservices Architecture: A New Era in Software Architecture,” in *Springer tracts in nature-inspired computing*, 2025, pp. 235–279. doi: 10.1007/978-981-96-0706-8_12.

[12] O. Kumar and A. Narang, “Securing Microservices: Challenges and solutions,” Jan. 26, 2025. https://ijircstjournal.org/index.php/ijircst/article/view/50

[13] E. Safeer, “Reinforcement learning approaches in cyber security,” in *Advances in information security, privacy, and ethics book series*, 2024, pp. 53–76. doi: 10.4018/979-8-3693-5415-5.ch002.

[14] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal Policy optimization Algorithms,” *arXiv.org*, Jul. 20, 2017. https://arxiv.org/abs/1707.06347

[15] C. Henríquez, J. D. R. Valencia, and G. S. Torres, “Architectural Evolution at Netflix: A Case Study on Microservices and the Transformation from Monolithic to Scalable Systems.,” *ojs.uac.edu.co*, Mar. 2025, doi: 10.15665/rp.v23i1.3683.

[16] S. Bhatnagar and R. Mahant, “Overview of microservices,” in *Apress eBooks*, 2025, pp. 53–134. doi: 10.1007/979-8-8688-1267-5_2.

[17] A. Shukla, “Exploring the integration of APIs in microservices for scalable application development,” Jan. 18, 2025. https://jrtcse.com/index.php/home/article/view/JRTCSE.2025.13.1.4

[18] D. Ordonez-Camacho, “Reduciendo la brecha de seguridad del IoT con una arquitectura de microservicios basada en TLS y OAuth2,” *Ingenius*, no. 25, pp. 94–103, Dec. 2020, doi: 10.17163/ings.n25.2021.09.

[19] G. Somashekar and A. Gandhi, “Towards optimal configuration of microservices,” in Proceedings of the 1st Workshop on Machine Learning and Systems, 2021, pp. 7–14.

[20] “A Systematic literature review of Inter-Service security threats and mitigation strategies in microservice architectures,” *IEEE Journals & Magazine | IEEE Xplore*, 2024. https://ieeexplore.ieee.org/abstract/document/10540127

[21] J. Clifton and E. Laber, “Q-Learning: Theory and applications,” *Annual Review of Statistics and Its Application*, vol. 7, no. 1, pp. 279–301, Mar. 2020, doi: 10.1146/annurev-statistics-031219-041220.

[22] J. Fan, Z. Wang, Y. Xie, and Z. Yang, “A theoretical analysis of deep Q-Learning,” *PMLR*, Jul. 31, 2020. https://proceedings.mlr.press/v120/yang20a

[23] C. Wu, W. Bi, and H. Liu, “Proximal policy optimization algorithm for dynamic pricing with online reviews,” *Expert Systems With Applications*, vol. 213, p. 119191, Nov. 2022, doi: 10.1016/j.eswa.2022.119191.

[24] M. Lopez-Martin, B. Carro, and A. Sanchez-Esguevillas, “Application of deep reinforcement learning to intrusion detection for supervised problems,” *Expert Systems With Applications*, vol. 141, p. 112963, Sep. 2019, doi: 10.1016/j.eswa.2019.112963.

[25] J. Schwartz and H. Kurniawati, “Autonomous Penetration Testing using Reinforcement Learning,” *arXiv.org*, May 15, 2019. https://arxiv.org/abs/1905.05965

[26 T. Zhang, “Information Security Challenges in FinTech: Strategies for Protecting Digital Assets,” Sep. 23, 2024. https://www.pioneerpublisher.com/jpeps/article/view/1000

[27] R. Mazzolin, A. Madni, “A survey of contemporary cyber security vulnerabilities and potential approaches to automated defence,” *IEEE Conference Publication | IEEE Xplore*, Aug. 24, 2020. https://ieeexplore.ieee.org/abstract/document/9275828

[28] S. Chugh, “Bridging the gap: industry perspectives and trends in cloud security, and opportunities for collaborative research,” *IEEE Conference Publication | IEEE Xplore*, Nov. 01, 2023. https://ieeexplore.ieee.org/abstract/document/10431562

[29] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, “Trust Region Policy optimization,” *PMLR*, Jun. 01, 2015. https://proceedings.mlr.press/v37/schulman15.html

[30] F. Sangoleye, J. Johnson and E. Eleni Tsiropoulou, “Intrusion detection in industrial control systems based on deep reinforcement learning,” *IEEE Journals & Magazine | IEEE Xplore*, 2024. https://ieeexplore.ieee.org/abstract/document/10713374

[31] A. d. Rio, D. Jimenez and J. Serrano, “Comparative analysis of A3C and PPO algorithms in Reinforcement Learning: A survey on general environments,” *IEEE Journals & Magazine | IEEE Xplore*, 2024. https://ieeexplore.ieee.org/abstract/document/10703056

[32] S. De Paoli and J. Johnstone, “A qualitative study of penetration testers and what they can tell us about information security in organisations,” *Information Technology and People*, vol. 38, no. 1, pp. 380–398, Oct. 2023, doi: 10.1108/itp-11-2021-0864.

[33] “Manual and automated penetration testing. Benefits and drawbacks. Modern tendency,” *IEEE Conference Publication | IEEE Xplore*, Feb. 01, 2016. https://ieeexplore.ieee.org/abstract/document/7452095

[34] M. Sewak, “Actor-Critic models and the A3C,” in *Springer eBooks*, 2019, pp. 141–152. doi: 10.1007/978-981-13-8285-7_11.

[35] V. R. Konda, J. N. Tsitsiklis, Laboratory for Information and Decision Systems, and Massachusetts Institute of Technology, “Actor-Critic algorithms,” journal-article. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/1999/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf

[36] K. Lange, D. R. Hunter, and I. Yang, “Optimization transfer using surrogate objective functions,” *Journal of Computational and Graphical Statistics*, vol. 9, no. 1, pp. 1–20, Mar. 2000, doi: 10.1080/10618600.2000.10474858.